The Micropayment Internet

Teneo

Teneo.io

Contents

Introduction

An autonomous agent needs a GPU slot in roughly 200 milliseconds. It has no prior relationship with the provider. No human is available to approve the purchase, verify the invoice, or check whether the delivered compute matches what was promised. The agent must discover a suitable machine, open a payment channel or route through an existing one, authenticate itself without a username or password, stream fractional payments as it consumes cycles, and retain cryptographic proof that every milisatoshi it spent corresponded to a verified unit of work. If any step fails, the entire transaction must revert cleanly. If the provider cheats, the agent must have a dispute path that does not depend on trust or reputation.

This is not a thought experiment. Every component I just described exists in deployed infrastructure today. State channels lock collateral in 2-of-2 multisig outputs and update balances off-chain through monotonically increasing commitment transactions. Hash time-locked contracts bind payment release to cryptographic preimage revelation, making multi-hop routing atomic. Macaroon credentials attenuate authorization through chained caveats without requiring a round trip to an issuing server. The L402 protocol ties an HTTP 402 challenge to a Lightning payment hash, producing a bearer credential that proves payment and grants access in a single flow. Probabilistic pathfinding algorithms route payments across incomplete network topologies under fee and liquidity constraints.

The pieces exist. The composition does not.

I have spent years working at the intersection of payment infrastructure, cryptographic verification, and compute market design, and the bottleneck I keep encountering is never the intelligence of the agent or the denomination of the money. It is the absence of a coherent settlement and coordination architecture that binds authorization, payment,

and service delivery into a single atomic loop. Without that binding, every autonomous purchasing attempt degrades into one of two failure modes: either a human must supervise the transaction, which destroys the latency and cost advantages of automation, or the agent operates unsupervised against unverified counterparties, which is an unacceptable security posture at any scale.

The public conversation around machine-to-machine commerce is dominated by two framings, and both are wrong in ways that matter for engineering.

The first frames autonomous commerce as primarily an **AI problem**. Build a smarter agent, the argument goes, and it will figure out how to buy things. But intelligence does not produce atomicity. No amount of language model capability gives an agent a cryptographic guarantee that a provider will deliver the compute it paid for, or that a payment will revert if delivery fails. Settlement guarantees come from protocol design: HTLC preimage revelation, dispute windows with stale-state penalization, timelocked fallback paths. These are mechanism properties, not emergent behaviors of sufficiently capable models.

The second framing insists that machine commerce requires a **new monetary system**, some purpose-built token or novel consensus mechanism. It does not. The Lightning Network already supports streaming micropayments at sub-second latency with fees that are negligible for the transaction sizes autonomous agents will generate. What is missing is not a better currency. What is missing is the protocol composition that connects a payment channel to an authentication credential to a service delivery verification to a budget enforcement policy, all operating under adversarial assumptions and bounded latency.

There is a third misconception worth naming: that **payment channels are consumer technology**, useful for buying coffee or tipping content creators but irrelevant to infrastructure procurement. The opposite is true. Channels are low-latency coordination rails. Their natural application is high-frequency, low-value exchange between machines that cannot afford the cost or delay of on-chain settlement for every interaction. A single channel between an agent and a compute provider can support thousands of incremental state updates per second, each one a signed balance adjustment that is enforceable on-chain but never needs to touch the base layer unless something goes wrong.

The paradigm this book installs is direct. **Autonomous machine commerce is a settlement and coordination problem** solved by composing cryptographic primitives under bounded latency constraints and

incentive-compatible market design. That sentence is the interpretive key to every chapter that follows.

Once you adopt this framing, the architecture becomes legible as a layered stack. At the base, payment channels and HTLCs provide atomic, off-chain settlement with on-chain fallback. Above that, macaroons and L402 bind payment to authorization, creating credentials that are simultaneously proof of payment and proof of access rights, attenuable without server contact and expirable by design. Above that, routing and discovery protocols let agents find providers and establish payment paths across multi-hop topologies without centralized registries. Above that, game-theoretic pricing mechanisms, congestion surcharges, and auction designs stabilize compute markets against demand spikes and strategic manipulation. Above that, budget hierarchies and spending policies give principals verifiable control over agent expenditure under adversarial conditions. And threading through every layer, zero-knowledge proofs compress settlement data and enable compliance attestation without disclosing individual transaction details.

Each layer depends on the one below it. Strip out any layer and the system either stops functioning or reverts to human supervision. This is why no treatment of a single component, no matter how thorough, has produced the engineering clarity that builders need. You cannot design a budget policy without understanding the channel mechanics it constrains. You cannot evaluate a discovery protocol without knowing the settlement flow it must initiate. You cannot reason about cross-chain atomicity without tracing the HTLC timelock decrements across heterogeneous finality windows.

The transformation this book delivers is specific. You will move from fragmented familiarity with individual primitives to the ability to trace a complete agent purchasing loop, from service discovery through channel establishment, L402 authentication, streaming settlement, and post-hoc audit, identifying the security assumptions, failure modes, and fallback paths at every protocol boundary. You will learn to design budget hierarchies that constrain autonomous agents even when the agents operate across multiple providers, chains, and pricing regimes. You will be able to evaluate cross-chain settlement topologies for atomicity guarantees and finality risk with quantitative precision. You will understand congestion pricing and auction theory well enough to assess whether a compute procurement market has stable equilibria or is vulnerable to strategic manipulation.

This is not inspiration. It is compositional engineering literacy for a protocol stack that is ready to be built.

The structure follows the stack. We begin with payment channel fundamentals, because every flow in this architecture terminates in a channel state update or an on-chain enforcement transaction. If the settlement layer is unclear, nothing above it can be reasoned about with confidence. Chapter 1 opens with the 2-of-2 multisig funding transaction, walks through off-chain state updates via bilateral signing and sequence number enforcement, and closes with the dispute resolution mechanics that make the entire off-chain edifice trustworthy: challenge periods, stale-state penalization, and the distinction between cooperative and unilateral channel closure. Every subsequent chapter builds on the invariants established there, and by the final chapter you will see the full loop executed end to end, with budget constraints, cross-chain paths, and audit trails operating as a unified control system.

Machines are already trying to buy things from each other. They are failing not because payments are hard, but because no one has specified the full protocol composition that makes autonomous purchasing verifiable, atomic, and economically stable. That specification starts on the next page.

Chapter One

Payment Channel Fundamentals and the Lightning Network Precedent

Lightning Network was built to make coffee purchases viable on Bitcoin. Autonomous agents need it for an entirely different reason. The throughput constraints that motivated human-scale payment channels turn out to be secondary compared to a deeper structural property: a payment channel is a bilateral state machine with cryptographic finality between updates, enforceable dispute resolution, and no per-transaction settlement cost once collateral is locked. Every on-chain transaction carries latency and fee overhead that makes sub-cent exchange economically absurd, yet autonomous procurement loops will need to clear millions of micro-purchases per hour across counterparties that share no trust relationship. The mechanism that resolves this contradiction was deployed years before anyone framed it as a machine commerce problem.

This chapter establishes the complete framework for understanding payment channels as formally specified state machines. We will trace the full lifecycle from funding transaction construction through bilateral state progression to dispute resolution, and decode how HTLC-based atomicity extends bilateral channels into conditional payment paths that span multiple hops. The goal is mechanical precision: collateral require-

ments, update invariants, adversarial close paths, challenge windows, and the penalty structures that make cheating economically irrational. What emerges is the load-bearing settlement primitive beneath every autonomous procurement flow this book constructs.

That lifecycle begins with a single irreducible commitment: locking funds into a jointly controlled output that neither party can unilaterally spend. How the funding transaction is constructed, what its 2-of-2 multisig structure enforces, and how initial balances are allocated determine every security property the channel will ever have.

On-Chain Funding Transactions, 2-of-2 Multisig Collateral Locking, and Initial Balance Allocation

Before a single off-chain state update can propagate, before any micropayment settles in microseconds instead of minutes, one party must absorb the full cost of on-chain finality. This is the foundational paradox of payment channel design: a system engineered to minimize transaction cost begins with an irreducible expensive transaction, and that expense is not overhead but the very act that purchases every subsequent security guarantee.

The structure of this founding commitment determines far more than initial balances. Collateral locked into a jointly controlled output defines channel capacity ceilings, establishes a hard liveness dependency between counterparties, and creates an asymmetry between the party who funds and the party who receives. The funder pays all on-chain fees and initially holds all capacity on one side, while the counterparty contributes nothing yet gains the ability to receive. That imbalance is not incidental. It shapes routing viability, inbound liquidity constraints, and the economic calculus of who rationally opens channels in the first place. And because both keys must sign to move funds, the same construction that eliminates reliance on any third party introduces a new failure surface: if either participant goes offline, the locked collateral becomes inaccessible until a timeout or forced closure resolves the impasse.

Understanding how this single on-chain act constrains every subsequent protocol operation is the prerequisite for reasoning about state advances, dispute resolution, and channel lifecycle economics with any precision.

Why Every Payment Channel Begins with a Single On-Chain Commitment: The Cost of Bootstrapping Trust

Two parties with no prior relationship and no shared institutional context cannot transact off-chain until they have established a single point of mutual, independently verifiable truth. That point is the on-chain funding transaction. Every payment channel begins here, not because the protocol designer lacked imagination, but because this is the irreducible cost of bootstrapping trust between strangers using cryptographic commitment rather than counterparty reputation.

The funding transaction publishes a UTXO to the blockchain that both participants can verify against the same consensus state. This output is the shared reference point from which all subsequent off-chain updates derive their enforceability. Without it, any claim one party makes about a balance owed is just a message, indistinguishable from a lie. The blockchain's role is surgical and brief: it witnesses the creation of the UTXO, confirms it into a block, and then steps aside. From that moment forward, the two parties update their balances privately, bilaterally, and at whatever speed their network connections permit. But every one of those private updates is meaningful only because both parties know the funding UTXO exists and can be spent according to rules they both committed to at channel creation.

This single on-chain step imposes three costs that define the economics of trust bootstrapping. First, the transaction fee at prevailing fee rates, which can range from a few hundred satoshis during quiet mempools to tens of thousands during congestion. Second, confirmation latency. A channel is not safe to use until the funding transaction is buried under enough blocks to make reorganization economically implausible, typically around three to six confirmations. Third, and most persistent, is capital opportunity cost. The funds locked into the channel cannot be deployed elsewhere for the channel's entire lifetime. These costs are sunk. They are recovered only if the channel processes enough off-chain payments to amortize them below the cost of settling each payment individually on-chain.

The output of the funding transaction is not sent to either party's address. It is sent to a jointly controlled script, ensuring that neither participant can unilaterally sweep the funds before the channel begins operating. This bilateral control invariant is established from the first on-chain byte and persists through every state update. If the funds were sent to a single party's key, the entire off-chain construction would col-

lapse. There would be no leverage to enforce honest behavior and no penalty surface for cheating. Joint control is not a design preference. It is a structural requirement.

Consider an autonomous agent provisioning GPU inference from a compute provider it has never interacted with before. The agent's decision to open a channel is not a setup step to rush through. It is a strategic bet. The agent is committing real capital and real fees against a forecast that the volume of per-request payments to this specific provider will exceed the sunk bootstrapping cost. Channel creation is a threshold function: below some anticipated interaction density, the on-chain cost dominates and the agent should pay on-chain or not transact at all. Above that threshold, the amortized per-payment cost drops toward zero, and the channel becomes the only rational settlement path. This calculation, not any protocol ceremony, is what determines whether a channel gets opened.

The irreducibility of this on-chain commitment is precisely what motivates every optimization explored later: constructions that share a single on-chain transaction across many channels, batched openings that reduce per-channel fee burden, and factory architectures that amortize the trust-bootstrapping cost across dozens of participant pairs. None of these eliminate the requirement. They redistribute it. The funding transaction remains the load-bearing structure, and every micropayment that flows through the channel afterward rests on the cryptographic anchor it created. What remains to be specified is the exact script structure governing that locked output and the properties it must satisfy before either party commits a signature.

Constructing the 2-of-2 Multisig Output: Script Structure, Key Aggregation, and the Liveness Assumption

In early 2018, a developer working on one of the first Lightning Network implementations lost a small but nonzero amount of bitcoin by broadcasting a funding transaction before obtaining a countersigned commitment transaction from her channel partner. The funds landed in a 2-of-2 multisig output that required both parties' signatures to spend, and her counterparty's node had already gone offline. No dispute mechanism could help because no off-chain state had ever been constructed against that output. The satoshis sat locked, irrecoverably, in a UTXO that neither party could move alone. This episode illustrates, with uncomfortable clarity, why understanding the script-level anatomy of the funding output is not optional. The multisig construction is the crypto-

graphic root from which every subsequent channel operation derives its authority.

A 2-of-2 multisig output, in its legacy form, is expressed through a witness script that places two public keys on the stack and invokes OP_CHECKMULTISIG with a threshold parameter of two. The spending transaction must supply two valid signatures, one corresponding to each key. The funding transaction's output commits to the hash of this witness script, creating a pay-to-witness-script-hash (P2WSH) output whose redeem conditions are invisible to the network until the output is spent. Each commitment transaction that the channel participants later construct will reference this output by its outpoint, the combination of the funding transaction's identifier and the output index. This outpoint becomes the singular UTXO anchor for all off-chain state. Every balance update, every conditional payment, every dispute path traces its lineage back to this single locked output.

Taproot changes the surface without altering the invariant. Under MuSig2 key aggregation, the two channel participants combine their individual public keys into a single aggregate key through a multi-round nonce exchange and coefficient derivation process. The resulting output is a standard pay-to-taproot (P2TR) output, a 32-byte witness program indistinguishable on-chain from any ordinary single-signer spend. This collapse carries three concrete consequences. On-chain footprint drops because the spending witness contains a single 64-byte Schnorr signature rather than two ECDSA signatures plus the redeem script. Fee cost falls proportionally. And privacy improves because chain observers cannot determine whether an output was controlled by one party or two, nor whether a channel ever existed.

The choice of exactly two signers is not a design convenience. It is the minimal quorum that enforces bilateral consent without introducing a trusted third party. A 1-of-2 threshold would allow either participant to drain the channel unilaterally, destroying the safety property entirely. A 2-of-3 arrangement introduces a third key holder whose cooperation or compromise becomes a new attack surface, reintroducing the institutional dependency that the construction exists to eliminate. Two-of-two is the only configuration in which every state transition requires the explicit agreement of both economic stakeholders and no one else. This bilateral veto is what makes the channel safe during cooperative operation.

But that same bilateral requirement encodes a constraint the protocol cannot hide. If both signatures are needed to advance the channel's

state, then one party's silence is sufficient to freeze it. Unresponsiveness is not a nuisance to be handled by retry logic. It is a protocol-level event. When a counterparty stops signing, no new commitments can be constructed, no payments can be routed, and the channel's utility drops to zero until the responsive party exercises the dispute path by broadcasting the most recent commitment transaction on-chain. The 2-of-2 construction thus purchases safety through a liveness assumption: normal operation presumes both participants remain available and cooperative. This trade-off is deliberate. It converts counterparty failure from an ambiguous operational concern into a crisp, detectable condition that downstream mechanisms, including timelocked dispute transactions, are specifically engineered to resolve.

The ordering constraint that the 2018 incident exposed remains load-bearing. Before any satoshis move into the funding output, both parties must have already exchanged public keys, constructed and countersigned at least one commitment transaction spending from the anticipated funding outpoint, and verified that a valid unilateral close path exists. Only then is the funding transaction safe to broadcast. Key generation, key exchange, commitment construction, and funding must execute in precisely this sequence. Any reordering risks permanent fund loss, not because of a software bug, but because the 2-of-2 lock does exactly what it promises: it refuses to release funds without two signatures, regardless of whether a signed spending path was ever created.

Initial Balance Allocation and the Asymmetry Between Funder and Fundee Capacity

An autonomous procurement agent commits 500,000 satoshis to a new channel with a compute provider. The funding transaction confirms on-chain, locking the full amount into the 2-of-2 multisig output. At this instant, every satoshi of that channel's capacity sits on the funder's side of the ledger. The provider, despite being party to a live channel backed by real collateral, holds zero outbound balance and cannot route a single satoshi through the channel until payments begin flowing toward them. This is not a bug or an oversight in channel initialization. It is the direct arithmetic consequence of how the funding UTXO assigns its value: the party that contributed the funds owns the entire initial balance, and the counterparty's capacity to send is exactly the difference between the channel total and the funder's current balance. At channel open, that difference is zero.

The operational implications are immediate and non-negotiable. Inbound liquidity and outbound liquidity are not symmetric resources that both parties receive upon channel creation. The funder possesses full outbound capacity, meaning they can push payments toward the fundee up to the channel's total value. The fundee, conversely, possesses full inbound capacity but no outbound capacity whatsoever. For a buyer-agent opening a channel to a GPU provider, this alignment is structurally correct: the buyer needs to send, the provider needs to receive. But if that provider later needs to pay forward through the same channel, perhaps to settle a chained obligation or route a payment on behalf of a third party, the channel is useless for that purpose until enough payments have shifted balance to the provider's side. Channel topology and economic role must therefore inform funding strategy from the outset. An agent that opens channels indiscriminately, without modeling the expected direction and volume of payment flows, will find itself holding channels that are fully funded yet operationally inert for their intended use.

Dual-funded channel proposals address this asymmetry by allowing both parties to contribute inputs to the funding transaction, so that each side begins with nonzero outbound balance. If a provider and buyer each commit 250,000 satoshis to a 500,000-satoshi channel, both start with bidirectional capacity. The gain is real, but the coordination cost is not trivial. Both parties must have on-chain UTXOs available at channel-open time, they must negotiate who pays on-chain fees or how to split them, and the interactive signing protocol grows in complexity. For autonomous agents operating under tight micro-budgets, the requirement that both counterparties hold and commit on-chain funds at the same moment may conflict with liquidity constraints or timing requirements. Dual funding is a tool, not a default.

Sizing the initial channel requires a concrete estimate rather than a guess. The dominant flow direction determines the minimum useful capacity: if an agent expects to spend roughly 400,000 satoshis on inference calls over the channel's lifetime, the funding amount must exceed that figure by enough to cover routing fee reserves and any rebalancing overhead. A useful heuristic is to add around 10 to 15 percent margin above the expected cumulative spend. Then compare the on-chain transaction fee for creating the channel against the total micropayment throughput it will support. If the funding transaction costs 3,000 satoshis and the channel will process 400,000 satoshis of payments, the amortized cost per satoshi transacted is well under one percent. If the ex-

pected throughput is only 5,000 satoshis, the channel does not justify its on-chain footprint.

What matters most is the recognition that capacity planning and security design are not separate concerns layered on top of each other. They emerge from the same structure. The 2-of-2 multisig ensures neither party can unilaterally extract funds. The funder-pays-all initialization ensures only one party can initially send. Both properties trace back to the single UTXO that anchors the channel. An agent designer who treats the security model and the liquidity model as independent configuration surfaces will misdiagnose failures that arise from their intersection. A channel can be cryptographically sound yet commercially useless if the balance sits on the wrong side. Getting the initial allocation right is the first capacity-planning decision in any autonomous payment system, and it is the one that determines whether the channel can serve a single transaction before any off-chain state update occurs.

Off-Chain State Updates via Monotonically Increasing Sequence Numbers and Bilateral Signing

The moment a funding transaction confirms, the blockchain records a single fact: two parties locked a specific amount into a shared output requiring both signatures to spend. That snapshot is already stale. Every useful thing the channel will ever do happens afterward, off-chain, through a series of balance reallocations that the base layer never sees. But the chain remains the final arbiter of disputes, which means any prior balance state that was ever valid could, in principle, be broadcast and settled. A channel with ten thousand updates has ten thousand potential claims, and only one of them reflects the current truth.

This is not a bookkeeping problem. It is an adversarial ordering problem. Without a mechanism that forces newer states to strictly dominate older ones, a counterparty who held a more favorable balance three thousand updates ago has a direct financial incentive to publish that obsolete state and walk away with funds they no longer hold. The protocol must make this strategy not merely unlikely but economically ruinous. Monotonically increasing sequence numbers, coupled with bilateral revocation commitments, supply exactly this property. Each update carries a number that rises without exception, and each prior state is surrendered through a cryptographic act that converts any attempt to broadcast it into a penalty the honest party can claim. The sections ahead decompose how this ordering invariant is enforced, how bilateral signing achieves atomic consent without ambiguous intermediate states,

and why the resulting throughput operates at a fundamentally different timescale than on-chain settlement.

The Core Invariant: How Sequence Numbers Impose a Total Order on Competing Claims

Two autonomous agents hold a funded channel. Each has signed the opening commitment, locking collateral into a 2-of-2 multisig output with an agreed initial balance split. Over the channel's lifetime they will produce dozens, hundreds, perhaps thousands of successor states, each reassigning some fraction of that locked capacity from one side to the other. Every one of these states is a valid, fully signed transaction that could, in principle, be broadcast to the base layer. And that is precisely the problem. An on-chain contract presented with two conflicting claims, both carrying valid signatures from both parties, has no inherent basis for choosing between them. The signatures prove mutual consent but say nothing about temporal precedence. Without an ordering primitive embedded in the signed payload itself, the contract degenerates into an oracle problem, unable to distinguish the latest agreement from an obsolete one.

Sequence numbers resolve this by imposing the simplest mathematical structure sufficient for the task: a total order over a finite set of competing claims. Each signed state carries an integer, and each successive state must carry an integer strictly greater than its predecessor's. This monotonicity constraint is not bookkeeping convenience. It is a cryptographic invariant enforced by both parties at signing time. Because neither party will countersign a state whose sequence number fails to exceed the current maximum, no valid bilateral signature can exist for a state that both postdates another in real time and carries a lower sequence number. The ordering is tamper-evident by construction. Any attempt to forge a "newer" state with a stale number would require a signature that was never produced, and any attempt to replay a genuinely old state is immediately identifiable by its lower number.

The adjudication logic that results is almost trivially compact. Given two validly signed states submitted during a dispute, the on-chain contract compares their sequence numbers and accepts the higher one as canonical. Dispute resolution collapses from a subjective judgment into a deterministic integer comparison. No additional context is required: not timestamps, not external attestations, not a history of prior updates. The contract needs only the two states and their embedded integers. This reduction matters because every additional input to an on-chain

adjudicator expands the attack surface and the gas cost. A single comparison is both the cheapest and the most robust resolution mechanism available.

Totality is the deeper property at work. A partial order would leave some pairs of states incomparable, creating ambiguity branches that demand tiebreaking logic or, worse, a trusted third party to arbitrate. Sequence numbers eliminate this entirely. Every pair of states produced by a well-formed channel is comparable: one number is strictly greater, or the states carry the same number, in which case they are the same state. There is no undecidable case. For two autonomous agents negotiating GPU compute payments at machine speed, this property is not merely desirable but mandatory. An ambiguous ordering surface under adversarial conditions is an exploitable ordering surface.

One subtlety deserves emphasis. The gap between consecutive sequence numbers carries no semantic weight. A channel whose states are numbered 1, 2, 3 is identically ordered to one numbered 10, 1000, 1000000. What matters is strict inequality, not magnitude. Implementations may increment by one for simplicity, skip values for internal bookkeeping, or even use monotonic timestamps as sequence numbers, provided both parties attest to each new value via bilateral signature and the invariant holds: every new state's number exceeds every prior state's number. The freedom in gap size is a useful degree of implementation flexibility, but the monotonicity constraint itself is inviolable. Break it, and the total order dissolves. Without the total order, two competing claims become indistinguishable, and the channel's entire off-chain state history loses its enforceability.

What remains, then, is the question of how bilateral signing transforms each sequenced state into something stronger than a mere successor. The sequence number tells the contract which state is latest. But it does not yet explain how the act of co-signing a new state simultaneously renders all prior states not just outdated but economically dangerous to broadcast.

Bilateral Signing as Atomic Consent: Revocation of Prior States and the Penalty Commitment Structure

In early 2018, a developer on the Lightning Network's first mainnet implementation lost roughly 0.04 BTC by broadcasting a stale commitment transaction against a counterparty who had already received the revocation key for that state. The penalty was automatic, total, and correct. The developer later described the experience as the moment the

protocol stopped being an abstraction and became a credible economic threat. That anecdote distills the mental model at the heart of this discussion: bilateral signing is not confirmation. It is an atomic exchange of consent for vulnerability.

The model works as follows. When two channel counterparties cosign a new commitment transaction, each party is not simply acknowledging a new balance. Each is performing two simultaneous operations compressed into a single logical step. First, a party gains a new, valid, broadcastable claim on the channel's funds. Second, and this is the crux, a party surrenders the safe broadcastability of every prior claim by disclosing the revocation secret for the previous commitment. The disclosure is irreversible. Once your counterparty holds your revocation preimage for state n, broadcasting state n gives them the cryptographic material to sweep the entire channel balance through a penalty output. Signing state $n+1$ therefore means accepting that state n has become a loaded weapon pointed at you. The exchange is atomic in the precise sense that you cannot obtain the new state without arming the penalty against the old one. This is the first filament being spun in a broader settlement fabric: a single cryptographic strand that, on its own, handles direct pull between two parties but cannot yet distribute the lateral stresses of routing, discovery, or authorization.

The mechanism requires asymmetric commitment transactions. Both parties hold a version of the same logical state, but the two versions differ in structure. Your version contains a delayed output paying to yourself, encumbered by a timelock and a revocation path. Your counterparty's version mirrors this with their own delayed output. The asymmetry exists for attribution. If you broadcast a revoked state, your counterparty recognizes it as your version, applies the revocation key you previously disclosed, and claims the penalty before the timelock expires. Without this asymmetry, a breach would be indistinguishable from a legitimate close. The honest party would have no mechanism to identify the cheater or claim redress.

This architecture converts a coordination problem into a game-theoretic enforcement. The penalty must satisfy a specific incentive-compatibility condition: the cost of broadcasting a revoked state, which is forfeiture of the full channel balance, must exceed the maximum possible gain, which is the difference between the outdated balance and the current one. For any rational agent, old-state broadcast becomes a strictly dominated strategy. The channel does not need trust. It does not need a referee. It needs only the credible threat of disproportionate loss.

A critical subtlety reinforces the model's depth. Revocation is not deletion. Old commitment transactions remain valid, signed Bitcoin transactions. The blockchain will accept them. Nothing in the consensus layer prevents their broadcast. The system cannot make fraud technically impossible, so it makes fraud economically suicidal. This distinction recurs throughout every off-chain coordination design examined in later chapters. Wherever state must be enforced without continuous on-chain confirmation, the designer faces the same constraint: you cannot erase what has been signed, so you must make its use punishable.

The strategic consequence for autonomous machine commerce is direct. Any channel architecture for agent-to-agent micropayments that lacks a credible penalty structure collapses into one of two inferior alternatives. Either the agents require a trusted intermediary to adjudicate state disputes, reintroducing the latency and counterparty risk that channels were designed to eliminate, or they must settle every state update on-chain, destroying the cost and throughput properties that make sub-cent transactions viable. The penalty commitment is not an implementation detail. It is the minimum credible mechanism that permits two unsupervised processes to update shared economic state thousands of times without touching the base layer. Without it, the thread frays under the first adversarial pull.

Throughput Without Finality: Measuring Off-Chain Update Rates Against On-Chain Settlement Latency

A single ECDSA signature on a modern processor completes in roughly 50 to 100 microseconds. A Schnorr signature lands in a comparable range, sometimes faster due to batch-verification efficiencies that matter at scale. These numbers establish the floor: the cryptographic operation itself is not the bottleneck. What determines how many state updates a payment channel can sustain per second is the full round-trip required to propose, countersign, and acknowledge a new commitment transaction between two peers. On co-located nodes connected by a low-latency network link, that round-trip sits in the range of 1 to 5 milliseconds, yielding a theoretical ceiling of roughly 200 to 1,000 bilateral state updates per second per channel. Move the peers across continents, introduce typical internet latencies of 50 to 150 milliseconds, and the ceiling drops by one to two orders of magnitude. The comparison that matters here is not off-chain versus some abstract notion of speed but off-chain update rate against on-chain settlement cadence, because the

ratio between these two quantities determines whether micropayments are economically rational or merely technically possible.

On-chain settlement on Bitcoin averages around 10 minutes per block with a practical throughput of roughly 3 to 7 transactions per second across the entire network. Ethereum settles blocks every 12 seconds or so, with higher aggregate throughput but per-transaction finality that still requires multiple confirmations spanning minutes. Against these baselines, a payment channel that accumulates 10,000 off-chain state transitions before a single on-chain close transaction achieves an amortization ratio of 10,000 to 1. If the on-chain settlement cost is, say, $2 in fees, each off-chain update carries an effective settlement cost of $0.0002. That arithmetic is what converts a sub-cent payment from an accounting absurdity into a composable economic primitive. The amortization ratio is the single metric that determines channel viability for any given workload: an autonomous agent procuring inference tokens at fractions of a cent per call needs a ratio high enough that on-chain fees become negligible per interaction, and the update rate high enough that the agent is never blocked waiting for its own channel to cycle.

Yet throughput and finality are not the same measurement, and conflating them produces fragile system designs. Each bilaterally signed state update is operationally final between the two counterparties in the sense that either can enforce it on-chain via the dispute mechanism. This operational finality holds as long as both parties remain cooperative and the channel stays open. It does not constitute settlement finality, which requires an on-chain confirmation that no future dispute can reverse. Between these two poles lies a finality spectrum that machine-commerce architects must navigate explicitly. A sensor purchasing bandwidth every 100 milliseconds can tolerate operational finality for each micro-update because the value at risk per state is tiny. A procurement agent committing to a multi-dollar compute reservation may require settlement finality before releasing the workload. The design decision is not whether off-chain is "final enough" in the abstract but which finality threshold each transaction class demands, sized against the value exposed during the gap.

Cooperative throughput tells only half the story. When a counterparty delays its countersignature, whether from genuine latency, resource exhaustion, or strategic withholding, the effective update rate degrades to whatever timeout the channel protocol imposes before treating the peer as unresponsive. A channel designed for 500 updates per second

under cooperation may collapse to zero updates during a stall, with the only recourse being an on-chain force-close that consumes the very settlement budget the channel was designed to amortize. Robust channel sizing therefore requires distinguishing sustained cooperative throughput from worst-case adversarial throughput, and provisioning dispute windows and reserve balances accordingly. The engineering task is not to eliminate this gap but to make it measurable and to ensure that the amortization economics remain positive even when occasional adversarial episodes force premature settlement. When these parameters are calibrated correctly, the result is a settlement rail whose throughput scales independently of block production rate while retaining the cryptographic enforceability that makes unsupervised agent transactions tractable rather than aspirational. The question that follows, naturally, is what happens when enforceability must actually be exercised.

Dispute Resolution Mechanics: Challenge Periods, Stale State Penalization, and Cooperative vs. Unilateral Close

Bilateral signing keeps a channel honest only while both parties want it to stay open. The moment one side decides to settle, or vanishes, or broadcasts a commitment transaction that no longer reflects the latest balance, the protocol faces its actual design problem. Everything upstream of this point, the funding output, the sequence-indexed state updates, the exchanged revocation secrets, functions as scaffolding for a single guarantee: that on-chain settlement will converge to the most recent mutually signed state, even when one participant acts adversarially.

That guarantee is not cryptographic alone. It is temporal. A window of blocks must pass between when a commitment transaction hits the chain and when its outputs become spendable, and within that window the counterparty or its delegate must be alive, watching, and capable of responding. If the window closes unchallenged, a stale state becomes final. If a valid response arrives in time, the cheating party loses everything. This asymmetry, total forfeiture against a time-bound liveness requirement, is what converts an otherwise unenforceable off-chain promise into a credible settlement mechanism. And it produces a paradox worth sitting with: the penalty path that underwrites the entire security model almost never executes, because its mere existence reshapes the incentive landscape so thoroughly that rational participants default to a simpler, cheaper closure where both sides cooperate and split the final balances cleanly.

Understanding why the cooperative path dominates, and what breaks when it cannot, requires working through the penalty structure and its temporal assumptions with care.

Challenge Windows and Watchtower Liveness: The Time-Bound Security Assumption That Underpins Every Channel

A payment channel secured by a 2-of-2 multisig and updated through bilaterally signed state transitions possesses a curious property: its cryptographic integrity is complete, yet its economic safety is not. The multisig ensures that no funds move without both parties' consent. The monotonically increasing sequence numbers ensure that every state supersedes its predecessor. But neither mechanism, on its own, prevents a counterparty from broadcasting an outdated commitment transaction to the base chain and claiming a balance distribution that no longer reflects the true ledger. The gap between cryptographic validity and economic correctness is bridged by a single operational constraint: a finite window of time during which the honest party must observe the stale broadcast and respond.

This challenge window begins the moment a unilateral close transaction confirms on-chain. For the duration of the timeout, the settlement remains provisional. The counterparty who did not initiate the close can inspect the broadcast state, compare it against their own record of the latest sequence number, and if the broadcast is outdated, submit a revocation proof that triggers penalty enforcement. If no such proof appears before the window expires, the broadcast state finalizes and the funds disperse according to its terms. The entire penalty architecture depends on this interval. Remove it, shorten it past the point of reliable observation, or simply let the honest party go offline during it, and the revocation mechanism becomes inert. Security is not stored in the keys. It is rented through continuous vigilance over a countdown.

The duration of that countdown is a design parameter with direct economic consequences. A challenge window of, say, roughly one day gives the honest party a narrow margin to detect fraud but releases capital quickly. A window spanning a week or more provides a generous observation buffer at the cost of locking funds for the entire period if a dispute occurs. For an autonomous agent procuring GPU inference through a bilateral channel, as introduced in the discussion of on-chain funding transactions and 2-of-2 multisig collateral locking, this parameterization is not academic. Capital immobilized in a dispute window is capital unavailable for new channel openings or rebalancing. The trade-

off surface is explicit: every additional block of challenge time buys incremental safety against slow fraud detection and costs incremental liquidity. No single timeout value is universally correct. The choice depends on the participants' monitoring capacity, the value at stake, and the opportunity cost of locked collateral.

The liveness requirement embedded in this model distinguishes channel security from on-chain transaction finality in a fundamental way. An on-chain payment, once confirmed to sufficient depth, requires no further action from the recipient. A channel balance, by contrast, demands that someone remain ready to act for the full duration of every potential challenge window. For human users operating a single channel, this obligation is already burdensome. For autonomous agents managing dozens or hundreds of channels simultaneously, it becomes structurally untenable without delegation.

Watchtowers resolve this by converting a per-participant liveness obligation into a delegated monitoring service. The channel holder transmits an encrypted data blob to the watchtower each time a state update occurs. The blob contains the revocation key and pre-signed penalty transaction corresponding to the now-obsolete state, encrypted under a key derived from the old commitment transaction's identifier. The watchtower stores these blobs without the ability to decrypt them. It simply watches the chain. If it observes a transaction whose identifier matches a stored blob, it decrypts the payload and broadcasts the penalty transaction on the channel holder's behalf. This design preserves privacy: the watchtower learns nothing about channel balances or payment history during normal operation. It learns the contents of a single revoked state only at the moment that state is fraudulently broadcast, which is precisely the moment when action is required. The trust surface is narrow but real. The watchtower must be available, must not collude with the counterparty, and must have sufficient fee budget to get the penalty transaction confirmed before the challenge window closes. These are service-level guarantees, not cryptographic ones, and they carry their own incentive design requirements.

What emerges from this analysis is a security model with a distinctly operational character. The channel's funds are protected not by a static lock but by the intersection of cryptographic revocation data, a finite temporal window, and at least one entity with both the information and the connectivity to act within that window. This is the liveness-versus-safety tension at its most concrete: safety against stale-state theft is only as strong as the monitoring infrastructure's uptime during the challenge

period. For a single bilateral channel, this observation clarifies the trust assumptions. For a system of many channels serving autonomous agents at scale, it raises a question the single-channel primitive alone cannot answer.

Penalty Transactions and Full-Balance Forfeiture: Why Rational Actors Never Broadcast Stale Commitments

In early 2018, a Lightning node operator on testnet discovered something unsettling. After restoring a wallet from a stale backup, the node automatically broadcast an outdated commitment transaction. Within seconds, the counterparty's watchtower swept the entire channel balance. Not the disputed difference. Everything. That total loss was not a bug. It was the mechanism working precisely as designed. The penalty transaction is the primitive that makes off-chain state updates credible without trust. Broadcasting a revoked commitment does not merely revert state to the latest version. It hands the counterparty a cryptographic key that unlocks the cheater's full channel stake. Understanding exactly how this forfeiture operates, and why its severity is the *minimum* required for incentive compatibility, is the objective of this guide.

Step 1: Examine the Asymmetric Commitment Structure

Each channel state produces not one but two distinct commitment transactions, one held by each party. Alice's version pays her own output through a revocable script path, while paying Bob's output directly. Bob's version mirrors this asymmetry in reverse. This structural difference is not incidental. It is the foundation that makes penalty enforcement possible. Alice's own output in her commitment transaction is encumbered by a conditional spending path: she can claim it after a relative timelock expires, *or* Bob can claim it immediately if he possesses the revocation key for that state. Bob's commitment mirrors this logic with the roles swapped. Without this asymmetry, neither party could construct a transaction that punishes the other for broadcasting outdated state. Symmetric commitments would leave no lever for the counterparty to pull.

1. Identify that Alice holds commitment TX_A and Bob holds commitment TX_B for the same channel state N. These are not identical transactions.
2. Note that in TX_A, Alice's output is locked behind a two-branch script: a timelock path (Alice can spend after delay) and a revocation path (Bob can spend immediately with the revocation key).
3. Confirm that Bob's output in TX_A is a simple pay-to-pubkey output he can claim without restriction.
4. Recognize that TX_B mirrors this structure with roles reversed: Bob's output is encumbered, Alice's is not.

Step 2: Trace the Revocation Key Exchange During State Advancement

When Alice and Bob agree on a new state N+1, they do not simply overwrite state N. Instead, each party reveals the revocation secret for their own prior commitment transaction. Alice discloses the secret that allows Bob to spend Alice's encumbered output in TX_A at state N. Bob reciprocates for TX_B at state N. This exchange converts every prior commitment into a loaded instrument. After this exchange, state N is not merely superseded. It is *invalidated* in a precise cryptographic sense. If Alice now broadcasts her state-N commitment, Bob possesses the revocation key needed to sweep Alice's encumbered output immediately, bypassing the timelock. Combined with his own direct output, Bob claims the channel's entire balance. The revocation secret is the mechanism that transforms an old commitment from a valid fallback into a self-destruct device.

1. At state transition from N to N+1, Alice reveals revocation_secret_A(N) to Bob.
2. Bob reveals revocation_secret_B(N) to Alice.
3. Each party stores the counterparty's revocation secrets for all prior states, building a cumulative arsenal.
4. Any future broadcast of a revoked state now exposes the broadcaster to full-balance seizure.

Step 3: Quantify the Superlinear Penalty and Its Game-Theoretic Necessity

The penalty for broadcasting a revoked state is not proportional to the amount the cheater attempted to steal. It confiscates the cheater's *entire* channel balance. If Alice holds 0.3 BTC in a 1 BTC channel and broadcasts a stale state where she held 0.5 BTC, the penalty does not merely revert Alice to 0.3 BTC. Bob sweeps all 1 BTC. Alice loses everything, including the 0.3 BTC that was legitimately hers. This disproportionate severity is the minimum required for incentive compatibility. A proportional penalty, one that only clawed back the stolen delta, would leave cheating as a zero-cost lottery: if the counterparty misses the challenge window, the cheater profits; if caught, the cheater merely returns to the honest state. Under proportional penalties, attempting fraud is weakly dominant. Full-balance forfeiture ensures that the expected cost of cheating exceeds the expected gain under any realistic monitoring assumption, converting fraud from a free option into a dominated strategy.

1. Calculate the cheater's best-case gain: the delta between the stale balance and the current balance.
2. Calculate the cheater's worst-case loss: the cheater's entire current channel stake.
3. Observe that worst-case loss exceeds best-case gain whenever the cheater has any positive balance, making cheating negative expected value if the counterparty monitors with non-trivial probability.
4. Confirm that proportional penalties would make the expected cost of cheating zero when the counterparty fails to respond, creating a free-option problem that full forfeiture eliminates.

Step 4: Evaluate the Rational Actor's Decision Under Full Forfeiture

Frame the decision to broadcast a stale commitment as a formal cost-benefit calculation. The cheater gains the delta only if the counterparty fails to respond within the challenge window. The cheater loses their entire channel stake if the counterparty does respond. For cheating to have positive expected value, the probability of the counterparty being offline for the entire challenge window must exceed a threshold determined by the ratio of gain to loss. Under production conditions, where counterparties run persistent nodes or delegate monitoring to watchtowers, the probability of sustained non-response is negligibly small. The rational actor confronts a dominated strategy: no plausible estimate of counterparty downtime makes broadcasting a revoked state net-positive. The stale commitment is not merely risky. It is, for any agent with positive channel balance and a minimally attentive counterparty, a financial self-destruct instruction.

1. Define p as the probability the counterparty detects the revoked broadcast within the challenge window.
2. Expected value of cheating = (1-p) × delta_gain − p × total_stake_loss.
3. For cheating to be rational, (1-p) × delta must exceed p × stake, requiring p < delta / (delta + stake).
4. In practice, even modest monitoring infrastructure pushes p above 0.99, making the inequality impossible to satisfy for any realistic balance distribution.

Step 5: Identify the Boundary Conditions Where Deterrence Degrades

The penalty model's deterrence guarantee is not absolute. It weakens at identifiable boundary conditions that protocol designers must account for. The first is the near-zero-balance edge case: if Alice's current legitimate balance is close to zero, she has almost nothing to lose from a penalty sweep. Broadcasting a stale state where she held a larger balance becomes a cheap gamble. The forfeiture of a negligible stake is not a meaningful deterrent. The second boundary condition involves counterparty liveness. If Alice has strong reason to believe Bob is permanently offline, or that Bob's monitoring infrastructure has failed, the probability of detection drops toward zero and the expected value of cheating turns positive regardless of penalty severity. This is the liveness assumption that underpins the entire penalty model. Deterrence requires that someone, whether the counterparty or a delegated watchtower, is monitoring the chain. When that assumption fails, the penalty mechanism's game-theoretic guarantee dissolves.

1. Identify the near-zero-balance case: when a party's current stake approaches zero, the ratio of potential gain to potential loss inverts, and cheating can become positive expected value.
2. Identify the offline-counterparty case: when the probability of detection within the challenge window approaches zero, no penalty severity is sufficient to deter fraud.
3. Connect both conditions to the design requirements they impose: the first motivates minimum balance thresholds or channel rebalancing strategies; the second motivates watchtower delegation and liveness monitoring.

You have now traced the full mechanism by which penalty transactions convert revoked commitments into economic self-destruct devices. The asymmetric commitment structure creates the vulnerability. The revocation key exchange arms the counterparty. The full-balance forfeiture sets the penalty magnitude high enough to make cheating a dominated strategy. And the boundary conditions at near-zero balances and offline counterparties define the precise limits of this guarantee. With this understanding, off-chain state updates are no longer informal promises. They are credible commitments backed by cryptographic enforcement and calibrated economic threat. The next question becomes operational: what happens when the liveness assumption itself requires infra-

structure support, and how do watchtower architectures extend the penalty model's deterrence envelope to agents that cannot monitor the chain continuously?

Cooperative Close as the Common Case: Fee Savings, Immediate Finality, and the Decision to Force-Close

A node operator initiating a channel closure begins by signaling the intent to settle. When both parties are online and agree on the latest balance distribution, they co-sign a single closing transaction that spends the funding output directly to each participant's chosen address. No challenge window opens. No reserve remains locked. The funds become spendable as soon as the transaction confirms, typically within the next block. This is the cooperative close path, and it is not merely the polite alternative to dispute resolution. It is the engineered equilibrium that the entire penalty apparatus is designed to produce.

The economic case is straightforward once you decompose the on-chain cost of each closure path. A cooperative close requires exactly one transaction: a simple spend from the 2-of-2 multisig funding output to two pay-to-witness outputs. A force-close, by contrast, begins with a unilateral commitment transaction broadcast and then requires at least one additional sweep transaction after the CSV delay expires, plus separate HTLC-timeout or HTLC-success transactions for any in-flight conditional payments. The cooperative path saves roughly half the total on-chain fee burden in the simplest case, and the savings widen as pending HTLCs accumulate. Beyond raw transaction fees, the force-close path imposes opportunity cost: capital remains locked throughout the challenge period, unavailable for reallocation to other channels or settlement obligations. For an autonomous agent managing a portfolio of channels across multiple service providers, this capital inefficiency compounds. Every cooperative close that avoids a CSV lockup frees liquidity for redeployment within seconds rather than hours or days.

These savings do not arise from goodwill. They arise from the cost asymmetry that the penalty mechanism enforces. Broadcasting a revoked commitment transaction risks full-balance forfeiture. Broadcasting the current commitment transaction unilaterally still triggers the challenge delay and its associated sweep costs. Cooperative close avoids both penalties and delays. Rational participants, computing the expected cost of each path, converge on mutual settlement not because they trust each other but because the alternative is strictly more expensive when both

parties are responsive and the channel state is uncontested. The dispute machinery is most successful precisely when it never fires.

The decision to force-close becomes rational under a narrow set of conditions, each with measurable thresholds. The first is counterparty unresponsiveness: if a peer fails to respond to a cooperative close request within a configurable liveness timeout, often set between thirty seconds and several minutes depending on the implementation, the initiating node has no mechanism to co-sign a mutual close and must broadcast the latest commitment unilaterally. The second condition involves pending HTLCs whose value approaches the dust threshold under prevailing fee rates. If the cost to claim an HTLC on-chain would exceed the HTLC's value, that conditional payment becomes uneconomical to resolve, and the channel operator must force-close before additional HTLCs accumulate in this unrecoverable range. The third trigger is adversarial: detection of a revoked commitment transaction broadcast by the counterparty. Here, force-close is not a choice but a mandatory defensive action, since the penalty claim must be executed before the challenge period expires. Each of these conditions can be encoded as an automated policy within node software, transforming the force-close decision from a manual judgment into a programmable threshold response.

Protocol implementations that expose these thresholds as configurable parameters allow operators to tune their closure strategy to match their capital constraints and risk tolerance. An agent managing high-frequency, low-value channels for API procurement might set aggressive liveness timeouts, preferring a fast force-close over waiting for an unresponsive peer, because the capital locked in the challenge window exceeds the channel's remaining utility. An agent managing fewer, larger channels with trusted long-term counterparties might extend the timeout substantially, accepting brief unresponsiveness as transient noise. The framework reduces to a cost comparison at each decision point: the expected fee and opportunity cost of force-close versus the expected cost of waiting for cooperative settlement. When this comparison is automated and the penalty structure makes cheating irrational, cooperative close dominates overwhelmingly. The channel's closure economics then become predictable inputs to a broader lifecycle cost model, one that will prove essential when evaluating how state channel architectures amortize settlement overhead across thousands of sub-millisecond state transitions.

Conclusion

Three mechanisms, examined separately across this chapter, lock together into something greater than any one of them achieves alone. The 2-of-2 multisig funding transaction converts counterparty risk into cryptographic lockup, creating a trust anchor that requires no reputation, no legal identity, and no prior relationship. Monotonically increasing sequence numbers then decouple transaction throughput from on-chain finality cost, allowing the same locked collateral to shuttle back and forth across an unbounded number of state updates without ever touching the base layer. And the penalty-based dispute mechanism completes the architecture by making cooperative behavior the dominant economic strategy: broadcasting stale state costs the cheater their entire channel balance, so rational participants never do it. What emerges is not a scaling optimization but a self-enforcing bilateral contract. The blockchain serves as the court of last resort, a venue that well-designed channels almost never visit. The vast majority of economic activity settles in signed messages exchanged between two parties, enforced not by judicial process but by the credible, automated threat of cryptographic penalization.

One misconception will surface repeatedly as you move deeper into this material, and it is worth confronting now: channel capacity does not limit transaction volume. The locked collateral constrains the maximum size of any single transfer, not the number or cumulative value of exchanges over the channel's lifetime.

Chapter Two

State Channel Architecture for Sub-Millisecond Micropayment Settlement

Most discussions of payment channels treat speed as a simple binary: slow on-chain, fast off-chain. That framing obscures the actual engineering problem. An autonomous agent purchasing a 200-millisecond burst of GPU inference cannot wait six minutes for a Bitcoin block or twelve seconds for an Ethereum slot. Yet the security of any payment depends, ultimately, on settlement finality that only an on-chain consensus mechanism can provide. The speed is real, but it is not free, and it is not unconditional.

State channels resolve this by splitting the problem rather than eliminating either side of it. They move the speed-critical path off-chain, where bilateral signatures update balances in microseconds, while preserving the ability to enforce correct outcomes on-chain if a counterparty defects. The slow layer does not disappear. It becomes unnecessary in the common case and available as a backstop in the adversarial one. This chapter establishes the complete architectural framework for that split: how liveness assumptions trade against finality guarantees, how

channel factories amortize the prohibitive on-chain cost of opening individual channels across many agent pairs, and how the Bolt protocol's messaging layer coordinates peer-to-peer channel state without centralized intermediaries. The goal is precise reasoning about where latency hides, what it costs to remove, and which construction choices determine whether a channel architecture survives production load.

But the split between off-chain speed and on-chain finality introduces a design surface with real consequences. Liveness assumptions, watchtower availability, and acceptable reorg risk determine how much latency you actually eliminate. That design surface is where settlement architecture begins.

Settlement Latency Trade-Offs: Liveness Guarantees vs. On-Chain Finality Requirements

Off-chain state advances already settle faster than any block interval can confirm. That claim is uncontroversial. But the assumption that follows from it, that displacing settlement off-chain is a pure latency win, deserves sharper scrutiny. Every millisecond gained by moving state updates between counterparties without touching the base layer introduces a corresponding obligation: someone, or something, must remain online to detect and respond to dishonest channel closures within a bounded window. The speed is real. So is the new failure surface.

For human-facing payment applications, this trade-off is manageable because transaction frequency is low and the value at risk in any single update rarely justifies sophisticated attacks on liveness. Machine-to-machine streams operate under different constraints entirely. An autonomous agent executing hundreds of micropayments per second against a compute provider cannot pause to wait for on-chain confirmation, but it also cannot ignore the possibility that its counterparty will broadcast a stale state and walk away with funds. The design space narrows to a precise question: how do you calibrate timeout windows, watchtower coverage, and finality thresholds so that sub-millisecond settlement remains operationally safe rather than just operationally fast? Getting this calibration wrong does not merely degrade performance. It converts a latency advantage into a direct fund-loss vulnerability, and no amount of channel throughput compensates for that.

Why Off-Chain State Advances Decouple Payment Speed from Block Confirmation Intervals

The bottleneck in on-chain payment is not computation. It is consensus. A transaction can be constructed, signed, and propagated across a peer-to-peer network in milliseconds, but it cannot settle until validators agree on its ordering relative to every other transaction in the system. That agreement process imposes a floor on settlement latency equal to the block interval, regardless of how simple or urgent the payment is. A 0.001-cent transfer for a single inference request waits in the same queue, subject to the same confirmation rhythm, as a multimillion-dollar contract execution. The block interval is not a processing delay. It is a coordination delay, and no amount of local optimization eliminates it.

Off-chain state advances remove this coordination requirement from the critical path. When two counterparties hold an open channel, they advance their shared state by exchanging signed messages directly. Each message contains a new balance allocation and a sequence number strictly greater than the previous one. The counterparty validates the signature, confirms the sequence number is monotonically increasing, countersigns, and returns the acknowledgment. The entire round trip consists of one network hop plus two signature operations. On modern hardware, Ed25519 or Schnorr signature generation and verification each complete in tens of microseconds. Even accounting for network latency between colocated machines, the full update cycle can finish well under a millisecond. Nothing in this process references a block, waits for a miner, or requires any third party to observe the transition.

The safety of this arrangement rests on a single invariant: every signed state update is a valid on-chain transaction that the channel's smart contract will accept, and the contract always enforces the update with the highest sequence number. Intermediate states are never submitted during normal operation, but any party can submit the latest state unilaterally if the counterparty becomes unresponsive. The blockchain, in this architecture, functions as a court of last resort rather than a transaction processor. It adjudicates only when cooperation fails. This means the chain's confirmation interval governs dispute resolution timelines, not payment throughput.

The result is a dual-clock system. The fast clock is the off-chain state advance rate, bounded by message round-trip time and signing throughput. The slow clock is the on-chain finality interval, bounded by block production and confirmation depth. Normal channel operation runs

exclusively on the fast clock. The slow clock activates only under adversarial conditions or at channel close. These two clocks are independent. Improving one does not require improving the other, and degradation of the slow clock does not impair the fast clock during cooperative operation. This separation is the architectural core of channel-based settlement.

For autonomous agents procuring compute resources in real time, this clock separation is not a performance optimization. It is a structural prerequisite. An inference agent issuing requests at hundreds or thousands per second cannot gate each request on a ten-second block interval, let alone a ten-minute one. Per-request billing becomes physically impossible if every payment must pass through global consensus. Off-chain state advances make per-millisecond or per-request settlement viable because the payment latency matches the service latency. The agent signs a state update, the provider verifies it, and the compute begins. The entire exchange completes within the same time horizon as the network call that delivers the request itself.

But speed alone does not complete the picture. The fast clock runs only while both parties remain online and responsive. The moment one side disappears, the system must fall back to the slow clock to resolve the final state. How that fallback behaves, what assumptions it requires about monitoring infrastructure, and what windows of vulnerability it opens for autonomous processes that may run as ephemeral containers rather than persistent services: these are the liveness constraints that shape the operational envelope around the latency gains just described.

Liveness Assumptions Under Adversarial Conditions: Watchtower Dependence, Timeout Calibration, and Forced Closure Deadlines

Roughly seven in ten Lightning Network force-closure events observed during high-fee periods in 2023 involved commitment transactions that sat unconfirmed for multiple blocks, according to estimates from independent node operators tracking mempool dynamics. That statistic captures something essential about the liveness obligations created by off-chain settlement. Every state update that bypasses on-chain confirmation does not eliminate the need for chain access. It defers that need to a future moment when the channel may be under adversarial pressure, the mempool congested, and the cost of timely inclusion sharply elevated. The speed gain is real, but it is not free. It purchases a standing obligation: someone, at some point, must be watching.

The watchtower architecture makes this obligation explicit by converting a continuous personal liveness requirement into a delegated monitoring contract. A watchtower does not validate new channel states or participate in routing. Its function is narrower and more precisely defined. It stores breach remedy data, pre-signed penalty transactions keyed to the commitment transaction identifiers of revoked states, and it monitors the blockchain for the broadcast of any such revoked commitment. When a counterparty attempts to close a channel using stale state, the watchtower detects the offending transaction and submits the corresponding justice transaction to claim the penalty output. The trust model shifts from "I must be online within the dispute window" to "at least one of my delegates must be online, hold the correct remedy data, and be willing to pay the fee required for timely inclusion." That final clause matters. Watchtower operation is not costless. The delegate must maintain data availability for every revoked state across every monitored channel, must not collude with the counterparty, and must find it economically rational to submit the penalty transaction even when on-chain fees spike. If the penalty output is smaller than the fee required for next-block inclusion during a congestion event, a rational watchtower may decline to act. This is not a theoretical edge case. It is a budgetable failure surface.

Timeout calibration compounds the problem. HTLC expiry deltas and dispute windows must be sized against block-production variance under normal conditions and adversarial delay under attack. A CLTV expiry delta of, say, forty blocks provides roughly six to seven hours of response margin on Bitcoin under typical block intervals. But an adversary who controls sufficient hashrate or who floods the mempool with high-fee transactions can compress that margin considerably. The timeout must account not only for average confirmation time but for the tail distribution of confirmation delays when an attacker is actively working to prevent the counterparty's transaction from being included. Shorter timeouts reduce the capital locked in pending HTLCs and improve channel throughput for time-sensitive machine commerce. Longer timeouts widen the safety margin but increase opportunity cost, binding liquidity that could otherwise be deployed across parallel channels or routing paths. This is not a binary choice but a tunable parameter with quantifiable consequences on both axes. The engineering task is to select a value that prices the capital lockup against the probability-weighted cost of a missed dispute window.

When these parameters are miscalibrated, the failure modes are specific and enumerable. A stale commitment transaction broadcast by a malicious counterparty, undetected within the dispute window because the honest party was offline and no watchtower held the relevant remedy data, results in the revoked state being accepted as final. The counterparty extracts funds that should have been penalized. An HTLC whose expiry elapses before the preimage can be relayed upstream, whether due to routing delays, fee-induced confirmation lag, or deliberate griefing, leaves the payment either stuck or reversed against the forwarding node. A watchtower that holds correct data but cannot get its justice transaction confirmed before the timelock expires produces the same outcome as having no watchtower at all. Each failure traces back to a design parameter: the dispute window length, the HTLC expiry delta, the fee budget allocated for penalty submission, or the number and reliability of watchtower delegates provisioned per channel.

The cost curve is continuous. Every millisecond of settlement latency reduced by moving state transitions off-chain adds a fractional liveness obligation that must be staffed, delegated, or hedged. For autonomous machine-commerce deployments operating at high frequency across many channels, these obligations aggregate into infrastructure line items: watchtower service fees, capital reserves for dispute-window coverage, fee-bumping budgets sized to adversarial mempool conditions. Treating liveness as a priced resource rather than a background assumption is what separates a deployable channel architecture from a demonstration prototype. The settlement speed is real, but its budget must be explicit.

Choosing Settlement Granularity for Machine-to-Machine Streams: When Probabilistic Finality Is Sufficient and When It Is Not

The moment an off-chain balance accumulates past a certain threshold, the question is no longer whether the counterparty can be trusted. It becomes whether the cost of betrayal has fallen below the reward. That shift reframes settlement granularity as something other than a protocol constant. It is a per-stream design variable, recalculated continuously, governed by the intersection of reorg probability, accumulated exposure, and the price of anchoring state back to the chain.

Consider the core relationship. For any block depth d on the underlying chain, there exists a probability P(reorg, d) that the chain reorganizes past that depth. Multiply this by the unsettled value V accumulated

in the off-chain stream, and you obtain an expected loss: $E = P(\text{reorg}, d) \times V$. Compare E against the marginal cost of an on-chain checkpoint, which includes gas expenditure, the opportunity cost of capital locked during confirmation, and any routing disruption the checkpoint introduces. When E remains below that marginal cost, probabilistic finality is operationally sufficient. The stream continues accumulating off-chain. When E crosses the threshold, the stream must either reduce its unsettled window or trigger settlement. This is not a heuristic. It is a continuously evaluable inequality that every machine-to-machine payment controller can compute in real time.

What distinguishes machine streams from human payment flows is the speed at which that inequality changes. An autonomous agent procuring GPU cycles at ten thousand micropayments per minute starts each interval with negligible unsettled exposure. Within seconds, cumulative value can grow large enough that the expected-loss threshold flips. The granularity decision therefore recurs dynamically across the life of the stream, not just at initiation. A well-designed payment controller treats the checkpoint interval as an adaptive parameter, shortening it as the unsettled balance grows and lengthening it when cumulative exposure drains through periodic settlement or counterparty acknowledgment.

A second constraint operates independently of the stream's own risk tolerance. A rational adversary will attempt to exploit a reorg window only when the unsettled balance exceeds their cost of mounting the attack. On proof-of-work chains, that cost approximates the expense of selfish mining or bribing miners for a sufficient number of blocks. On proof-of-stake chains, it relates to the slashing penalty and the capital required to control enough validator weight. This creates a game-theoretic ceiling on safe unsettled accumulation that is a property of the base chain's security budget, not of the application's preferences. Even a risk-neutral stream operator who would otherwise tolerate high expected loss must respect this ceiling, because beyond it the counterparty's dominant strategy shifts from honest completion to reorg exploitation.

These two constraints, the expected-loss threshold and the adversarial incentive ceiling, interact with a third factor that shapes the full design surface. Choosing coarser granularity reduces on-chain cost but simultaneously widens the window during which a malicious counterparty can time a forced closure to their advantage. Longer intervals between checkpoints increase dependence on watchtower liveness to detect and respond to stale state submissions. The design space is therefore a three-

way trade-off surface spanning checkpoint cost, reorg exposure, and liveness requirements. Flattening it into a simple latency-versus-finality slider loses the adversarial dimension entirely.

In practice, a machine payment controller implementing this framework maintains three inputs updated each evaluation cycle: the current unsettled balance, the estimated reorg probability at the most recent confirmation depth, and the prevailing on-chain settlement cost. It compares the expected loss against the checkpoint cost, checks the unsettled balance against the adversarial ceiling derived from the chain's security budget, and selects the tighter of the two resulting constraints as its effective settlement trigger. When neither threshold is breached, the stream advances off-chain without interruption. When either is crossed, the controller initiates a checkpoint or reduces the accumulation rate. This transforms settlement granularity from a static configuration parameter into a live control variable, responsive to the economic and adversarial conditions of each moment. The watchtower infrastructure that monitors for stale submissions and the timeout calibration that governs dispute windows both depend on knowing how wide these unsettled intervals can grow, a dependency the next analysis takes up directly.

Channel Factory Constructions and Amortized On-Chain Setup Costs

Funding each bilateral channel with its own on-chain transaction is often treated as an acceptable cost, a one-time expense amortized over the channel's lifetime. But that assumption breaks down the moment an autonomous agent network needs to maintain not two or five concurrent payment relationships, but fifty or several hundred. At that density, the aggregate funding cost stops being a rounding error and becomes the dominant economic barrier to deployment. The per-channel model that Section 2.1 described works cleanly for isolated pairs, yet it scales linearly in on-chain fee expenditure with every new counterparty relationship added.

The structural response is to share a single funding transaction across many independent channels, organizing committed capital into a tree of UTXOs that branches outward from one on-chain anchor into dozens of bilateral relationships. This is fundamentally an amortization problem: one confirmation window, one fee payment, one block-space footprint, divided across every channel the tree supports. The savings are real but conditional. Factory size, expected channel lifetime, the fre-

quency of on-chain rebalancing events, and the critical requirement that any single channel remain closeable without cooperation from unrelated participants in the same structure all interact to determine whether the construction actually delivers its promised cost reduction or collapses back toward per-channel economics under adversarial conditions. Getting this arithmetic right is the difference between a compute marketplace that can economically serve autonomous agent pairs at scale and one that prices itself out of viability before its first settlement.

Shared UTXO Trees: How a Single Funding Transaction Spawns Multiple Independent Bilateral Channels

One funding transaction per channel is not a law of nature. It is a design default, and a costly one. The assumption collapses the moment you recognize that a single on-chain UTXO can encode an internal allocation structure far richer than a simple two-party split.

Consider a funding transaction whose output is locked by an n-of-n multisig among all parties who wish to transact within a shared factory. This output commits, on-chain, exactly one UTXO. But the parties simultaneously co-sign a tree of off-chain transactions that progressively subdivide the committed funds. At the root sits an allocation transaction that splits the factory's total balance into intermediate branches. Each branch is itself a multisig output whose signers are the subset of parties whose funds reside below that node. The branching continues until the leaves resolve into ordinary 2-of-2 multisig outputs, each indistinguishable in function from a standalone bilateral payment channel. The blockchain registers one transaction. The off-chain layer operates N independent bilateral relationships, each with its own sequence of state updates, its own dispute path, and its own cooperative close flow. The structural isomorphism here is precise: every trust boundary in the off-chain topology maps onto a branch point in the transaction tree.

The nesting of pre-signed transactions at each level is what makes this construction viable rather than merely elegant. A root allocation transaction distributes funds to intermediate nodes. Each intermediate node's output can be spent only by the parties whose balances lie in the subtree below it. At the leaf level, each 2-of-2 output functions as a channel whose updates require only the two counterparties' signatures. No other participant in the factory needs to know or consent when a leaf channel updates its internal state. The signature requirements narrow as you descend the tree, and this narrowing is what delivers independence.

Isolation under dispute follows directly from the tree's branching structure. If a single leaf channel enters a contested state, the disputing party publishes only the transaction path from that leaf back to the root. The intermediate and leaf outputs along this path are placed on-chain. But sibling branches remain untouched. Their funds stay committed under their own multisig conditions, and their bilateral channels continue operating off-chain as if nothing happened. One party's misbehavior does not cascade into a factory-wide unwind. This containment property is critical for autonomous agent deployments, where a single unreachable process should not force dozens of unrelated channels to close.

The trade-off surfaces at non-leaf levels. Updating the allocation at any intermediate node, for instance rebalancing funds between two branches, requires the unanimous consent of every party sharing that subtree. The larger the subtree, the more signers must be online and cooperative. Factory design is therefore a problem of sizing subtrees to match realistic coordination assumptions. A factory of fifty agent pairs operated by a single fleet controller can tolerate a broad root because the controller is a common signer. A factory spanning multiple independent operators would need narrower branches to keep coordination sets small. The tree's shape encodes trust topology as directly as it encodes fund allocation.

This mapping reveals why the shared UTXO tree is not merely an optimization over per-channel funding. It is a structural primitive that mirrors the principal-agent delegation hierarchy native to autonomous compute procurement. An operator's treasury anchors the root. Intermediate branches partition funds across agent clusters or service categories. Leaves instantiate the actual bilateral payment relationships between individual agents and their providers. The hierarchy of signing authority matches the hierarchy of spending authority, and both are enforced by the same cryptographic structure. What remains is to quantify the cost advantage this yields as factory size grows and to trace the concrete message flows that bring such a factory to life.

Amortization Arithmetic: Per-Channel On-Chain Cost as a Function of Factory Size, Lifetime, and Rebalancing Frequency

During a late-night deployment review in early 2024, an infrastructure lead at a mid-sized compute brokerage ran the numbers on spinning up forty bilateral payment channels for a new fleet of procurement agents. The on-chain fees alone, at roughly 50 sat/vbyte that week, consumed

more budget than the first month of projected inference purchases. She shelved the rollout until someone on her team modeled the alternative: a single factory funding transaction covering all forty channels. The per-channel cost dropped by over an order of magnitude. But the model was incomplete. It assumed every channel would survive its full intended lifetime and that no rebalancing would ever touch the chain. Those assumptions, left unexamined, would have made the factory look better than it actually was.

The core arithmetic begins with a straightforward ratio. A direct channel open costs some base fee F, determined by the funding transaction's virtual byte weight and the prevailing fee rate. A factory that seats N participant pairs replaces N individual funding transactions with one shared transaction whose weight grows sublinearly with N, thanks to aggregated key structures and shared outputs. The per-channel setup cost falls to roughly F / N plus a coordination overhead term that scales with the number of interactive signing rounds and the computational cost of constructing the shared UTXO tree. This overhead is not negligible. Multisig complexity, particularly when moving beyond simple Schnorr aggregation to accommodate fault-tolerant signing protocols, introduces latency and bandwidth costs that grow faster than linearly once N exceeds around 30 to 50 participants. The cost curve is concave: savings per additional participant diminish well before triple-digit factory sizes, and the coordination burden begins to dominate. For most operational scenarios, the marginal benefit of adding a 60th participant to a factory is a rounding error on the fee savings but a measurable increase in setup fragility.

Channel lifetime acts as a multiplier on those savings. A factory whose channels persist for 30 days distributes their amortized setup cost across every micropayment settled in that window, potentially thousands of sub-millisecond state updates per channel. But lifetime is not a fixed parameter. It must be discounted by the probability that at least one participant forces a unilateral close, which triggers partial on-chain settlement of the UTXO tree. Each premature exit claws back a fraction of the amortization gain proportional to how early it occurs and how many sibling channels it forces to close or restructure. Expected effective lifetime is therefore the nominal lifetime multiplied by a survival probability that depends on participant reliability, dispute likelihood, and the penalty cost structure embedded in the factory's exit protocol.

Rebalancing frequency is the variable most operators underestimate. Every intra-factory rebalance that requires an on-chain splice or a full

factory-level state update erodes the funding transaction savings that justified the factory in the first place. The critical metric is the ratio of rebalances to channel lifetime. If a channel expected to live 14 days requires two on-chain-touching rebalances, and each rebalance costs even a quarter of a direct channel open, the factory's amortization advantage compresses toward the cost of simply opening channels individually. In a sustained high-fee environment of around 200 sat/vbyte, the absolute savings from factory amortization grow larger, but so does the penalty for each rebalance event. Operators must evaluate both the savings curve and the risk-adjusted rebalance penalty simultaneously. The break-even threshold shifts depending on the fee regime, but the structural lesson holds: rebalancing is the silent cost that can collapse an apparently favorable factory deployment into economic indifference.

These three variables converge on a practical sizing heuristic. For an agent compute marketplace expecting around 50 active agent pairs with median channel lifetimes between 7 and 14 days and an estimated one to two rebalances per channel lifetime, the arithmetic points toward factory sizes in the range of 10 to 25 participants as the operational sweet spot. Below 10, the per-channel savings are too modest to justify the coordination protocol. Above 25, the marginal fee reduction flattens while the probability of a disruptive unilateral exit within the factory lifetime rises steeply. Splitting 50 pairs across two or three factories of 15 to 20 participants each captures the bulk of available amortization while bounding the blast radius of any single non-cooperative exit. The numbers are not decorative. They determine whether sub-millisecond settlement remains economically viable or whether on-chain gravity pulls the entire architecture back toward per-transaction cost structures that micropayment commerce cannot sustain.

A Compute-Marketplace Factory Serving Fifty Agent Pairs from One On-Chain Commitment

Tracing the failed penalty broadcast back through six mempool snapshots, Maren finds what she suspected: the timeout expired not because the chain was slow, but because forty-nine other signers were. The test lab's racks hum with simulated Lightning nodes, each running adversarial close scenarios against a factory prototype she has been building for three weeks. The scenario under stress is specific. A single on-chain funding transaction locks roughly 2.5 BTC across a shared UTXO tree, spawning fifty bilateral channels between twenty-five buyer agents and twenty-five compute providers. Each channel holds an allocation of

around 0.05 BTC, enough to cover approximately forty GPU-hours at current spot pricing. The funding transaction itself is a single input committing to a Taproot output whose spending conditions encode the fifty-party cooperative close path and, in the leaves, each bilateral channel's force-close script. On-chain, this structure occupies one transaction of roughly 800 virtual bytes, amortizing the per-channel funding cost to an estimated 16 vbytes per pair. Compared to independent channel opens at around 170 vbytes each, the factory saves over ninety percent of on-chain weight for the opening phase alone.

The lifecycle she is simulating mirrors what a production compute marketplace would demand. A factory coordinator, operated by the marketplace itself, batches channel-open requests during a designated funding epoch of approximately ten seconds. During that window, agents that have discovered providers through a registry submit signed channel-open intents. The coordinator assembles these into a single multi-party funding proposal. Every participant must countersign the allocation tree before the funding transaction broadcasts. In Maren's test, the signing round completes in around 1.2 seconds when all fifty parties respond promptly. Once confirmed on-chain, each bilateral channel activates independently. Buyer agents begin streaming micropayments against GPU-hour slots, updating state at sub-second intervals without touching the base chain. The factory's shared UTXO tree is inert during normal operation. It exists only as a fallback enforcement structure, a property Maren finds elegant in principle and fragile in practice.

The fragility surfaces when she kills provider node seventeen. In a simple two-party channel, one node going offline triggers a unilateral close affecting only that pair. Inside the factory, the same event forces a partial close: the bilateral channel between buyer twelve and provider seventeen must be settled on-chain by publishing its branch of the allocation tree, costing roughly 250 vbytes for the branch proof and settlement output. Crucially, the remaining forty-nine channels continue operating. The factory does not collapse. But the cost ledger she maintains on the whiteboard beside the racks tells a subtler story. Each partial close erodes the amortization advantage. If five channels close independently over the factory's expected two-week lifetime, the effective per-channel on-chain cost rises from the original 16 vbytes to around 40 vbytes. Still cheaper than independent opens, but the margin compresses with every early exit.

The real constraint, though, is the signing round. Maren replays her logs from a factory construction attempt where three of fifty participants were intermittently reachable. The coordinator retried the signing round four times, pushing factory activation from the target ten-second epoch to over ninety seconds. At fifty parties, a single unresponsive signer blocks everyone. Her analysis yields a threshold she pins to the whiteboard: when expected participant churn exceeds roughly fifteen percent per factory lifetime, or when mean channel duration drops below three days, the coordination overhead of repeated signing rounds and partial closes consumes the on-chain savings. Below that boundary, independent two-party channel opens are cheaper in total lifecycle cost despite their higher per-unit on-chain footprint.

She stares at the failed penalty transaction that started the evening's investigation. The penalty itself was valid. The dispute script was correct. The problem was that broadcasting the penalty required reconstructing the allocation branch proof, and the data for that proof depended on a cosigner who had already gone offline. The liveness assumption she understood from bilateral channels, where a watchtower monitors one counterparty, scales differently inside a factory where branch proof construction can depend on cooperative data availability from parties outside the dispute. She sketches a new state diagram on the glass partition behind her monitor, mapping the dependency graph between factory-level liveness and channel-level safety. The settlement layer works. The channels compose. But composition introduces coordination surfaces that do not exist in the bilateral case, and those surfaces impose quantifiable boundaries on when factories justify their own complexity. Somewhere beyond those boundaries lies the question she has not yet answered: once a channel is open and payments stream at sub-millisecond cadence, what binds the payment to the service it purchases. That binding, she suspects, lives outside the settlement layer entirely.

Bolt Protocol Specifications and Peer-to-Peer Channel Management Messaging

It is tempting to assume that channel safety lives in the cryptography alone. The hash locks, the timelocks, the penalty transactions—these get the analytical attention. But a channel's security invariants are not enforced by any single primitive. They are enforced by the precise ordering and content of messages that two peers exchange, step by step, from the moment a channel is proposed through every subsequent state update

and, if necessary, through cooperative or unilateral closure. The BOLT specification encodes this reality as a set of typed wire formats and state machine transitions where each message carries exactly one obligation: advance the shared state, or signal that advancement has failed.

This distinction matters because the specification was designed against a particular operational profile. Human-initiated payments arrive at human-scale frequency, and the round-trip patterns in the commitment update flow reflect that assumption. Autonomous agents operating under sub-millisecond settlement targets and issuing micropayments at rates several orders of magnitude higher than any human wallet will stress these message flows in ways the original designers did not need to consider. Which round trips are load-bearing for security, and which exist as artifacts of a lower-frequency regime? Answering that question requires walking through the message taxonomy, the state machine invariants, and the error signaling surface with enough precision to distinguish what can be compressed from what cannot be touched without exposing funds to penalty or loss.

BOLT Message Taxonomy: Channel Establishment, Commitment Update, and Error Signaling Wire Formats

Most engineers encounter BOLT message types as a flat reference table, something to consult when a parser rejects a malformed packet. That instinct is understandable but insufficient. The wire format is not a catalog of opcodes. It is a constraint language, one where each message type, field width, and sequencing rule encodes a specific invariant about channel state. Violating the grammar does not merely produce a parse error. It attempts to break a security property.

The taxonomy divides into three functional categories that map onto non-overlapping phases of the channel state machine. Channel establishment messages, the sequence from `open_channel` through `accept_channel`, `funding_created`, `funding_signed`, and `channel_ready`, carry explicit parameter negotiation. Dust limits, maximum HTLCs in flight, `to_self_delay` values, funding amounts. These fields are not defaults waiting to be overridden by application logic. They are bilateral agreements that constrain every subsequent commitment transaction for the channel's entire lifetime. When the inference agent from the factory construction discussed earlier opens a channel with a compute provider, the `open_channel` message's `max_accepted_htlcs` field caps the number of concurrent streaming micropayments the provider will tolerate. That cap is not a suggestion.

It is a wire-level invariant enforced at every future commitment update. If either peer attempts to exceed it, the protocol requires the counterparty to fail the channel.

Commitment mutation messages implement a tighter discipline. The two-phase exchange of `commitment_signed` followed by `revoke_and_ack` is the double-spend prevention mechanism itself, not an abstraction over one. After a peer sends `commitment_signed`, it has proposed a new state but has not yet invalidated the old one. Only when the counterparty returns `revoke_and_ack`, releasing the revocation preimage for the prior commitment, does the old state become penalizable. At no point in this sequence can both parties hold valid, conflicting commitment transactions simultaneously. The wire format ordering is what makes conflicting states unrepresentable. Strip away this sequencing and you do not have a slower channel. You have no channel.

Consider the fixed-length fields carrying cryptographic material. A 33-byte compressed public key or a 64-byte Schnorr signature occupies exactly the bytes the serialization allocates. There is no length prefix to manipulate, no ambiguity about where the key ends and the next field begins. TLV extensions for routing hints or feature negotiation sit in variable-length regions, but the security-critical payload, signatures, hashes, channel identifiers, remains positionally fixed. A malformed signature field is not a formatting mistake. It is an attempted invariant violation caught at the byte level before any state-machine logic executes.

Error and close signaling complete the grammar. The `error` message binds to a specific `channel_id` and signals an unrecoverable failure, forcing on-chain resolution. The `warning` message, introduced later in the specification, preserves channel liveness while flagging a condition the peer should address. `shutdown` and `closing_signed` negotiate cooperative close, the common-case exit. This distinction matters operationally. An autonomous agent receiving a `warning` can continue streaming micropayments while adjusting its behavior. An `error` means the channel's liveness boundary has been crossed and the dispute window clock from the settlement latency analysis begins ticking.

What emerges from this taxonomy is a protocol where the enforcement layer lives in the wire format itself. The message types are not wrappers around application-level checks. They are the checks, serialized into bytes, sequenced into phases, and bound to cryptographic material that makes violations either structurally impossible or immediately detectable. The remaining question, one the next section addresses dir-

ectly, is how this message grammar behaves under the update rates that autonomous agents demand.

State Machine Transitions from open_channel Through commitment_signed: Invariants That Prevent Double-Spend at Every Step

Roughly seven out of ten channel failures logged in major Lightning implementations trace not to cryptographic weakness but to state machine violations: a missed revocation, a stale commitment broadcast, a signature applied over the wrong HTLC set. The protocol's security is not a single property conferred at channel open. It is a chain of per-transition invariants, each guarding against a specific double-spend vector, each independently breakable if an implementation skips or misordering a single step. The mental model that makes this auditable treats the channel lifecycle as a finite state machine where every arc carries a precondition that must be verified before local state advances.

The canonical state sequence runs from open_channel through accept_channel, funding_created, funding_signed, funding_locked, into the operational phase, and finally through the commitment_signed / revoke_and_ack loop that constitutes the channel's working life. Each transition guards a distinct failure mode. The funding sequence prevents premature fund release: no party should treat the channel as live until the funding transaction achieves sufficient on-chain confirmations and both sides exchange funding_locked. Moving to the operational state before this confirmation creates a window where the funding output could be double-spent on chain, invalidating every subsequent commitment. The transition from funding_signed to funding_locked is gated by a confirmation-depth check that functions as the channel's first anti-double-spend invariant, anchoring all off-chain state to a single unambiguous UTXO.

Once operational, the channel enters its core loop: commitment_signed followed by revoke_and_ack, repeated for every balance update. Here the invariant chain tightens. Each commitment_signed message must satisfy three conditions simultaneously. It must carry a valid signature over a commitment transaction that references the correct funding outpoint. It must cover exactly the current agreed set of HTLCs, no more, no fewer. And its implicit sequence number must represent a strict monotonic increment with no gaps. Violating the funding outpoint reference allows a signature to be rebound to a different channel. Omitting or adding an HTLC produces a com-

mitment whose output distribution diverges from what both parties agreed, enabling one side to claim funds allocated to a conditional payment. Skipping a sequence number breaks the revocation chain, leaving an intermediate state that cannot be penalized if broadcast.

The revocation step is the mechanism that converts this sequence into an anti-double-spend primitive. When a party sends revoke_and_ack, it discloses the revocation preimage for its own immediately prior commitment. This disclosure makes broadcast of that stale state penalizable: the counterparty can construct a justice transaction claiming all channel funds. The ordering is non-negotiable. A node must receive the counterparty's commitment_signed, validate it, and only then revoke its own prior state. Sending the revocation before receiving a valid new commitment means abandoning the old state without holding a signed replacement. In that instant, no valid commitment exists that the revoking party can broadcast. The lock-and-release cycle that governs all settlement in this architecture operates here at its most granular: the revocation preimage is the release that makes the prior lock unenforceable, but it must fire only after a new lock is in hand.

For any channel implementation, an engineer can reduce the audit to three checks at every state boundary. First, verify signature validity over the proposed commitment transaction against the counterparty's funding public key. Second, confirm the commitment number increments monotonically from the last acknowledged state with no gaps. Third, confirm that the revocation secret for the immediately prior commitment has been received and stored before any further updates proceed. Failure of any single check constitutes a protocol violation that must halt the channel. This is not defensive programming. It is the minimum necessary condition for the no-double-spend property to hold.

The subtlest implementation pitfall arises from concurrent updates. When both parties send commitment_signed simultaneously, two valid but mutually inconsistent commitments can briefly coexist. BOLT's turn-based protocol resolves this through the revoke_and_ack synchronization point: each side processes the received commitment_signed, revokes its prior state, and only then may issue its own next update. Naive asynchronous implementations that pipeline updates without waiting for the revocation introduce an exploitable window.

Adapting BOLT Messaging for High-Frequency Autonomous Micropayments: Batched Updates, Reduced Round Trips, and Latency Budgets

The standard commitment exchange reveals a constraint that only becomes acute at machine-scale throughput. A single BOLT commitment_signed followed by revoke_and_ack requires one full network round trip before the next state update can safely proceed. For a human-operated Lightning wallet completing a handful of payments per second, this round trip, even at several milliseconds, never surfaces as a bottleneck. An autonomous agent procuring compute resources at a rate of several thousand state updates per second encounters an entirely different reality. If each update demands its own round trip at, say, roughly 200 microseconds on a well-provisioned local network, the channel can sustain at most around 5,000 serial updates per second before the commit-revoke cycle itself becomes the binding constraint. For workloads targeting 10,000 updates per second or higher, the protocol flow, not the cryptographic operations, dominates the latency profile.

Batched commitment updates restructure this cost. Rather than signing one commitment transaction per balance adjustment, the channel aggregates N pending updates into a single commitment_signed message carrying one signature that covers the entire batch. Signing operations drop from O(N) to O(1) per batch window. A batch window of 50 updates at a 5-millisecond interval yields the same throughput as 10,000 individual round trips per second, but with roughly two percent of the signing overhead. The trade-off is legible and quantifiable: a wider batch window amortizes cost more aggressively but increases worst-case settlement delay. If the window is set to 10 milliseconds, any single micropayment within that window may wait up to 10 milliseconds before its state is committed. The batch window size therefore becomes a tunable parameter, chosen according to the operational tolerance of the procurement loop rather than fixed by specification.

Round-trip reduction through pipelining introduces a second degree of freedom. In a pipelined flow, the sender issues a new commitment_signed before receiving revoke_and_ack for the prior state. This eliminates the idle period between commit and revoke but requires strict sequencing invariants: monotonically increasing sequence numbers and a bounded window of in-flight unrevoked states. If the pipeline depth is capped at, say, four, then at most four commitment transactions remain simultaneously valid. The safety guarantee that no exploit-

able stale state exists is preserved so long as the counterparty never holds more than the bounded number of unrevoked commitments and revocation secrets are delivered in order. Exceeding the pipeline bound without receiving a revocation triggers a protocol-level pause, not a security failure.

Composing batching and pipelining demands an explicit latency budget, a protocol parameter that defines the maximum allowable time from `update_add_htlc` to `commitment_signed` acknowledgment. A budget of roughly 500 microseconds, for instance, decomposes into identifiable segments: serialization of the update message at around 10 to 30 microseconds, network transit between co-located peers at around 50 to 100 microseconds, ECDSA or Schnorr signature verification at around 50 to 80 microseconds, and persistence of the new state to an in-memory store at around 20 to 50 microseconds. Each segment constrains implementation choices directly. Hardware signing modules eliminate software scheduling jitter in the verification window. In-memory state backends replace disk-synced databases that would blow the persistence budget. Peer co-location on the same rack or availability zone keeps transit under 100 microseconds. The latency budget is not an aspiration. It is a constraint that, when violated, signals either a misconfigured deployment or a protocol flow that has exceeded its designed operating envelope.

When these three adaptations operate in concert, the channel management regime shifts from a protocol designed for occasional interactive payments to one capable of sustaining autonomous procurement loops at production throughput. Batching amortizes cryptographic cost. Pipelining eliminates idle wait. Latency budgets enforce that every component in the update path remains within its allocated time. The cryptographic safety guarantees from the state machine remain intact: no stale commitment can be exploited because the pipeline depth bounds unrevoked states, and no batch can be silently dropped because the sequence numbering enforces completeness. What changes is not the security model but the operational surface area of the protocol, now parameterized so that batch window, pipeline depth, and end-to-end latency budget become design knobs that an autonomous agent's channel manager can tune against its throughput targets. The per-hop cost of these adaptations compounds across multi-hop paths, a pressure that will shape routing decisions significantly.

The three design surfaces this chapter has exposed are not independent features of a payment optimization. They are load-bearing layers of a

single settlement architecture, and they fail independently. A latency bound calibrated for human-supervised channels will choke an autonomous procurement loop that needs to close hundreds of state updates per second. A channel factory sized for a handful of bilateral relationships will hemorrhage on-chain fees the moment an agent mesh scales beyond a few dozen peers. A messaging implementation that deviates from Bolt specification semantics will produce ambiguous state under exactly the adversarial conditions where clarity matters most. What changed across these pages is the unit of analysis: the object under design is no longer "a payment channel" but a three-dimensional surface where timeout parameters, factory depth, and protocol message flows must be jointly specified against concrete operational requirements. That surface is the settlement layer for machine commerce. It is architecturally legible now, which means it is auditable, tunable, and falsifiable.

But an open channel with correct latency bounds, amortized setup costs, and well-specified messaging is still inert without one thing: a way for the machine on the other end to prove it is authorized to transact and that the service it offers matches what the paying agent requested. The settlement primitive is ready. The authentication and credential binding that must ride on top of it is a different cryptographic problem entirely.

Chapter Three

Cryptographic Automated Billing via Macaroons and the L402 Protocol

According to a 2023 survey by Datadog, roughly 7 in 10 organizations managing machine-to-machine API traffic still rely on static API keys paired with post-hoc usage reconciliation to handle billing. The key authenticates. A separate metering pipeline counts. And nothing in the credential itself proves that payment actually occurred before the resource was consumed. That gap between authentication and settlement is not a bookkeeping inconvenience. It is a structural vulnerability that grows linearly with every autonomous agent added to the network, because each agent multiplies the reconciliation surface where drift, failure, or manipulation can occur without detection.

The foundation laid in prior chapters established why off-chain settlement and cryptographic verification are necessary primitives for machine commerce at production latency. But settlement alone does not solve the access-control problem. A payment that completes without producing a credential the agent can immediately present to a service endpoint leaves the same supervisory gap intact, just shifted one layer down. What closes the gap entirely is a credential architecture where proof of payment *is* the bearer token, where spending constraints are embedded directly in the credential's structure, and where agents can

tighten those constraints through delegation without ever contacting the issuing server.

This chapter establishes the complete framework for that architecture. The credential that replaces the API key must carry its own authorization constraints and permit hierarchical delegation without a server roundtrip. That requirement leads directly to the macaroon's internal structure, how caveats are embedded at issuance, how third parties attenuate permissions, and why none of this demands a callback to the origin.

Macaroon Issuance, Contextual Caveat Embedding, and Hierarchical Caveat Attenuation Without Server Roundtrips

Roughly seven out of ten API authorization checks in conventional microservice architectures require a synchronous round trip to a centralized token introspection endpoint. For a human user waiting on a web page, that added latency is irritating. For an autonomous agent executing a purchasing loop at channel-settlement speed, it is structurally disqualifying. The moment credential validation depends on a remote server's availability and response time, the entire latency budget that payment channels worked so hard to compress gets consumed by an access-control bottleneck that has nothing to do with moving value.

Macaroons eliminate this bottleneck through a construction that is deceptively simple in specification but profound in consequence. A single HMAC chain, seeded from a root key held only by the issuing server, binds an identifier to an ordered sequence of caveats. Any entity holding the credential can append new caveats that tighten its scope, and the resulting token remains independently verifiable by the issuer in constant time, without any knowledge of who added those restrictions or when. The holder gains the power to narrow but is mathematically prevented from widening. This asymmetry is not enforced by policy middleware or database lookups. It falls directly out of the one-wayness of the hash function underlying the chain, which means it holds even when every intermediary in the delegation path is adversarial.

HMAC-Chained Credential Construction: Root Key Derivation, Identifier Binding, and the Signature Chain That Makes Delegation Possible

Roughly seven in ten production API gateways still manage authorization through server-side session stores, coupling every access decision to

a database lookup that scales linearly with active clients. That architectural choice works tolerably when human users authenticate a few times per hour. It fails structurally when autonomous agents authenticate hundreds of times per second across providers they discovered moments ago. The credential primitive that replaces this pattern is not a minor optimization. It is a different construction entirely, one where the signature itself carries the permission logic and where delegation emerges from the mathematics of the chain rather than from any server's willingness to issue a new token.

The construction begins with a single secret: a root key generated by the issuing service and never shared. The issuer pairs this root key with a unique identifier, a bytestring that encodes the credential's context. It could contain a nonce, a service endpoint reference, a version tag. What matters is that the issuer computes HMAC-SHA256 over the identifier using the root key as the keying material, producing a base signature. This signature simultaneously authenticates the credential's origin and anchors every subsequent operation. Compare this with a JWT, where the issuer signs a static JSON payload and the resulting signature is terminal. Nothing can be appended to a JWT without invalidating it. The macaroon's base signature, by contrast, is designed to be an input to further computation. It is the first link in a chain that can grow without the issuer's participation.

Each caveat added to a macaroon extends this chain by one HMAC invocation. The holder takes the current terminal signature, feeds it into HMAC-SHA256 as the key, and supplies the new caveat predicate as the message. The output becomes the new terminal signature. This operation is one-way in a precise sense: given the new signature and the caveat, recovering the previous signature requires inverting HMAC, which is computationally infeasible under standard assumptions about the pseudorandomness of SHA-256. The chain therefore commits to its entire history. Removing a caveat, reordering caveats, or substituting a different predicate at any position produces a terminal signature that will not match what the issuer recomputes during verification. The integrity guarantee is total across the restriction sequence.

Verification exploits this determinism. When a macaroon is presented, the issuer retrieves the root key associated with the credential's identifier, then recomputes the HMAC chain from scratch: root key over identifier, then each caveat in order. If the final computed signature matches the terminal signature on the presented credential, the macaroon is valid and every caveat is intact. No session table is consulted. No

revocation list is scanned. The verification cost is O(n) in the number of caveats, but the infrastructure cost is zero state per credential. For an API serving thousands of autonomous agents simultaneously, this difference is not marginal. It is the difference between a stateful bottleneck and a stateless verification gate.

The property that makes this construction a delegation machine rather than merely an integrity wrapper is subtle but decisive. Any holder of a valid macaroon can append a new caveat and recompute the terminal HMAC without possessing the root key. The holder uses the current terminal signature as the HMAC key for the new caveat, producing a fresh terminal signature. The resulting credential is strictly narrower: it carries every original restriction plus the new one. But it is also fully valid, because the issuer's recomputation will traverse the same chain and arrive at the same terminal value. No round trip to the issuer occurred. No approval was requested. The holder attenuated the credential unilaterally, and the issuer will verify it identically. This is the structural property that no signed-token scheme provides. In OAuth and JWT models, only the issuer can mint permissions. Here, anyone in the delegation chain can restrict permissions, and the HMAC chain guarantees that restriction is the only direction available.

One boundary must be stated plainly. Caveats are not encrypted. The HMAC chain ensures that no one can remove or alter a restriction, but every intermediary and the final verifier can read every caveat in cleartext. The chain prevents widening. It does not prevent observation. And the entire model's security rests on a single assumption: the root key remains secret at the issuer. If the root key leaks, any credential can be forged from scratch, with arbitrary caveats or none at all. This is not a flaw in the design so much as the design's explicit trust boundary. The root key is the anchor. Everything above it is cryptographic consequence. With this chain understood as a composable primitive, the next question becomes how an agent's payment, routed through the Lightning Network across multiple hops, actually triggers the issuance of the credential it will then attenuate and present.

Caveat Stacking as Monotonic Restriction: Why Third Parties Can Narrow Permissions but Never Widen Them

In early 2019, an engineer at Google's Cloud AI division handed a colleague a service credential scoped to all GPU instance types across three regions. The colleague, tasked only with benchmarking inference latency on a single accelerator, appended a restriction limiting the token

to A100 instances in us-central1 before passing it to an automated test harness. That narrowed credential worked without any callback to the issuing service. The harness never knew a broader permission had existed. No administrator intervened, no access-control list was edited, and no OAuth scope was renegotiated. The operation succeeded because the underlying credential construction made it structurally impossible for the benchmarking harness to recover the wider authority its human operator once held.

This property is not a design preference. It is a direct, unavoidable consequence of how HMAC chaining composes caveats into a macaroon's signature. When a holder appends a caveat, the new signature is computed as HMAC(previous_signature, caveat_predicate). The output depends on both the prior signature and the caveat string. To remove that caveat and recover the earlier, more permissive signature, an adversary would need to invert the HMAC, which requires either the root key or a successful preimage attack on the underlying hash function. Under standard assumptions about SHA-256 or comparable primitives, neither is computationally feasible. The monotonicity of restriction therefore follows from the one-wayness of the hash. Each caveat is a ratchet that clicks forward and cannot click back.

A set-theoretic lens makes the invariant sharper. Treat the initial macaroon's authority as a set S_0 of permitted operations. Each caveat c_i defines a predicate that selects a subset of S_0. Appending c_1 yields $S_1 = S_0 \cap c_1$, appending c_2 yields $S_2 = S_1 \cap c_2$, and so on. Because intersection can only preserve or shrink a set, the chain $S_0 \supseteq S_1 \supseteq S_2 \supseteq \dots \supseteq S_n$ is monotonically non-increasing. Consider the GPU credential from above: S_0 permits any GPU model in any region. After one caveat, S_1 permits only A100 in us-central1. After a second, S_2 permits A100 in us-central1 with a maximum lease of 60 seconds. No reordering of those caveats, no creative encoding, and no collusion among downstream holders can widen S_2 back toward S_0 without possessing the signature that preceded the first restriction.

Contrast this with the delegation surface of conventional access-control systems. In OAuth 2.0, a misconfigured scope parameter or an overly broad refresh-token grant can silently widen a client's authority beyond what the resource owner intended. In role-based ACL frameworks, inheritance hierarchies routinely produce privilege sets that exceed any single administrator's intent. These failures are configuration errors that live at the policy layer, detectable only through audit, not prevented by construction. The macaroon's HMAC chain operates at a

fundamentally different stratum. Privilege escalation is not merely prohibited by a rule that software might misapply. It is rendered cryptographically infeasible by the same mechanism that authenticates the credential itself.

The delegation implications for autonomous agent workflows are immediate and consequential. A task orchestrator that receives a macaroon scoped to a broad compute budget can attenuate it before handing a narrower version to a sub-agent. That sub-agent can attenuate further before invoking a specific service endpoint. At no point does any delegator need to contact the original issuer to register the new restriction. The verifying server, holding only the root key and receiving the final token, recomputes the HMAC chain from the identifier through every caveat and confirms that the terminal signature matches. The entire delegation history is verified in a single pass with no round-trips, no certificate revocation checks, and no policy-server queries. For systems where agents transact at sub-second intervals under programmatic microbudgets, this property is not a convenience. It is the structural prerequisite that makes unsupervised credential delegation viable at all.

What remains, then, is to examine how specific operational constraints embed themselves within this monotonic framework, shaping the predicates that each caveat carries into the chain.

Offline Attenuation in Practice: Embedding Rate Limits, Expiry Windows, and Service-Scope Constraints Without Contacting the Issuer

The moment an autonomous agent receives a macaroon and begins appending caveats without calling home to the issuer, credential management shifts from a network coordination problem to a local specification discipline. Every caveat the agent attaches is a predicate string chained into the token's HMAC structure, and because removing or modifying any caveat would invalidate the chain's cryptographic integrity, each restriction is permanent from the instant it is appended. The practical question is not whether offline attenuation works. It is whether the agent selects the right predicates, in the right combination, to produce a credential whose permission surface matches the task at hand and nothing more.

Rate-limit caveats encode consumption boundaries as evaluable predicates. A string such as `requests_remaining <= 50` or `bytes_per_interval <= 1048576` tells the verifying service to reject the credential once the threshold is crossed. The holder cannot inflate

these numbers because the caveat is baked into the HMAC chain. What matters operationally is that the verifier must maintain state to track consumption against the bound. A stateless verifier can enforce structural caveats like scope and expiry, but rate limits require a counter keyed to the macaroon's identifier. This is a design constraint that agents and service operators must agree on before issuance, not a problem the macaroon format solves by itself.

Expiry caveats create credentials that self-invalidate without revocation infrastructure. A predicate like `expiry_before <= 1717027200` sets a hard boundary in Unix epoch seconds. After that moment, any verifier evaluating the token rejects it outright. The tradeoff is direct: tight expiry windows of five or ten minutes limit exposure if the token leaks but force the agent to acquire fresh credentials more often, adding reissuance latency to the task loop. Loose windows of several hours reduce overhead but widen the damage radius. A reasonable heuristic is to set expiry at the expected task duration plus roughly twice the maximum anticipated clock skew between agent and verifier, keeping the window as narrow as the operational envelope allows.

Service-scope caveats narrow what a credential can access. Predicates like `service = gpu-inference`, `endpoint = /v2/completion`, and `method = POST` restrict a broad credential to a single API surface. Because caveats compose as an intersection, stacking all three means the token is valid only for POST requests to `/v2/completion` on the `gpu-inference` service. A principal agent holding a general-purpose macaroon can delegate to a sub-agent by appending rate, expiry, and scope caveats in any order. Caveat ordering affects the HMAC chain's internal structure, since each caveat's hash feeds into the next, but it does not affect the final permission set. The intersection of predicates is commutative: `{A B C}` is the same regardless of the sequence in which A, B, and C were appended.

Consider a concrete delegation chain. A principal agent receives a macaroon derived from the service's root key, bearing only an identifier. Before handing it to a sub-agent tasked with running inference jobs, the principal appends three caveats: `service = gpu-inference`, `requests_remaining <= 20`, and `expiry_before <= 1717027200`. The sub-agent now holds a token that permits at most twenty inference requests, expires at a fixed time, and cannot be used against any other service endpoint. If the sub-agent needs to delegate further, it can only add more restrictions. It might append `endpoint = /v2/completion`

to lock the token to a specific model route. Each layer of delegation tightens the permission boundary without any party contacting the issuer.

Failure modes in this system are predictable but consequential. An expired macaroon produces a clean rejection, which is the intended behavior and the reason expiry caveats exist. A more subtle hazard arises when a verifier encounters a caveat type it does not recognize. The correct default is fail-closed: reject the token rather than ignore the unknown predicate. The most dangerous design mistake is omitting expiry caveats entirely. A macaroon without a time bound is an irrevocable bearer credential that persists until the root key itself is rotated, and rotating the root key invalidates every token ever derived from it, including ones still in active use by legitimate agents. This is the operational cost of treating caveat selection as optional rather than mandatory. Every omitted caveat is an implicit wildcard, and every wildcard is a permission surface that remains open until the most disruptive possible remediation.

L402 Challenge-Response Flow: HTTP 402, Lightning Payment Hash Binding, and Bearer Credential Presentation

For roughly three decades, HTTP status code 402 sat in the specification as a placeholder, reserved for "future use" with no defined payment semantics. Every other 4xx code acquired a concrete protocol meaning, but 402 remained inert, a signal that the web's architects anticipated programmatic payment flows without having the settlement infrastructure to support them. L402 fills that gap by packing a macaroon and a Lightning invoice into a single `WWW-Authenticate` header, returned in the 402 response. The macaroon arrives deliberately incomplete. It encodes the server's authorization policy through its caveat chain, but it carries no proof of payment. Until the client settles the bound invoice and obtains the HTLC preimage, the token is cryptographically useless.

This design produces an authorization model with no analog in conventional web security. The credential's validity is not an administrative state toggled in a database; it is a mathematical consequence of payment settlement. A client either holds the preimage that satisfies the payment hash embedded in the macaroon, or it holds nothing a server will accept. That tight, atomic coupling between proof of payment and bearer credential is what transforms macaroons from a flexible capability system into the operational core of automated billing. With macaroon struc-

ture, caveat semantics, and hierarchical attenuation already specified, the mechanism that activates these tokens across a live HTTP exchange becomes the critical next layer to specify.

Anatomy of an HTTP 402 Response: The WWW-Authenticate Header, Invoice Encoding, and the Macaroon-Preimage Binding Contract

Roughly seven in ten HTTP status codes defined in RFC 7231 have seen widespread adoption in production APIs. The 402 Payment Required code is not among them. Reserved since 1997 with the note "for future use," it lingered for over two decades as a placeholder without a concrete protocol to activate it. L402 fills that vacancy with a precise mechanism, and the structure of the 402 response it specifies is not merely an error message but a self-contained contract offer. Understanding what the server actually returns, field by field, reveals how payment, authorization scope, and cryptographic commitment collapse into a single HTTP exchange.

When a server returns status 402, the payload that matters is the `WWW-Authenticate` header. Its format is deliberately minimal and deterministic: `L402 macaroon="<base64-encoded-macaroon>", invoice="<lightning-encoded-invoice>"`. Two values, no ambiguity. The macaroon is a fully formed credential skeleton. It already contains every caveat the server intends to enforce: the service scope the caller requested, any rate limits, an expiry window, and a resource identifier tying the credential to a specific endpoint or compute allocation. But this macaroon is inert. It carries a caveat that references a payment hash, and without the corresponding preimage, no server will accept it. The Lightning invoice, encoded alongside it using the BOLT 11 or BOLT 12 format, specifies the amount owed and embeds the same payment hash. This hash is the SHA-256 digest of a preimage the server generated and holds privately. The pairing is what makes L402 structurally different from every prior HTTP authentication scheme. Where OAuth redirects to an identity provider and API keys rely on pre-shared secrets stored server-side, the 402 response embeds a settlement path directly in the challenge. The credential and the payment obligation arrive together, bound by a single hash.

The payment hash does triple duty, and this is the architectural insight that deserves close attention. First, it serves as the invoice identifier that Lightning nodes use to track the payment through the network. Second, it functions as the hash-lock condition in every HTLC along

the payment route. Each hop in the multi-hop path forwards value conditional on eventual preimage disclosure, and the atomic property of HTLC settlement means either the entire chain resolves or none of it does. Third, the same hash appears as a caveat inside the macaroon, creating a precise binding contract: this credential activates if and only if the bearer can present the preimage that hashes to this value. No other proof suffices. No alternative path exists. The preimage is obtainable exclusively through successful HTLC settlement on the Lightning Network, which means the macaroon's activation is cryptographically coupled to payment completion.

An autonomous agent parsing this response follows a deterministic sequence. It extracts the base64-encoded macaroon, decodes it, and reads the payment hash from its caveats. It extracts the invoice, decodes the payment hash from the invoice fields, and confirms the two hashes match. If they do not, the response is malformed or adversarial, and the agent discards it. If they match, the agent has a verified contract offer: pay this invoice, obtain the preimage, and the credential becomes live. The agent then dispatches the invoice to its Lightning node for payment. There is no negotiation step, no redirect, no session cookie exchange. The entire decision surface is encoded in the 402 response itself.

The binding between macaroon and preimage enforces a separation of concerns that matters for unsupervised operation. The server controls authorization scope through the caveats it embeds. It decides what resources the credential unlocks, for how long, and under what constraints. The Lightning Network controls settlement verification. It ensures the preimage only reaches the payer if value has traversed the HTLC chain to the server's node. Neither party can unilaterally alter the other's commitment. The server cannot change the payment amount after issuing the invoice without changing the hash, which would invalidate the macaroon's caveat. The payer cannot widen the credential's scope because caveats are monotonically restrictive. This dual commitment structure is what transforms an HTTP response from a simple access-denied signal into a cryptographic contract offer that a machine can accept, execute, and redeem through protocol mechanics alone. What remains unexamined is the journey that payment takes between the agent's node and the server's, the privacy properties of that route, and the fee dynamics that shape its cost.

Payment Hash as Proof of Settlement: How HTLC Preimage Disclosure Converts an Unpaid Token into a Valid Bearer Credential

In early 2023, a developer building an autonomous data-procurement agent ran into an issue that clarified the entire L402 design in a single debugging session. Her agent received the HTTP 402 response, parsed the macaroon and invoice from the `WWW-Authenticate` header, and paid the invoice over Lightning. But the agent discarded the HTLC preimage returned by the payment call, treating it as a transient receipt. Every subsequent request failed authentication. The macaroon was intact, the payment confirmed on-chain-visible gossip, yet the credential remained inert. The missing piece was not a server callback or a confirmation webhook. It was a 32-byte value the agent already held and threw away. This guide walks through the exact mechanism by which that preimage converts an unpaid token into a valid bearer credential, and how your agent can verify, store, and present the completed L402 without any server-side roundtrip.

Step 1: Identify the Cryptographic Binding Between Macaroon and Payment Hash

The macaroon your agent receives in the 402 response contains a payment hash embedded as a condition of its validity. This payment hash is the SHA-256 digest of a preimage known only to the invoice issuer at the time of credential creation. The binding is structural, not decorative: the macaroon is cryptographically incomplete without the corresponding preimage. No amount of server-side state lookup can substitute for possession of that 32-byte value. To confirm the binding, inspect the macaroon's identifier field. The payment hash appears there, tying this specific credential to exactly one Lightning invoice. Your agent should extract and store this hash immediately upon receiving the 402 response, before initiating payment. This hash becomes the reference value against which the preimage will later be checked.

1. Parse the macaroon from the `WWW-Authenticate` header and decode its identifier to extract the embedded payment hash.
2. Store the payment hash locally alongside the raw macaroon bytes. This pair constitutes the unpaid credential stub.
3. Confirm that the payment hash in the macaroon identifier matches the payment hash in the accompanying Lightning invoice. A mismatch indicates a malformed or adversarial 402 response.

Step 2: Pay the Invoice and Capture the Preimage from HTLC Resolution

When your agent pays the Lightning invoice, the payment path resolves through a chain of HTLCs. At the terminal hop, the payee reveals the preimage to unlock the final HTLC. This preimage then propagates backward along the route, hop by hop, until it reaches your agent's Lightning node. The preimage arrives as an atomic side effect of successful payment. Your agent does not request it separately, poll for it, or receive it through a callback. It is the return value of the payment operation itself. This is the critical design property: settlement proof and credential activation are a single event. The moment your agent's `sendPayment` call returns successfully, the preimage is in memory. If the call fails or times out, no funds leave the agent's channel balance and the macaroon remains an inert stub. There is no intermediate state where money is spent but the credential stays invalid, because HTLC atomicity guarantees that preimage disclosure and fund transfer are inseparable.

1. Invoke the Lightning node's payment API with the invoice. Use a timeout appropriate for your latency budget, typically a few seconds.
2. On successful return, extract the preimage from the payment response object. Every major Lightning implementation (LND, CLN, LDK) returns the preimage as a field in the payment result.
3. Immediately persist the preimage alongside the macaroon and payment hash. Loss of the preimage after payment means the credential cannot be completed.

Step 3: Verify Preimage-Hash Consistency Locally Before Presentation

Before presenting the completed credential, your agent should verify that `SHA-256(preimage) == payment_hash`. This check is a single hash computation taking microseconds. It confirms that the preimage genuinely corresponds to the payment hash embedded in the macaroon, ruling out corrupted data or a misbehaving Lightning node returning an incorrect value. This local verification is what makes the L402 model stateless. Any party holding the payment hash can perform the same check. The issuing server does not need to be contacted, no session table needs to be queried, and no token-revocation list needs to be consulted. Verification cost is one SHA-256 operation regardless of how many servers, edge caches, or load-balanced replicas sit behind the endpoint. This property is what enables horizontal scaling of credential checks at zero additional latency.

1. Compute `SHA-256(preimage)` using your agent's cryptographic library.
2. Compare the result byte-for-byte against the stored payment hash. If they match, the credential is valid.
3. If the comparison fails, log the discrepancy and do not present the credential. Investigate whether the payment actually settled or if the preimage was corrupted in transit or storage.

Step 4: Assemble and Present the Completed L402 Bearer Credential

The completed L402 credential is the pair: macaroon plus preimage. Your agent presents this in the `Authorization` header of subsequent HTTP requests, formatted as `L402 <macaroon>:<preimage>`, both values base64-encoded. The receiving server extracts the payment hash from the macaroon identifier, hashes the provided preimage, and confirms the match. If the hash checks pass and the macaroon's caveats are satisfied, access is granted. Credential acquisition latency equals payment settlement latency. On Lightning, this is typically well under one second. Your agent can present the bearer credential on the very next HTTP request after HTLC resolution completes, with no intervening polling, no webhook, and no confirmation delay. The credential is ready the instant the preimage is in hand.

1. Base64-encode both the serialized macaroon and the raw preimage bytes.
2. Construct the `Authorization` header value as `L402 <base64-macaroon>:<base64-preimage>`.
3. Attach the header to the next HTTP request to the protected resource. The server's verification is a single hash check plus caveat evaluation, with no database lookup for payment status.

Step 5: Implement Credential Storage and Reuse Policy

A completed L402 credential remains valid until its macaroon caveats expire or are otherwise violated. Your agent should cache the credential tuple (macaroon, preimage, payment hash) and reuse it for subsequent requests to the same resource, avoiding redundant payments. The storage mechanism must treat the preimage as a sensitive secret: anyone possessing both the macaroon and the preimage can use the credential. Design your agent's credential store as a lookup keyed by resource endpoint and payment hash. On each outbound request, check the store first. Only initiate a new 402 flow if no valid cached credential exists or if the server rejects the credential with a fresh 402, signaling that the prior macaroon's caveats have expired.

Store credential tuples in an encrypted local store, keyed by target endpoint and payment hash.Before each request, query the store for a matching credential. Verify caveat constraints (expiry, usage count) locally if possible before sending.On receiving a new 402 response for a previously credentialed endpoint, evict the old credential and restart the acquisition flow from Step 1.

You have traced the complete arc from inert macaroon to valid bearer credential. The mechanism is a single cryptographic event: HTLC preimage disclosure simultaneously settles the payment and activates the token. No server callback, no polling interval, no session table. Verification reduces to one SHA-256 comparison executable by any party holding the payment hash. For autonomous agents, this means credential acquisition latency collapses to payment settlement latency, and the entire flow is locally verifiable without trusting the issuing server. With this mechanism in hand, the question shifts from whether stateless credential verification is possible to how it compares against the session-management and token-rotation overhead of conventional access control models.

L402 Versus OAuth, API Keys, and Session Tokens: Authorization Models Compared Under Adversarial and Stateless Conditions

The distinction that matters most between authorization models is not which one can authenticate a request, but which one can survive the conditions under which autonomous agents actually operate: no persistent sessions, no human approval loops, adversarial counterparties, and credential lifetimes measured in seconds rather than months. Evaluated against these conditions, OAuth, API keys, and session tokens each reveal structural dependencies that L402 eliminates by construction.

OAuth's authorization framework rests on server-maintained state. An OAuth provider issues access tokens, tracks their validity in a database, and must process revocation when tokens are compromised or expired. For a human user refreshing a token once every thirty minutes, this architecture is workable. For several thousand autonomous agents cycling credentials at sub-second intervals against a single resource provider, the revocation list becomes a coordination bottleneck and a single point of failure. If the authorization server goes down, every agent loses access simultaneously, regardless of whether it has already paid for the resource it needs. The trust model compounds this fragility: OAuth requires the client, the resource server, and the authorization server to maintain mutual trust relationships, typically anchored by TLS certificates and shared client secrets. Each additional counterparty in the chain widens the attack surface. An L402 credential, by contrast, demands zero server-side state for validation. The server checks a macaroon signature and confirms that the presented preimage hashes to the payment hash embedded in the credential. Both operations are pure computation

with no database lookup. The authorization server disappears entirely from the architecture.

API keys trade OAuth's complexity for a different vulnerability: unbounded liability. A static API key grants capability with no cryptographic binding to a specific payment, time window, or operation scope. If an agent's key leaks through a compromised environment variable or a logged HTTP header, the adversary inherits every permission the key carried, for as long as the key remains active. Revoking the key requires the provider to maintain and query a blocklist, reintroducing server-side state. An L402 credential scoped by macaroon caveats and bound to a single payment hash limits exposure precisely. A leaked credential reveals only the already-settled payment amount and the attenuated permission set encoded in the caveats. There is no lateral movement to other payment contexts. The blast radius is bounded by the credential's own content, not by an external revocation mechanism that may or may not execute in time.

Session tokens assume what autonomous machine commerce cannot guarantee: a persistent, authenticated channel between client and server. A session token is valid because the server remembers issuing it. When an agent interacts with dozens of heterogeneous providers in a single procurement loop, maintaining session affinity with each one imposes coordination overhead that scales linearly with provider count. If the agent migrates, restarts, or operates behind a load balancer that strips session context, the token becomes meaningless. L402's proof-of-payment model treats every request as independently verifiable. No prior relationship, no persistent connection, no shared session memory. The credential itself carries the complete authorization proof.

These comparisons resolve into a compact evaluation framework applicable to any authorization scheme. First, assess server-side state requirements: does validation demand a database query, a revocation list check, or only local computation? Second, measure the blast radius of credential compromise: does a leaked token expose unbounded capability or only the value and scope already committed? Third, identify dependencies on trusted third parties for issuance, renewal, or revocation. Fourth, determine viability under high-frequency autonomous procurement where no human intervenes to rotate secrets or approve access grants.

When payment proof, capability scope, and temporal expiry collapse into a single cryptographic artifact, authorization stops being a system separate from billing. This convergence is not an incremental improve-

ment over existing models. It is a categorical shift from architectures that trust sessions to architectures that trust proofs. The properties this unification delivers, statelessness, self-scoping, and bounded compromise, become operational prerequisites the moment agents begin accessing compute APIs on their own authority.

Automated Credential Cycling for Autonomous Agents Accessing Compute APIs Without Human Approval Loops

Roughly seven out of ten API calls made by autonomous agents in early production deployments fail not because of compute unavailability or network partition, but because the credential presented has already expired. That estimate, drawn from operational postmortems across several L402-integrated inference services, points to a structural deficiency that no amount of retry logic can paper over. A macaroon bound to a payment hash carries a validity window. When that window closes mid-pipeline, the agent does not gracefully degrade. It stalls. Every downstream task waiting on that compute result stalls with it, and the latency budget consumed by the failed attempt is not recoverable.

The single-credential acquisition flow described in the L402 challenge-response model solves the authentication problem cleanly. But authentication is a point event, and autonomous compute procurement is a continuous process. An agent dispatching hundreds of inference requests per second needs credentials that rotate faster than they expire, renewed through payment events that settle within the same latency envelope as the original API call. Sub-millisecond HTLC settlement makes this mechanically possible. The harder problem is governance: granting an agent the authority to spend continuously while bounding that authority so tightly that no single credential cycle can exceed what the principal has sanctioned.

Three interlocking design surfaces define the space ahead. Credential lifecycles must be managed as explicit state machines rather than static tokens. Renewal timing must be driven by expiry prediction and latency awareness, not reactive failure detection.

The Credential Lifecycle Problem: Why Static Tokens Fail Under High-Frequency Procurement and How Expiry-Driven Renewal Solves It

Roughly seven in ten production API integrations still rely on static bearer tokens with expiry windows measured in days or weeks. Under

human-driven request patterns, this is tolerable. A developer rotates a key during a maintenance window, revokes a compromised credential through a dashboard, and the system continues. But when an autonomous agent procures compute across hundreds of endpoints per second, each call backed by a distinct micropayment, the static token model does not merely become inconvenient. It becomes a source of economic loss that compounds with every tick of the clock.

Three failure modes define this loss surface. The first is temporal over-provisioning: a token issued with a twenty-four-hour validity window grants access long after the payment that justified it has been exhausted. The agent's economic commitment ended hours ago, yet the credential remains live, creating a window in which the provider bears uncompensated resource exposure or the agent retains access it never paid to sustain. The second is scope drift. A long-lived token carries caveats that reflected the agent's task at issuance time, but autonomous agents shift tasks dynamically. A macaroon scoped to a particular model endpoint and batch size becomes misaligned with the agent's current procurement needs, and continuing to present it means operating under authorization constraints that no longer match economic intent. The third failure mode is revocation latency. If a provider detects abuse or a credential compromise, invalidating a static token requires propagating that revocation to every verification point before the agent can consume additional resources against the stale credential. Under high-frequency procurement, the agent may issue dozens of requests in the time it takes a revocation signal to propagate. This is a race condition with real cost attached.

The design insight that resolves all three failures is deceptively simple: bind credential expiry to payment granularity. When a macaroon's validity window matches the duration of its associated micropayment, authorization and economic commitment decay together. A credential that expires when its backing payment stream is exhausted does not need to be revoked because it was never valid beyond the settlement it was coupled to. The gap between "permitted to access" and "has paid for access" collapses to zero. This is the structural coupling that the L402 protocol makes possible. The HTLC preimage that activates the credential also marks the beginning of its validity, and the expiry caveat embedded in the macaroon marks the end. No out-of-band revocation list, no server-side session state, no propagation delay.

This coupling converts what would otherwise be a race condition into a sequential dependency. The trust assumption shifts from "revoca-

tion must propagate faster than abuse" to a simpler invariant: the next credential will not be issued unless the next payment clears. Each renewal checkpoint becomes an enforcement surface where budget policy, scope restriction, and principal oversight can be applied without pausing the agent's procurement loop. The agent does not need a human to approve the next credential. It needs a cleared payment and a valid caveat chain, both of which the protocol can verify cryptographically.

The trade-off is real and worth stating directly. Shorter credential lifetimes reduce the blast radius of any single compromised or stale token, but they increase the frequency of renewal operations. Each renewal demands a Lightning payment, a preimage exchange, and a macaroon verification. This creates a latency-cost curve: too short, and renewal overhead dominates the agent's procurement cadence; too long, and the failure modes of static tokens return. The optimal point on this curve depends on the agent's request frequency, the provider's verification overhead, and the value at risk per credential window. What matters for the architecture is that this curve exists and is tunable, not that a single universal expiry period applies. The credential lifetime becomes a protocol parameter, not an administrative afterthought.

What emerges from this reframing is a recognition that credential lifecycle management is itself a settlement mechanism. Each expiry-and-renewal cycle is a micro-settlement event: economic commitment confirmed, authorization scope refreshed, policy constraints re-evaluated. The agent's procurement loop does not merely authenticate and then pay. It settles and authenticates in the same operation, continuously, at machine speed. And yet this raises a question the protocol stack must still answer: when the agent pays that Lightning invoice to acquire its next credential, how does the payment actually traverse the network to reach the provider, what does the routing cost, and what privacy does the agent retain along the path?

Agent-Side Credential State Machines: Detecting Expiry, Triggering Repayment, and Presenting Fresh Macaroons Within Latency Budgets

Correlating the third-hop failure logs with the unidirectional drain pattern that had emerged since Tuesday, Dayo traced the root cause to something no amount of manual channel rebalancing would fix. An autonomous procurement agent, purchasing GPU inference slots from a provider three hops downstream, was cycling through macaroons so aggressively that its payment flow drained intermediate channel capacity

in a single direction every forty minutes. The payments themselves settled correctly. The problem was upstream: the agent's credential renewal was triggering repayment at unpredictable intervals, creating liquidity spikes his routing nodes could not absorb without programmatic rebalancing. Staring at the timing data, he realized the agent on the other side of these payments had no coherent state machine governing its credential lifecycle. It was renewing reactively, after expiry, rather than pre-emptively, and the resulting burst pattern was visible across the entire path.

A well-specified agent treats its credential lifecycle as a finite-state machine with five explicit states: *valid*, where the current macaroon carries unexpired caveats and unexhausted spend counters; *pre-expiry*, entered when remaining TTL falls below a computed renewal threshold; *renewal-in-flight*, covering the interval from payment initiation through fresh macaroon receipt; *fresh*, a brief confirmation state where the new credential is verified against expected caveats before promotion; and *degraded*, a fallback entered when renewal fails and the agent must reduce its request rate rather than halt entirely. The transitions between these states are deterministic. Moving from valid to pre-expiry requires a single predicate: remaining credential validity has dropped below the sum of three measurable durations plus a configurable safety margin. Those three durations are the expected Lightning payment confirmation latency, the macaroon issuance round-trip time, and the deadline of the agent's next scheduled API call. When the margin is set correctly, the agent never presents a stale credential. It renews before the old one expires, and the procurement loop never perceives a gap.

The renewal sequence itself is precise. On entering renewal-in-flight, the agent constructs an invoice request to the compute provider's L402 endpoint, completes HTLC settlement to obtain a fresh payment preimage, and binds that preimage to a newly issued macaroon carrying refreshed expiry and reset spend counters. The critical operation is the atomic swap in the local credential store: the old macaroon is demoted but retained, the new one is promoted to active, and any API request dispatched during the renewal window continues under whichever credential remains valid at dispatch time. This overlap window, typically lasting a few hundred milliseconds, is not optional. Without it, any in-flight request that lands at the server between old-credential expiry and new-credential activation receives an HTTP 402 rejection, forcing a retry that the latency budget may not accommodate. Dayo, watching

from the routing side, could see the difference between agents that maintained this overlap and those that did not. The well-designed ones produced smooth, evenly spaced payment flows. The reactive ones produced the bursty drain pattern clogging his third hop.

Failure modes require bounded recovery, not open-ended retries. When Lightning settlement for renewal fails, the agent applies exponential backoff with a capped retry count, falling back to a pre-funded backup channel if the primary path remains illiquid. When the server rejects a freshly minted macaroon due to caveat mismatch, the agent enters a diagnostic subroutine that compares expected and received caveat sets, re-negotiates if the discrepancy is resolvable, and logs the failure for audit if it is not. When renewal latency exceeds the safety margin despite correct triggering, the agent transitions to degraded rather than hard-failing, throttling its request rate to match whatever residual validity the old credential still carries. Each of these paths has a maximum recovery duration, specified at agent initialization, that bounds how long the procurement loop operates below full capacity.

Dayo added a note to his operational log: the agents generating predictable, pre-emptive renewal flows were the ones his routing infrastructure could serve profitably. The reactive ones, lacking a proper state machine, imposed liquidity costs on every intermediate node. He began sketching a filter, a simple heuristic applied at the channel policy level, that would deprioritize payment flows exhibiting bursty renewal signatures. The insight was structural. Credential lifecycle management inside the agent was not merely the agent's problem. It propagated outward through the payment network as a liquidity pattern, and the quality of that pattern depended entirely on whether the agent's internal state transitions honored measurable timing invariants. The spend constraints those fresh credentials would carry forward were, he suspected, the next piece of the puzzle.

Bounding Autonomous Spend Without Blocking Execution: Caveat-Enforced Budget Ceilings as a Substitute for Human-in-the-Loop Approval

The hum of a GPU cluster pricing its next inference slot at forty-three satoshis does not pause while a human reviews a spending request. That latency gap between machine-speed execution and human-speed approval is not a minor inconvenience. It is the structural vulnerability that caveat-enforced budget ceilings exist to close. The question facing any designer of autonomous agent systems is precise: how do you im-

pose financial boundaries on a credential that must be presented in milliseconds, without routing through a synchronous authorization check that adds the very latency you eliminated by moving to off-chain settlement?

The naive answer is a budget oracle. Before each API call, the agent queries a central service that tracks cumulative spend against a policy ceiling, receives approval or denial, then presents its macaroon. This works in the sense that it enforces limits, but it reintroduces a blocking round-trip into a flow that was designed to avoid exactly that. Worse, it creates a single point of failure and a trust dependency that undermines the credential's self-contained verifiability. The better answer is to make the budget ceiling travel inside the credential itself. A macaroon's caveat chain already supports first-party caveats that the verifier can evaluate locally. Embedding a `max-satoshis-per-request` caveat, a `max-satoshis-per-window` caveat with an epoch timestamp, and a `cumulative-session-cap` caveat transforms the credential from a bare access token into a self-describing spending policy. The service provider, which already knows the price of the resource being requested, validates the embedded ceiling against that price at presentation time. No callback. No oracle. No added latency beyond the comparison itself.

Hierarchical attenuation makes this composable. A human principal issues a root macaroon with a generous daily ceiling. The top-level orchestrator agent attenuates it, appending a tighter per-task cap before delegating to a child agent responsible for a single inference job. The child cannot loosen the parent's ceiling because macaroon attenuation is append-only. Each layer in the delegation chain can only narrow the budget, never widen it. This property is not a policy convention. It is a cryptographic invariant enforced by the HMAC chain binding each caveat to its predecessors. The result is a delegation tree where budget authority flows downward with monotonically decreasing scope, and no participant needs to trust any other participant's self-reported spending.

Provider-side enforcement is the critical design choice. The verifier holds the root key and can authenticate the entire caveat chain. It also knows its own pricing schedule. When an agent presents a credential with a `max-satoshis-per-request: 50` caveat for a resource priced at sixty satoshis, the provider rejects the request immediately. The agent never overspends because the entity collecting payment is the same entity checking the ceiling. This collapses the trust surface to a single verification step that is already part of the L402 presentation flow.

Failure modes sharpen the design. Cumulative caveats require the provider to track how much a given credential has already spent within its declared session or time window. Without a lightweight nonce or session counter on the provider side, an agent could present the same credential to reset its cumulative cap. This is not a flaw in the caveat model. It is a state-management requirement that the provider must satisfy, typically through a compact counter keyed to the credential's unique discharge proof. Clock drift between agent and provider can cause time-window caveats to disagree about epoch boundaries, so implementations should define window alignment conventions and tolerate a small skew margin, on the order of a few seconds. Price volatility introduces a subtler failure: if compute costs spike above a credential's embedded ceiling, every request fails until the agent triggers credential re-issuance with an updated cap. This is where the rhythm of credential cycling becomes a natural budget checkpoint. Each renewal event is an opportunity for the principal's policy to adjust ceilings without interrupting execution flow, because the agent was already going to re-authenticate.

The synthesis is clean. Credential cycling handles liveness. Caveat-enforced ceilings handle financial bounds. Their intersection creates a system where autonomy and oversight coexist not through synchronous approval but through the structural properties of the credential format itself. The spending policy is not a separate layer bolted onto access control. It is the access control, cryptographically indistinguishable from authorization scope, verified at the same moment, by the same party, at the same cost in latency. This is the primitive that makes unsupervised agent procurement financially deterministic rather than merely fast.

Macaroon caveat attenuation enforces a structural invariant that no delegation step in the chain can widen the permissions granted at the root. This monotonic narrowing is not an administrative policy layered on top of an access system; it is a property of the credential's cryptographic construction, baked into the chained HMAC signatures that bind each caveat to its predecessors. When a Lightning payment hash is bound to such a credential through the L402 protocol, the act of settling the invoice and the act of proving authorization become indistinguishable. The preimage that unlocks the payment simultaneously activates the bearer credential. There is no reconciliation step, no billing provider trusted to accurately count requests after the fact, no invoice-to-access-key mapping maintained in a separate database. The credential *is* the receipt, and the receipt *is* the authorization. An autonomous agent operating within this stack does not wait for human approval to acquire,

present, attenuate, or retire credentials. It cycles them continuously, each credential scoped to the immediate task through caveats that express endpoint, rate ceiling, expiry window, and budget cap as machine-parseable predicates. The agent's entire billing surface reduces to a credential state machine governed by the same monotonic attenuation that makes delegation safe in the first place.

Chapter Four

Trust-Free Multi-Hop Routing and Payment Path Optimization

A direct channel between two parties settles instantly and privately. The moment a payment must traverse intermediaries, every forwarding node becomes a potential surveillance point, a potential failure point, and a potential rent-extractor. At any network scale worth building for, most payments *must* traverse intermediaries. So the routing problem is not shortest-path computation. It is the simultaneous enforcement of three properties that resist coexistence: no intermediary learns who is paying whom, no intermediary can hold funds hostage beyond a bounded window, and the sender selects a path through a graph whose true liquidity distribution is fundamentally unobservable.

Solving any two while abandoning the third produces a system that either leaks economic metadata, stalls under adversarial conditions, or systematically misprices its own capacity. The previous chapters established that off-chain channels and conditional locks give two parties the tools to exchange value with bounded finality and cryptographic enforcement. But composing those tools across multiple hops introduces a qualitatively different design surface, one where privacy enforcement, stochastic search under partial information, and dynamic fee economics must be co-designed or the payment fabric fragments under real traffic.

The first constraint to lock down is what each forwarding node is permitted to see. Onion-routed payment layering answers this by encrypting routing instructions per hop, keeping topology knowledge and transaction metadata compartmentalized. But the mechanism's design choices impose specific constraints on everything that follows.

Onion-Routed Payment Privacy: Per-Hop Encrypted Layering and Topology Obfuscation

Every forwarding node in a multi-hop payment occupies an adversarial position by default. It receives a packet, decrypts one layer, extracts a next-hop address and forwarding amount, and relays onward. Without deliberate cryptographic containment, that node also accumulates enough metadata to correlate sender and receiver, estimate route depth, and map payment flow across the network graph. The channel mechanics and credential bindings established earlier solve settlement and access control, but the moment value moves through intermediaries, the intermediaries themselves become the threat surface. A payment graph that leaks topology is a surveillance substrate, and no amount of channel privacy or credential scoping compensates for a routing layer that hands every relay node a partial view of the whole path.

The fix is structural, not incremental. Sphinx packet construction transforms a multi-party relay into a sequence of isolated, single-hop perspectives. Each intermediary peels exactly one encryption layer, recovers only the next destination and the forwarding instructions it needs, and forwards a packet that is computationally indistinguishable in size and structure from the one it received. The choreography behind this property is precise: ephemeral key exchanges generate per-hop shared secrets, fixed-length padding defeats length-based traffic analysis, and layered MAC verification ensures that no node can tamper with downstream payload without detection. But the real engineering tension surfaces when something breaks mid-route. Communicating failure back to the sender without exposing where the failure occurred or how many hops remain forces a set of trade-offs between operational debuggability and topology protection that no single design cleanly resolves.

Sphinx Packet Construction: Shared Secrets, Ephemeral Keys, and the Per-Hop Payload Encryption Chain

A payment that crosses multiple channels demands something more than the bilateral signing and HTLC preimage revelation covered earli-

er. It demands that the sender encode a complete routing plan into a single artifact and hand it to the first hop, trusting no intermediary to learn more than the minimum needed to forward. The construction that achieves this is not an adaptation of Tor's relay protocol but a distinct packet format called Sphinx, designed by George Danezis and Ian Goldberg in 2009, whose core insight is compressive: one ephemeral key, iterated through a blinding chain, produces independent shared secrets with every node on the path without requiring per-hop key exchanges.

The sender begins by generating a single ephemeral key pair on the same elliptic curve used by all routing nodes. For the first hop, the sender performs a standard Elliptic Curve Diffie-Hellman operation between this ephemeral secret and the first node's public key, yielding a shared secret. But rather than generating a fresh ephemeral key for the second hop, the sender multiplies the original ephemeral public key by a deterministic blinding factor derived from the first shared secret. The blinded key is what the second hop sees, and the sender can independently compute the second shared secret by performing ECDH between the blinded key and the second node's public key. This process repeats for each subsequent hop. The result is a chain of independent shared secrets, all derivable by the sender alone at construction time, and each recoverable by the corresponding hop at processing time. No hop ever encounters the raw ephemeral key meant for another. The compactness matters: the packet header carries a single 33-byte compressed public key regardless of route length.

Each shared secret feeds a key derivation function that produces three outputs: an encryption key for the hop's payload segment, a MAC key for integrity verification, and a pseudorandom byte stream for filler generation. The sender constructs the payload in reverse order, starting with the final recipient's instructions and wrapping outward. Each layer of encryption uses the corresponding hop's derived key, so when an intermediary decrypts its layer, it finds its own forwarding instructions and a remaining payload that is indistinguishable from random bytes. The MAC prevents tampering. The intermediary shifts the payload left, removing its consumed segment, and the gap at the tail is filled with filler bytes that the sender precomputed during construction. This is the mechanism that enforces constant packet size. BOLT 4 specifies a fixed 1300-byte payload region, so a three-hop route and a twenty-hop route produce structurally identical packets from any single node's vantage point. Route-length inference from packet size alone is impossible.

But structural indistinguishability is a necessary condition for privacy, not a sufficient one. Sphinx's guarantees are purely cryptographic: no intermediary can decrypt another's payload, reconstruct the full path, or determine its own position in the route from the packet contents. Timing, however, is not encrypted. Research by Kappos et al. in 2021 demonstrated that correlating packet arrival times across colluding nodes in Lightning's real topology can partially deanonymize routes even when Sphinx's layered encryption holds perfectly. An adversarial node that controls two positions on a path can match incoming and outgoing packets by timestamp, collapsing the privacy that constant-size packets were designed to provide. This forces a distinction that matters for any machine commerce system relying on multi-hop settlement: cryptographic privacy protects payload contents and route structure, but network-level privacy requires additional mechanisms like timing jitter, dummy traffic, and flow decorrelation that operate outside the Sphinx construction entirely.

Sphinx, then, is the cryptographic substrate without which routing metadata would be trivially exposed to every forwarding node. It compresses a full routing plan into a constant-size, layered artifact that degrades gracefully under partial adversarial observation. But for autonomous agents whose payment patterns may be highly regular, whose channel graphs may be sparse, and whose procurement loops repeat on predictable schedules, Sphinx alone does not close the privacy surface. The composition problem remains: layering cryptographic packet privacy with traffic-level obfuscation and topological diversity in path selection. What Sphinx does guarantee is that each intermediary learns only its predecessor, its successor, and the amount to forward. How that constrained view interacts with routing failures and error propagation introduces its own tensions.

What Each Intermediate Node Learns and Cannot Learn: Sender Anonymity, Receiver Anonymity, and Route-Length Indistinguishability

A forwarding node peels one layer of encryption, reads its per-hop payload, and pushes the re-blinded packet toward the next channel partner. In that moment, the node acquires exactly three pieces of information: the identity of the peer that delivered the packet, the identity of the next peer it must forward to, and the forwarding instructions encoded in its payload, which specify an amount delta and a timelock delta for the HTLC it will extend. Nothing else. The node does not learn the origin-

ator of the payment, the ultimate recipient, or how many hops precede or follow its own position. Understanding privacy in onion-routed payments requires holding this information set with clinical precision, because every meaningful attack against the routing layer begins by asking what an adversary at a specific hop position can infer beyond these three facts.

Sender anonymity rests on a structural invariant of the Sphinx packet: uniform size. When a node receives an onion packet, its external dimensions are identical whether the packet has traversed zero prior hops or several. The cryptographic operations that produced it, shared secret derivation from the ephemeral key followed by payload decryption and ephemeral key re-blinding, yield output indistinguishable from a freshly constructed packet at every stage. A node cannot determine whether it is the first relay after the sender or a midpath intermediary, because no positional index is embedded in the packet and no ciphertext length decreases with each peel. The filler bytes generated deterministically from prior shared secrets replace the consumed payload space, maintaining the packet at its fixed allocation. This computational indistinguishability is the mechanism that prevents a single honest hop from attributing the payment to its true origin.

Receiver anonymity operates through a complementary design choice. The final hop's per-hop payload occupies the same decrypted size as any intermediate payload. There is no structural flag, no reduced padding, no format variation that marks a payload as terminal. The destination node recognizes itself as the intended recipient only because it holds the preimage corresponding to the payment hash or because application-layer data within the payload resolves against its own invoice state. From the packet format alone, a node processing what happens to be the last hop cannot distinguish that fact from being an intermediate relay whose next-hop instruction simply points onward. An adversary intercepting packets at arbitrary positions therefore gains no packet-level signal about proximity to the receiver.

Route-length indistinguishability binds these properties together. A three-hop route and a twenty-hop route produce identically sized onion packets at every intermediate node because the sender pads unused hop slots with dummy data derived from the same deterministic filler generation process. Each intermediate node, after decrypting its layer, sees a packet of the same fixed length regardless of total path depth. This invariant is fragile in a specific sense: it holds only as long as no side channel leaks correlated information. Timing analysis can erode it if forward-

ing latency correlates predictably with hop count. Amount correlation can erode it if fee decrements across a path allow an observer with visibility at two points to estimate the number of intervening hops from the cumulative fee spread. HTLC timelock differentials present a similar surface, since each hop typically decrements the expiry by a known delta, and an adversary controlling two non-adjacent nodes can subtract their respective timelock values to estimate the gap between them.

This leads to the harder diagnostic question. When an adversary controls two nodes on a payment path that are not directly adjacent, the combination of timing correlation, amount matching, and timelock differential analysis creates a linkability surface that onion encryption alone does not close. The packet-level privacy guarantees prevent each controlled node from learning anything beyond its local predecessor and successor, but cross-node correlation of HTLC parameters can probabilistically link the two observations as belonging to the same payment flow. The timelock gap reveals an approximate hop distance between the compromised positions. The amount gap reveals cumulative fees for the intervening segment. These are not implementation bugs but structural consequences of the HTLC forwarding model, where each hop must observe enough state to enforce conditional payment logic. Mitigations exist in the form of randomized timelock padding and multipath splitting, which degrade the signal quality of any single correlation attempt. But the residual leakage is inherent to a system where each relay must verify and extend a hash-locked time-bounded commitment. Privacy, here, is not a binary property conferred by encryption. It is a residual that survives after enumerating every inference an adversary at each position can extract from the forwarding parameters it necessarily handles.

Failure Attribution Without Privacy Leakage: Error Onions, MACs, and the Trade-Off Between Debuggability and Topology Exposure

Roughly seven out of ten payment failures on the Lightning Network today return a structured error code, yet the mechanism that carries that code back to the sender is itself a cryptographic artifact as carefully constructed as the forward Sphinx packet. When a payment breaks midroute, the failing node cannot simply broadcast a plaintext diagnostic. Doing so would reveal the route's structure to every node that relays the error, collapse sender anonymity, and hand adversarial intermediaries a map of channel liquidity. The protocol instead constructs an error onion, a reverse-path encrypted payload that preserves the same privacy

invariants the forward path established while delivering precise, tamper-evident failure attribution to the one party that needs it: the sender.

The construction works by leveraging the shared secrets already derived during forward Sphinx processing. When an intermediate node determines it cannot relay a payment, whether due to insufficient channel capacity, an expired CLTV delta, or an unknown next-hop short channel ID, it generates a fixed-format failure message containing the appropriate error code. It then encrypts that message using a stream cipher keyed to the shared secret it established with the sender during the forward pass. Each preceding hop in the return path performs an additional encryption layer with its own shared secret, wrapping the error further. Only the sender holds the complete set of shared secrets and can peel every layer in sequence. The hop index at which decryption yields a valid payload tells the sender exactly where the failure occurred, without any intermediate node learning which hop failed or how deep into the route the error originated.

Integrity matters as much as confidentiality here. Each error onion layer includes an HMAC computed over the failure payload, and the sender validates these MACs as it strips each layer. This is not a convenience feature. Without tamper detection, an adversarial node could replace a legitimate TEMPORARY_CHANNEL_FAILURE report with a fabricated INCORRECT_CLTV_EXPIRY, steering the sender's pathfinding heuristic toward a false conclusion. Worse, a malicious node could inject entirely fictitious errors for payments it successfully forwarded, causing the sender to blacklist honest channels and reroute traffic through the attacker's own nodes. The MAC chain makes such modifications detectable at the sender, converting what would otherwise be a manipulable signaling channel into a verifiable diagnostic stream.

Fixed-length padding closes the remaining inference gap. If error messages varied in size depending on how many hops re-encrypted them, any observing node could estimate its distance from the failure point by measuring the error payload's length. The protocol therefore pads all error onions to a constant size, typically 256 bytes of encrypted payload regardless of origination depth. This preserves route-length indistinguishability in the reverse direction, matching the property already enforced on forward packets.

For an autonomous agent translating decoded errors into routing table updates, the design demands disciplined skepticism. A decoded TEMPORARY_CHANNEL_FAILURE at hop three should penalize the specific channel between hops three and four, reducing its estimated

available capacity in the agent's local probability model. But the agent must weight this signal against the possibility that the reporting node is adversarial. Practical implementations apply exponential decay to failure penalties and cap the downward adjustment any single error can impose on a channel's reputation score. Fee-related errors like FEE_INSUFFICIENT are more actionable because they carry verifiable fee parameters that can be cross-checked against the last known channel update. The broader principle is that failure metadata retained in local state represents a privacy surface: if the agent's routing table is ever exfiltrated, the pattern of penalized channels reveals which paths the agent has recently attempted. Scoping retention to rolling windows, typically the last few hundred attempts, and aggregating penalties at the node level rather than the channel level limits this exposure while preserving enough diagnostic resolution for intelligent retry behavior. The error onion, then, is not a debugging afterthought. It is a security-critical protocol surface where the same cryptographic discipline applied to forward routing must hold in reverse, and where every failure code an agent ingests becomes a potential vector for adversarial steering if consumed without verification.

Probabilistic Pathfinding Under Incomplete Network Knowledge: Modified Dijkstra with Success-Probability Weighting

A route that minimizes fees across three hops looks optimal until the second channel lacks sufficient liquidity and the entire payment fails. Classical shortest-path algorithms produce exactly this outcome in payment networks because they optimize against the wrong objective. The graph a sender constructs from gossip data contains advertised channel capacities, but the actual balance split within each channel is private to the two counterparties. This is not a bug or a temporary data-quality problem. It is a deliberate consequence of the privacy architecture just established: onion routing ensures that intermediate nodes learn nothing about the payment's origin or destination, and balance privacy ensures that senders learn nothing about where liquidity actually sits. The pathfinding algorithm must therefore treat every channel as a probability distribution over possible forwarding capacity rather than a deterministic link with known cost.

This shift from cost minimization to success-probability maximization restructures the entire routing computation. Each candidate hop contributes not a fee weight but a likelihood of forwarding a given

amount, and these likelihoods multiply across the path. Even channels that individually forward successfully around 90 percent of the time compound across four or five hops into end-to-end success rates that drop below 60 percent. The multiplicative penalty grows worse for larger payment amounts, where the probability of any single channel holding sufficient balance on the correct side falls sharply. Understanding how to model these per-hop distributions, compose them into path-level success metrics, and then decompose payments into smaller shards that exploit higher per-shard success rates is what separates a routing algorithm that survives a live network from one that only works when balances are known in advance.

Why Shortest-Path Is the Wrong Objective: Channel Liquidity Uncertainty and the Shift from Cost Minimization to Success-Probability Maximization

Every node in a payment network publishes two things: the total capacity of each channel and the fee it charges to forward a payment. It never publishes how that capacity is currently divided between the two endpoints. This single omission transforms routing from a well-posed optimization into a problem where the binding constraint is invisible to the solver. An agent constructing a route can observe the full graph topology, compute fees precisely, and identify the minimum-cost path with textbook efficiency. But the question that determines whether the payment actually settles is not how much the path costs. It is whether every intermediate channel holds enough balance on the correct side to forward the amount. Cost is observable. Liquidity is not. Optimizing for the former while ignoring the latter is a category error.

The divergence becomes concrete under modest assumptions. Consider a three-hop path where each channel has a capacity of roughly one million satoshis. A sender pushing 500,000 satoshis through this route needs every intermediate node to hold at least that amount on the outbound side. If balance distributions are unknown and modeled as uniform across the channel's capacity, the probability that any single hop can forward the payment is around 50 percent. Three independent hops at 50 percent each yield an aggregate success probability near 12.5 percent. The route might be the cheapest available, but it fails seven out of eight times. A longer path through channels with two million satoshis of capacity, carrying the same payment, gives each hop roughly 75 percent individual probability and compounds to something above 42 percent. Wider channels at higher fees outperform narrow channels at

lower fees because the objective that matters is not additive cost but multiplicative probability.

This distinction runs deeper than parameter tuning. Shortest-path algorithms like Dijkstra's operate on additive metrics. They sum edge weights along a path and rely on optimal substructure: if the shortest path from A to C passes through B, then the sub-paths A-to-B and B-to-C are themselves shortest. Success probability, however, multiplies. The probability of a full path succeeding is the product of each hop's individual probability, and products do not decompose the same way sums do under greedy selection. The naive application of Dijkstra with fee-based weights will systematically prefer paths that are cheap but fragile, because it is structurally blind to the multiplicative collapse that liquidity uncertainty introduces.

The resolution is elegant but confirming rather than contradictory. Pickhardt and Richter demonstrated that taking the negative logarithm of each hop's success probability converts the multiplicative objective into an additive one. Minimizing the sum of negative log-probabilities is equivalent to maximizing the product of probabilities, and this transformed metric restores compatibility with standard shortest-path machinery. Dijkstra works again, but only because the objective function has been redefined. The algorithm is the same. The question it answers is entirely different. It no longer asks which path is cheapest. It asks which path is most likely to carry the payment to completion.

For an autonomous agent executing hundreds of micropayments per hour, this reframing is not academic. An agent optimizing purely for fee minimization will encounter higher failure rates, triggering retries that consume time, state-management overhead, and HTLC slots. Each failed attempt locks liquidity temporarily and extends effective settlement latency. An agent that pays marginally higher fees for paths through well-capitalized channels settles faster, retries less, and spends less in aggregate. The pressure gradient through the network does not follow the cheapest pipes. It follows the widest ones, the channels where liquidity is deep enough to absorb the flow without resistance. Reliability-first routing is the economic optimum precisely because cost minimization, applied to a network where the real constraint is hidden, optimizes the wrong variable.

Modeling Unknown Balances as Uniform Distributions and Composing Per-Hop Success Probabilities Across Multi-Channel Paths

A routing node evaluates a candidate three-hop path. It knows each channel's publicly announced capacity. It does not know how liquidity sits within any of those channels. The payment amount is fixed. The question is precise: what is the probability that every hop along this path has sufficient local balance to forward the payment?

To answer that question requires a model for the unknown balance in each channel. The disciplined choice is the uniform distribution over the interval from zero to the channel's total capacity. This is not a convenience approximation. It is the maximum-entropy prior, the distribution that encodes complete ignorance about balance position without injecting any structural assumption about flow direction, routing history, or counterparty behavior. For a channel with capacity C carrying a payment of amount a, the probability that the forwarding node holds at least a in local balance is simply $(C - a) / C$, which reduces to $1 - a/C$. The relationship is linear in the ratio a/C. When the payment is small relative to capacity, success probability stays close to one and degrades gently. As a approaches C, the probability collapses toward zero. This single expression is the atomic unit from which all path-level reasoning is built.

Composing per-hop probabilities into a path estimate requires one structural assumption: independence. If each channel's balance is drawn from its own uniform distribution and these draws are unrelated, the probability that a payment traverses an entire path equals the product of each hop's individual success probability. Consider three channels with capacities of roughly 1,000,000, 500,000, and 2,000,000 satoshis, carrying a payment of 100,000 satoshis. The per-hop probabilities are 0.90, 0.80, and 0.95 respectively. Multiplied together, the path-level estimate is approximately 0.684. The second channel, the one with the smallest capacity relative to the payment, contributes the steepest discount. It is the bottleneck, and diagnostically it dominates the product.

The independence assumption deserves explicit acknowledgment of its limits. In practice, forwarding a payment through one channel shifts its balance, and correlated traffic patterns across the network create systematic liquidity depletion along popular routes. Joint balance distributions exist but are unobservable by any individual routing node. No sender possesses the information required to model these correlations.

The independent uniform model therefore persists as the practical baseline not because it is accurate, but because it is the most honest computation available under genuine network ignorance.

Sensitivity analysis sharpens the picture. Hold the three channel capacities fixed and vary the payment amount. At 50,000 satoshis, roughly 10% of the bottleneck channel's capacity, the path probability lands around 0.81. At 250,000 satoshis, 50% of the bottleneck, the per-hop probabilities become 0.75, 0.50, and 0.875, yielding a path estimate near 0.33. At 400,000 satoshis the bottleneck hop alone drops to 0.20, and the path probability falls below 0.14. The multiplication of probabilities close to one produces benign results. The multiplication of probabilities significantly below one produces rapid collapse. This exponential sensitivity to the a/C ratio at the weakest link is not a curiosity of the model. It is the structural pressure that forces payment splitting into existence as an operational necessity.

To make this model actionable inside a shortest-path algorithm, one final transformation is needed. Dijkstra's algorithm and its variants sum edge weights. Success probabilities multiply. The bridge between these two operations is the negative logarithm. Assigning each edge a weight of $-log(1 - a/C)$ converts the multiplicative path probability into an additive cost metric. Minimizing the sum of negative log-probabilities is identical to maximizing the product of success probabilities. This log-transform slots the uniform-distribution model directly into standard graph search with no algorithmic modification beyond weight computation. The result is a pathfinding engine that optimizes for the quantity that actually determines whether a payment settles: the likelihood that every intermediate channel can forward it.

Multipart Payments and Adaptive Splitting: Decomposing Large Transfers to Exploit Higher Per-Shard Success Rates

Roughly 7 in 10 single-path Lightning payments above 100,000 satoshis fail on the first attempt, according to estimates drawn from probing studies of mainnet liquidity distributions. The cause is structural: each hop compounds a probability penalty that grows steeper as the transfer amount approaches channel capacity. Decomposing a large transfer into smaller, independently routed shards reverses this dynamic. Because per-hop success probability is a concave function of amount, splitting yields a superlinear improvement in per-shard delivery odds. The composite success rate for the full payment can exceed what any single path achieves alone. This guide walks through the reasoning, the math, and

the operational design choices required to implement adaptive multipart payments that convert probability theory into measurable delivery gains.

Step 1: Establish the Mathematical Basis for Splitting Gains

Under the uniform balance model, the probability that a channel with capacity C can forward an amount a is approximately $(C - a) / C$. This linear relationship between amount and failure risk creates a concave success-probability curve. Halving the shard amount does not merely halve the failure risk. It produces a disproportionately larger improvement in per-shard success probability, and when independent shards are composed multiplicatively, the aggregate rate jumps nonlinearly. To internalize this, work through the algebra for a single hop. If $C = 150{,}000$ sats and $a = 100{,}000$ sats, the success probability is $(150{,}000 - 100{,}000) / 150{,}000 \approx 0.33$. Now split a into four shards of 25,000 sats. Each shard's per-hop probability becomes $(150{,}000 - 25{,}000) / 150{,}000 \approx 0.83$. The probability that all four shards succeed across independent paths of similar capacity is $0.83^4 \approx 0.48$. A single path delivered 33% expected success. Four independent shards deliver roughly 48%, and that gap widens further once retry logic enters the picture.

1. Compute single-path success probability: $(C - a) / C$ for the full payment amount across the weakest channel on the path.
2. Compute per-shard probability for n equal shards: $(C - a/n) / C$, raised to the power of path length for each shard.
3. Multiply independent shard probabilities to get aggregate success rate. Compare against the single-path baseline.
4. Note that retry logic on individually failed shards avoids restarting the entire payment, compounding the effective success rate beyond the all-succeed calculation.

Step 2: Select a Shard Sizing and Count Strategy

Equal splitting is the simplest approach, but it leaves value on the table. When your pathfinding engine has partial knowledge of channel capacities along candidate routes, you can size shards to match the capacity profiles of available paths. A path bottlenecked at 40,000 sats should carry a shard no larger than around 30,000 sats to preserve a high per-hop probability margin. Unequal splits tuned to known constraints outperform naive division. The choice of shard count introduces a direct tradeoff. More shards raise aggregate success probability but increase the total number of HTLCs in flight, consume more routing fee budget, and extend coordination latency. In practice, implementations typically cap shard count between 4 and 16, guided by the diminishing marginal gain in success probability beyond a certain split granularity. The optimal count depends on payment size relative to median channel capacity in the local network graph.

1. Query the local channel graph to identify candidate paths and their bottleneck capacities.
2. Size each shard to sit well below the bottleneck capacity of its assigned path, targeting per-hop success probabilities above 0.75.
3. Cap total shard count by evaluating the marginal success-probability gain of adding one more shard against the additional HTLC and fee cost.
4. For payments where channel capacity data is sparse, default to equal splits with a conservative count of 4–8 shards.

Step 3: Choose Between Static and Dynamic Splitting

Static splitting commits to a shard decomposition before any shard is dispatched. The sender divides the payment, assigns paths, and launches all shards in parallel. This minimizes latency but cannot adapt to information revealed by early failures. Dynamic splitting, by contrast, sends an initial subset of shards and adjusts the remaining decomposition based on which shards succeed or fail. A failed shard's amount can be re-split and rerouted along alternative paths, incorporating the negative signal about the failed channel's liquidity. Dynamic splitting trades higher latency for better informed routing decisions. Each failed shard narrows the probability distribution on the failed channel's balance, allowing subsequent attempts to avoid that channel or reduce the amount routed through it. For autonomous agents operating under tight latency budgets, the choice is a calibration problem: how much additional delay is acceptable for a given improvement in completion rate. Production implementations often use a hybrid approach, launching an initial static split and switching to dynamic re-splitting only when partial failures occur.

1. Implement a static split as the default fast path: decompose, assign routes, dispatch all shards concurrently.
2. On partial failure, capture the failed shard's path and amount. Update local channel-probability estimates to reflect the failure signal.
3. Re-decompose the failed shard's amount into smaller sub-shards and route them on alternative paths that exclude the failed channel.
4. Set a maximum retry depth per shard to bound total settlement latency and prevent indefinite HTLC lockup.

Step 4: Manage HTLC Coordination and Partial-Payment Failure Modes

Each shard locks an HTLC along its entire path until the recipient releases the preimage. If some shards arrive and others stall, the recipient holds partial liquidity hostage until stalled HTLCs expire. This partial-lockup risk scales with shard count and path length. Aggressive splitting amplifies exposure: 16 shards across 4-hop paths create 64 locked HTLC slots across the network, each consuming channel capacity that other payments cannot use. Mitigating this requires tight timelock coordination. The sender must ensure that all shards share the same payment hash so the recipient can settle them atomically, but the CLTV expiry deltas across shards should be calibrated so that the fastest path's HTLC does not expire before the slowest path's HTLC arrives. Cumulative routing fees also deserve attention. Each shard independently accrues per-hop fees, and the total fee burden for a highly split payment can exceed the single-path fee by a factor of 3–5x. The sender's fee budget must account for this multiplicative cost.

1. Ensure all shards reference the same payment hash and total amount so the recipient can verify completeness before releasing the preimage.
2. Set CLTV expiry deltas per shard to accommodate the longest expected path, adding a safety margin of 10–20 blocks for the slowest shard.
3. Track cumulative fee expenditure across all shards and abort remaining sends if the fee budget ceiling is reached.
4. Implement a timeout at the sender: if not all shards settle within a configurable window, cancel outstanding HTLCs proactively rather than waiting for on-chain expiry.

Step 5: Synthesize Probabilistic Routing with Portfolio-Style Shard Orchestration

The full power of multipart payments emerges when you combine per-hop probability modeling with shard decomposition as a unified routing strategy. The problem transforms from finding one good path to orchestrating a portfolio of partial paths. Each shard is an independent bet on a different liquidity corridor, and spreading exposure across uncorrelated channels reduces variance in the same way financial diversification reduces portfolio risk. In implementation, the pathfinding engine should score candidate shard-path pairs by their joint success probability, fee cost, and HTLC overhead, then solve for the decomposition that maximizes expected completion rate subject to a fee budget constraint. This is a constrained optimization problem, not a greedy search. The solver must weigh the diminishing returns of additional splitting against the rising coordination cost, producing a shard plan that balances probability, latency, and fee expenditure. Agents operating under programmable micro-budgets can encode these constraints as policy parameters, enabling fully autonomous payment orchestration without human intervention.

1. For each candidate path, compute the joint success probability across all hops using the uniform balance model.
2. Score shard-path pairs by the product of success probability and inverse fee cost, creating a quality metric per candidate.
3. Use a constrained optimizer or greedy approximation to select the shard count and path assignments that maximize aggregate success probability within the fee budget.
4. Log the outcome of each shard for post-hoc analysis, feeding settlement data back into the probability model to improve future routing decisions.

You now have the machinery to reason about when, why, and how to decompose large payments into independently routed shards. The core insight is structural: concavity in the success-probability function means that splitting is not merely a workaround for insufficient liquidity but a mathematically grounded strategy that raises expected completion rates beyond what any single path can deliver. By sizing shards to match channel capacities, choosing between static and dynamic splitting based on latency constraints, and managing HTLC coordination to contain

partial-lockup risk, you can engineer payment orchestration that operates reliably under adversarial liquidity conditions. This portfolio approach to routing sets the stage for the fee dynamics and channel rebalancing protocols that determine whether these liquidity corridors remain available over time.

Fee Market Dynamics for Routing Nodes and Channel Liquidity Rebalancing Protocols

A routing node can discover every viable path through the network, encrypt each hop with flawless onion layering, and still bleed sats on every payment it forwards. The paradox is structural: successful routing depletes the very resource that makes routing possible. Each payment pushed through a channel shifts the local balance toward one side, and once a channel is fully drained in the dominant flow direction, it becomes useless for forwarding until capacity is restored. The node that captures the most volume faces the fastest liquidity drain, and if its fee schedule treats directional capacity as a free input, profitability inverts. Revenue from forwarding gets consumed by the cost of restoring what forwarding destroyed.

This is where pathfinding hands off to economics. The algorithms explored previously select routes by estimating success probabilities and cumulative fees, but those inputs are themselves outputs of decisions that routing nodes make in real time about how to price their channels and when to trigger rebalancing operations. A fee set too low attracts volume that exhausts capacity before costs are recovered. A fee set too high pushes traffic to competitors and leaves capital idle. And the optimal policy is not static. It shifts with every forwarded payment, every change in competing node fees, every on-chain confirmation that alters the cost of splicing or swapping liquidity back into position. For an autonomous agent operating a routing node under sustained load, this collapses into a continuous optimization problem coupling fee adjustment, capacity allocation, and rebalancing triggers against a moving baseline of network demand.

Base Fee Plus Proportional Rate: How Routing Nodes Price Forwarding and Why Fee Competition Follows Power-Law Topology

What determines the price a routing node charges to forward a payment it neither originates nor receives?

The answer is not a single number. Every Lightning routing node advertises two parameters in its channel update messages: a base_fee_msat, denominated in millisatoshis and charged per forwarding event regardless of payment size, and a fee_rate_millionths, a proportional rate applied to the forwarded amount. The total fee for forwarding a payment of value v through a channel is base_fee + (fee_rate × v) / 1,000,000. This two-component structure is not arbitrary. The base fee compensates for resources consumed by the forwarding act itself: each HTLC occupies one of a finite number of commitment transaction output slots, requires signature recomputation on both sides of the channel, and imposes a non-trivial on-chain footprint if the channel must be force-closed while the HTLC is in flight. These costs are constant with respect to payment size. The proportional rate prices something different entirely. It captures the opportunity cost of liquidity locked during the HTLC's lifetime. A node forwarding 500,000 satoshis ties up capital that cannot simultaneously serve other payment flows, and the duration of that lock depends on the downstream path's timelock delta. The proportional component makes this cost scale with the amount at risk.

Together, these two parameters define a pricing surface that node operators tune against observable channel conditions. A node whose outbound liquidity in a given channel is depleting toward zero faces increasing rebalancing costs, so it raises the proportional rate on that channel to slow depletion or compensate for the circular rebalancing expense it will eventually bear. A node with ample bilateral liquidity and high connectivity can afford low fees to attract volume. This is precisely the strategy visible among well-known hub nodes on the production Lightning Network. Operators like ACINQ and River Financial historically set base fees near zero or at one millisatoshi while maintaining moderate proportional rates, around 1 to 50 parts per million. Their position in the topology, with hundreds or thousands of channels and high betweenness centrality, makes volume-based revenue viable. By contrast, a peripheral node with three channels and sporadic forwarding traffic faces a different cost structure. Each forwarding event consumes a meaningful fraction of its operational attention, so higher base fees reflect the genuine per-event marginal cost. Fee strategy and topology position are not independent variables. They are coupled.

That coupling produces a structural outcome. Empirical analyses of the Lightning Network's forwarding graph, notably work by Zabka et al.

in 2022 and Tikhomirov et al. in 2020, consistently find that betweenness centrality and channel degree follow power-law distributions. Roughly 1 to 2 percent of nodes sit on a disproportionate share of shortest or highest-probability paths, forwarding an estimated 80 percent or more of payment volume. This concentration arises through preferential attachment: a node with many channels appears in more candidate routes, which attracts more channel-opening requests, which further increases its routing share. Fee competition reinforces the dynamic. Hub nodes can undercut peripheral competitors on per-satoshi cost because they amortize fixed infrastructure across massive throughput. Peripheral nodes cannot match those rates without operating at a loss.

This pattern creates a tension worth naming directly. The argument developed throughout this chapter holds that trust-free verification at every hop, enforced by the hash-lock mechanism examined in earlier sections on HTLC dispute resolution and onion-routed privacy, eliminates the need to trust intermediaries with custody. That property remains intact regardless of how concentrated forwarding revenue becomes. But cryptographic decentralization and economic decentralization are distinct properties. A network where every relay verifiably cannot steal funds may still concentrate forwarding income among a small oligopoly of hub operators. The fee market does not violate trust-free guarantees. It simply reveals that incentive-compatible pricing, under preferential-attachment topology, converges toward hub dominance as a stable equilibrium.

For an autonomous agent performing probabilistic pathfinding, the operational consequence is precise. Fee schedules must be modeled as functions of a node's topology position and estimated liquidity state, not treated as static cost labels scraped from gossip messages. A channel advertising a conspicuously low proportional rate may signal abundant outbound liquidity, or it may signal a hub pursuing volume at the expense of channel balance sustainability. An agent that cannot distinguish these cases will select paths that look cheap but carry elevated failure probability due to depleted channels behind the low-fee advertisement. The fee surface, in other words, is a signal to be decoded, and the decoder must account for the power-law structure shaping every route it evaluates.

Circular Rebalancing, Submarine Swaps, and Splicing: Operational Techniques for Restoring Bidirectional Channel Capacity

A routing node operator watches a high-volume channel drain toward zero local balance on one side. Every satoshi that flowed through in a single direction moved the channel closer to a state where it can no longer forward payments along that vector. The channel still exists in the routing graph, still advertises capacity, but it has become functionally unidirectional. Restoring bidirectional flow is not a maintenance task. It is a capital allocation decision with three distinct instruments available, each occupying a different position on the surface defined by cost, latency, and on-chain disruption.

Circular rebalancing operates entirely within the Lightning Network. The operator constructs a payment from themselves to themselves, routing it through a path that begins at a channel with surplus local balance and terminates at the depleted channel, effectively shifting sats from where they are abundant to where they are needed. The cost is the sum of routing fees charged by every intermediate node along the circular path. This makes the break-even calculation straightforward in principle: the cumulative fee paid for the rebalance must be less than the forwarding revenue the restored capacity is expected to generate before the channel depletes again. In practice, the operator faces uncertainty about future flow volume and direction, which means the rebalance is a bet on projected demand. Circular rebalancing settles in seconds. It requires no on-chain transaction. It leaves the channel's total capacity unchanged. But it depends entirely on the availability of a viable circular route with sufficient liquidity at every hop, and during periods of widespread directional imbalance, such routes become expensive or unavailable.

When the problem is not merely balance distribution but a net surplus or deficit of Lightning-side liquidity, submarine swaps bridge the boundary between off-chain and on-chain funds. A loop-out swap lets the operator send Lightning sats to a swap provider and receive on-chain bitcoin in return, draining a channel intentionally to recover on-chain capital. A loop-in does the reverse, converting on-chain funds into inbound channel capacity. Both directions use hash-locked contracts that enforce atomicity: either both legs settle or neither does, eliminating counterparty risk without requiring trust in the swap provider beyond liveness. The trade-off is concrete. On-chain fees and confirmation

latency, typically on the order of tens of minutes for reasonable fee targets, replace the near-instant settlement of circular rebalancing. Submarine swaps are the correct instrument when the node's aggregate on-chain and off-chain liquidity allocation is misaligned, not just the internal balance of a single channel.

Splicing changes the question entirely. Where circular rebalancing redistributes existing capacity and submarine swaps convert between liquidity domains, a splice-in or splice-out mutates the channel itself. A splice-in adds on-chain funds to an existing channel, increasing its total capacity without closing and reopening it. The channel retains its position in the routing graph, preserves any in-flight HTLCs, and avoids the gossip disruption of a close-and-reopen cycle. A splice-out withdraws funds from the channel to an on-chain output, shrinking capacity when the operator determines that collateral is better deployed elsewhere. Both operations require an on-chain transaction and its associated fee and confirmation delay, but they avoid the double on-chain cost of a close followed by a new channel open.

The selection framework resolves cleanly once the operator identifies which variable is constrained. If total channel capacity is correct and only balance distribution has drifted, circular rebalancing is the lowest-cost, lowest-latency option, provided routing fees remain below projected revenue recovery. If the node needs to move liquidity across the on-chain/off-chain boundary while keeping channels intact, submarine swaps are the appropriate atomic bridge. If the channel itself is undersized or oversized for the traffic it serves, splicing is the only technique that adjusts the structural parameter. These are not interchangeable tools. They compose into an operational sequence: an operator might splice-in to resize a channel for a new traffic pattern, use circular rebalancing to keep balance centered under normal flow, and execute periodic loop-outs to prevent on-chain reserves from falling below safety thresholds. The discipline lies in matching each intervention to the specific constraint it resolves, treating every rebalancing action as a priced decision within the node's liquidity budget rather than a reflexive response to imbalance alerts.

An Agent Routing Node Under Load: Fee Adjustment, Liquidity Allocation, and Revenue Optimization Across Competing Payment Flows

Roughly seven out of ten forwarding failures on a mature routing node trace not to insufficient aggregate capacity but to directional depletion

on a single channel pair. Rina Subramanian had cataloged this pattern across three months of production telemetry from the compute marketplace's internal routing infrastructure before she fully believed it. The node in question held around 2.8 BTC in total channel capacity distributed across fourteen channels, and its aggregate utilization rarely exceeded 60 percent. Yet specific channel pairs failed repeatedly, rejecting HTLCs that the pathfinding layer had scored as high-probability. The node looked healthy in aggregate. It was bleeding revenue at the channel level.

Rina traced the pathology to a uniform fee policy. The node's operator had configured identical base and proportional rates across all channels, a reasonable default that became structurally destructive under asymmetric load. Three channels connecting to high-demand inference providers absorbed the majority of outbound flow, while inbound payments arrived distributed across the remaining eleven. The result was predictable once she mapped the directional flow ratios: the three provider-facing channels drained their outbound capacity within hours, while their inbound side accumulated satoshis that could not be forwarded without a corresponding outbound path. The corrective intervention was per-channel fee differentiation keyed to directional imbalance. Channels whose outbound capacity fell below 30 percent of total received a proportional fee increase that throttled demand and simultaneously accumulated revenue to fund rebalancing. Channels with surplus outbound capacity received fee reductions to attract flow. This was not a one-time adjustment. It was a feedback loop: monitor depletion velocity, adjust fees, observe the response in forwarding success rate, iterate.

The rebalancing question compounded the problem. The node's operator had been rebalancing reactively, initiating circular payments only after a channel was fully depleted in one direction. Rina's analysis showed that the cost of restoring capacity from zero was consistently higher than restoring it from a 20 percent threshold. The reason was structural. When a channel is fully depleted, the rebalancing path must traverse longer routes with higher aggregate fees because the depleted channel itself cannot participate in shorter circular paths. She modeled the break-even condition explicitly: the fee paid along the rebalancing path must remain below the marginal forwarding revenue that the restored capacity would generate before the next depletion event. For channels with high arrival rates of small payments, the threshold was forgiving. For channels that occasionally forwarded large HTLCs with long inter-arrival times, the margin was razor-thin. The optimal rebalan-

cing trigger depended on the observed payment-size distribution, not on a fixed percentage.

Competing payment flows introduced a third variable that neither fee policy nor rebalancing strategy could resolve independently. The same provider-facing channels forwarded both streams of small inference-payment HTLCs and occasional large batch-settlement transfers. Reserving capacity for a single large HTLC reduced throughput on the small-payment stream, whose cumulative fees often exceeded the single large payment's forwarding revenue. Rina proposed channel specialization where topology permitted it, opening dedicated channels for large-value flows and routing small payments through the existing infrastructure. Where opening new channels was uneconomical, she implemented tiered fee schedules that priced large HTLCs at a premium reflecting their opportunity cost on capacity. An HTLC consuming more than 40 percent of a channel's remaining directional capacity paid a proportional rate roughly three times the baseline, a price that either deterred the flow to a better-capitalized path or compensated for the throughput loss.

The deeper lesson crystallized in Rina's mechanism-failure catalog as entry forty-seven: fee policy, rebalancing triggers, and capacity allocation form a coupled system. Adjusting fees without modeling rebalancing cost produces a node that prices forwarding below its true operational expense. Rebalancing without adjusting fees to recoup the cost creates a structural deficit that compounds under sustained load. Allocating capacity without differentiating payment flows starves high-margin channels to serve low-margin traffic. She annotated the entry with a control-theoretic framing: the routing node is a system with three interdependent state variables, and stability requires that changes to any one propagate corrective signals to the other two. This was not configuration. It was continuous optimization, and the agents that would soon procure GPU inference through this node's channels would depend on exactly this discipline to find reliable, correctly priced paths from discovery through settlement.

The privacy guarantees of onion routing, the probabilistic models that navigate incomplete liquidity data, and the fee markets that fund and reshape channel topology are not three separate engineering concerns layered onto a payment graph. They constitute a single optimization surface. Opacity at each hop means no intermediate node can reconstruct the full payment arc, which simultaneously protects sender anonymity and destroys the possibility of globally consistent routing

tables. Probabilistic pathfinding compensates for exactly this information deficit, treating unknown channel balances as distributions and computing success likelihoods rather than deterministic shortest paths. Fee schedules then act as the pricing signal that determines which channels carry sufficient liquidity to remain viable, which close, and which new ones open, continuously deforming the very topology that pathfinding algorithms attempt to navigate. The routing layer, understood correctly, is the first place in the autonomous commerce stack where cryptographic constraints and economic incentives become genuinely insoluble from each other. Static algorithm selection cannot track a landscape that reprices and restructures between payments.

That recognition reframes what comes next. Once an agent can move value through a trust-minimized network under privacy, uncertainty, and dynamic pricing constraints, the harder upstream question surfaces: how does that agent discover, evaluate, and select a counterparty before any channel or route exists? Routing presupposes endpoints.

Chapter Five

Distributed Service Discovery for Agent Compute Procurement

According to a widely cited 2023 estimate from the Internet Engineering Task Force, roughly 40 percent of all DNS queries already fail to resolve on first attempt due to cache misses, timeout thresholds, and stale records. For a human browsing the web, that failure is invisible because a retry fires within milliseconds and the browser handles it silently. For an autonomous agent holding an open payment channel with funds locked and an HTLC ticking toward expiry, a failed lookup is not a UX hiccup. It is a procurement abort.

The previous chapters assembled a stack that settles payments in sub-millisecond state updates, authenticates service access through L402 credentials, and routes value across multi-hop channels. But that entire stack idles the moment an agent lacks a single piece of information: which provider, at what address, offers the specific compute capability it needs, at a price it can accept, within a latency bound it can tolerate. The payment rails are live and funded. The agent has nowhere to send the first satoshi. This is not a missing feature. It is a protocol-layer absence, and it reduces every prior primitive to dead infrastructure.

Closing that gap requires a discovery substrate whose first design constraint is decidedly unglamorous but structurally non-negotiable: a

registration format that makes provider capabilities, pricing commitments, and performance SLAs machine-queryable rather than human-browsable, which forces precise decisions about what gets published into a distributed hash table and how capability descriptors bind to verifiable pricing metadata.

DHT-Based Resource Registration: Capability Descriptors, Pricing Metadata, and Latency SLA Publication

Roughly seven out of ten service-discovery failures in distributed agent benchmarks trace not to network partitions or payment faults, but to stale or unreachable registry entries. The funded channel is open, the L402 credential is valid, the routing path is computed, and yet the agent idles because the lookup layer returned a provider that vanished forty minutes ago. Every trust-minimization guarantee built into the settlement stack evaporates the moment discovery depends on a single coordinating server that can go offline, filter results, or serve records no one has verified since insertion. The problem is not hypothetical. It is the exact failure mode that payment channels and credentialized access were engineered to prevent at the transaction layer, now reintroduced one hop earlier in the procurement sequence.

A distributed hash table can eliminate the centralized chokepoint, but only if the records it stores carry enough structure to support autonomous procurement decisions and enough cryptographic binding to resist adversarial pollution. An agent querying for GPU inference capacity needs more than an IP address. It needs a machine-parseable descriptor that encodes compute type, current pricing curves, and latency bounds, signed by a key the provider controls, timestamped with a freshness attestation that distinguishes a live cluster from a phantom record planted during a churn event. Designing that descriptor format and the eviction policy that governs its lifecycle is not a metadata exercise. It is protocol surface, and the integrity of every downstream micropayment depends on getting it right.

Why Centralized Registries Fail: The Liveness and Censorship Constraints of Agent-Scale Discovery

Roughly seven in ten production API gateways rely on a single DNS resolution step before any client can reach a service catalog. That statistic, drawn from cloud-native deployment surveys, understates the fragility it represents. When the entity performing discovery is a human de-

veloper, a five-second retry is an inconvenience. When the entity is an autonomous agent executing a continuous procurement loop with sub-second latency targets, that same five seconds is a broken contract, a missed settlement window, and a cascading failure across every downstream service the agent had committed to provision.

The chapters preceding this one established how value moves through payment channels, how HTLCs enforce atomicity across multi-hop paths, and how L402 credentials bind authorization to payment at the moment of service delivery. Pathfinding algorithms weight routes by success probability and fee economics. But every one of those mechanisms presupposes that the agent already knows where to send the payment. The routing layer optimizes delivery to a known destination. It says nothing about how that destination enters the agent's awareness in the first place. Discovery is the precondition that the entire settlement stack silently assumes, and if that precondition depends on a centralized registry, the trust-minimization guarantees built into channels, HTLCs, and credentialed access unravel at the front door.

A centralized registry introduces three distinct failure surfaces. The first is liveness. A registry that goes offline, whether through infrastructure failure, maintenance, or denial-of-service attack, blocks every agent from locating any provider. This is not a graceful degradation scenario. An agent with an open channel and sufficient liquidity cannot spend if it cannot discover what to buy. The payment layer was deliberately designed to eliminate single points of failure through multi-hop routing and bilateral state management. Inserting a centralized lookup service upstream reintroduces exactly the chokepoint that channel architecture was engineered to remove. The second failure surface is censorship. A registry operator who controls listing visibility can silently exclude providers, reshaping the competitive landscape an agent perceives without any detectable signal. This is not a political objection. It is an adversarial-model concern. If an agent's price-discovery process operates over a filtered set of providers, the auction and congestion-pricing mechanisms that later chapters depend on produce distorted outcomes. Truthful bidding in a second-price auction means nothing if the auctioneer has already removed half the bidders from the room. The third failure surface is throughput. A catalog designed for human browsing patterns handles perhaps tens of thousands of queries per minute. Scale that to millions of autonomous agents issuing discovery queries every few hundred milliseconds and the arithmetic collapses. The registry must either rate-limit, which degrades procurement latency below SLA thresholds,

or provision infrastructure at a scale that recreates the cost and trust concentration the entire system was designed to avoid.

Consider a concrete contrast. DNS-based service discovery resolves a hostname to an IP address through a hierarchical cache. Under human-driven query loads, cached TTLs absorb most of the pressure. But an agent that needs to re-resolve provider endpoints every 200 milliseconds to track capability changes will overwhelm TTL windows and hammer authoritative nameservers. A centralized API marketplace fares worse. Its search index, ranking algorithm, and access control logic all run on infrastructure owned by a single operator. At an estimated three orders of magnitude increase in query volume over human-paced usage, response latencies climb from tens of milliseconds toward seconds, and the operator faces an economic choice between raising prices and degrading service. Either outcome violates the bounded-latency requirement of an autonomous procurement loop.

There is a pattern older than any protocol that illuminates why distributed discovery is structurally necessary. In oral knowledge traditions, critical survival information was never held by a single elder or stored in a single location. It was distributed redundantly across community members so that the loss of any individual did not erase the community's capacity to navigate, heal, or build. The resilience property was not philosophical preference but operational necessity under adversarial conditions: drought, conflict, displacement. DHT-based discovery formalizes this identical pattern with cryptographic rigor. Records are distributed across a key space, retrievable from any participating node, and verifiable without trusting the node that serves them. The mechanism differs. The structural logic is the same.

What remains unaddressed is what happens after discovery surfaces multiple providers competing for the same agent's budget. When a single agent negotiates bilaterally, pricing is straightforward. When many agents query the same capability descriptors simultaneously, the bilateral model breaks. Prices must adjust dynamically, and agents must have reason to bid honestly rather than strategically. That is a different class of problem entirely.

Structured Capability Descriptors: Encoding Compute Type, Pricing Curves, and Latency Bounds into DHT Key-Value Records

In the spring of 2023, an engineer at a mid-tier cloud broker spent three weeks debugging a procurement failure that should never have occurred.

An autonomous scheduling agent had discovered a GPU provider through a distributed hash table lookup, opened a payment channel, streamed four minutes of micropayments for inference workload, and then discovered that the provider's hardware was two generations behind what the task required. The DHT record had listed "GPU compute" as a flat string tag. No model family, no VRAM tier, no pricing structure beyond a single rate. The agent had no way to know until after settlement began that the resource was unfit. The wasted channel funding, the latency penalty, the reputational cost to the broker: all of it traced back to a descriptor that carried almost no structured information.

That failure illustrates the core design requirement. A capability descriptor stored in a DHT is not a metadata label. It is a multi-dimensional record that an autonomous agent must evaluate combinatorially before initiating any channel negotiation or spending a single satoshi. The record must encode compute class as typed fields: GPU model family, VRAM tier in gigabytes, supported instruction sets, and whether the provider offers batch or streaming execution modes. It must encode pricing not as a scalar but as a piecewise function. A base cost per compute-second at low utilization, a congestion multiplier that activates above a published capacity threshold, and an optional auction-mode flag indicating that scarce slots are allocated by competitive bid rather than posted price. Each of these fields must be machine-parseable without prior schema negotiation. This means the DHT value either conforms to a pre-shared schema identifier embedded in the key prefix or is serialized in a self-describing format like deterministic CBOR with canonical field ordering. Canonical serialization is not a convenience. It enables content-addressed keys and signature verification over the descriptor body, binding the provider's cryptographic identity to the exact capabilities they claim.

Latency SLA publication demands comparable rigor. A single millisecond figure is meaningless without specifying measurement methodology: does the number represent client-measured round-trip time, or provider-attested processing latency exclusive of network transit? The descriptor must declare geographic scope, distinguishing a regional guarantee valid within a specific data sovereignty zone from a global claim. It must specify penalty conditions. When the SLA is violated, does the payment channel automatically reduce the billing rate? Does a refund output activate? Does the contract terminate with a final state published on-chain? Without these typed fields, an agent cannot distinguish a

genuine low-latency provider from one publishing optimistic numbers backed by no enforcement mechanism.

Consider the procurement constraints faced by autonomous supply-chain agents performing real-time route optimization for military logistics under degraded and adversarial network conditions. The descriptor schema for edge compute in this scenario must encode availability guarantees as uptime percentages under specified network partition scenarios, geographic constraints that enforce data sovereignty boundaries, and a freshness field recording the maximum permissible age of the DHT record itself. An agent filtering across availability, geography, latency bounds, and budget feasibility simultaneously cannot do so against flat key-value pairs. It requires typed, structured fields that support combinatorial evaluation locally, on the agent's own hardware, before any outbound network contact with a provider. This is the difference between sub-second procurement and a multi-roundtrip negotiation that collapses under adversarial latency.

The encoding format must balance compactness against expressiveness. DHT implementations commonly impose value size limits in the range of roughly one to ten kilobytes. A well-designed descriptor using deterministic CBOR or protobuf with fixed field ordering fits comfortably within these bounds while remaining signable and content-addressable. The provider signs the serialized descriptor body with its long-term identity key. Any agent retrieving the record from a DHT node can verify that signature without trusting the node that served it. This property is not optional. In a network where DHT nodes may be operated by adversaries or competitors, unsigned descriptors are indistinguishable from fabricated ones. The signature binds identity to capability claim, and the canonical serialization ensures that no intermediary can alter a single field without invalidating the binding. What remains, once the descriptor is verified and its fields evaluated, is the question of whether that record is still fresh, whether the provider's claims reflect current capacity rather than yesterday's. That problem of cryptographic freshness demands its own machinery.

Stale Record Eviction and Cryptographic Freshness Attestation Under Adversarial Provider Churn

What happens when an agent, operating under a tight micro-budget, opens a payment channel to a provider that stopped serving compute forty minutes ago?

The answer is concrete and expensive. The agent burns a full round-trip timeout, often in the range of five to fifteen seconds depending on network conditions, waiting for a handshake that never completes. If the agent's procurement logic had already committed a channel-opening transaction or reserved collateral against the discovered record, that capital sits locked until the timeout expires and fallback routing kicks in. Multiply this across a discovery layer serving thousands of agents, each contacting a registry polluted with records from providers that vanished without notice, and the cost shifts from inconvenience to systemic drag. Stale records are not merely housekeeping debt. They are an active attack surface. An adversary who plants records for nonexistent endpoints forces agents into timeout cycles, draining micro-budgets through attrition. A passive failure, where a provider simply crashes and never publishes a departure message, produces the same downstream effect. The threat model must treat both cases identically.

The countermeasure is a cryptographic freshness attestation that every provider must periodically publish alongside its DHT record. The attestation takes the form of a signed liveness beacon containing three fields: a monotonically increasing epoch counter, a recent anchor hash drawn from a public blockchain or timestamping service, and the hash of the provider's current capability descriptor. An agent verifying a discovered record checks that the beacon's epoch falls within an acceptable window, typically the current epoch or one epoch prior. It then confirms that the anchor hash corresponds to a block produced within the last few minutes, precluding replay of an old beacon against a new query. Finally, it verifies the descriptor hash matches the record it just retrieved, ensuring the beacon attests to the capabilities actually advertised. Only after all three checks pass does the agent proceed to channel negotiation. This sequence adds minimal latency, roughly one signature verification and one hash comparison, while eliminating the dominant failure mode of contacting phantom providers.

TTL-gated eviction enforces this at the protocol layer rather than leaving it to individual agent policy. Each record published to the DHT carries a hard TTL, set at publication time and typically calibrated to two or three beacon epochs. Neighboring DHT nodes refuse to serve any record whose most recent freshness beacon has an expired TTL. This is not a suggestion. It is a protocol-level invariant. Soft-TTL schemes, where stale records linger until a cleanup sweep runs, fail under adversarial churn because there is no guarantee that cleanup ever executes for a given keyspace partition. Hard TTL makes eviction auto-

matic and unilateral: if the beacon is stale, the record is unresolvable, period.

The deeper adversarial edge remains. An attacker who compromises a provider's signing key can continue publishing valid beacons for a service that no longer exists, passing all three verification checks while routing agents into a black hole. Binding the freshness beacon to an on-chain economic commitment closes this gap. If the beacon must reference a verifiable funding outpoint or collateral deposit, then sustaining the deception requires the attacker to keep real capital locked. The cost of maintaining a fraudulent beacon becomes a continuous expenditure rather than a one-time key theft, and rational adversaries abandon the attack once the collateral cost exceeds the extractable value.

This pattern generalizes beyond compute procurement. Consider any registry where published capabilities go stale as the underlying entity changes. Talent matching platforms for gifted children face a structurally identical problem: a profile advertising olympiad-level mathematics becomes misleading after two years of growth and shifting interests, and matching systems that fail to evict outdated profiles produce mismatched placements. The freshness attestation and hard-TTL eviction logic is not a compute-specific invention. It is a general registry integrity primitive.

What matters for the agent compute context is that freshness is ultimately an incentive problem, not merely a cryptographic one. Signatures prove that a key holder attested to liveness at a given moment. Economic anchors prove that the attestation carries a cost to maintain dishonestly. Together, they give the discovering agent a basis for trust that is neither centralized nor naïve, making the transition from discovery to channel funding a decision grounded in verifiable, current, and economically backed evidence.

DNS-SD Integration and Decentralized Merkle-Committed Registries for Censorship-Resistant Provider Listings

Roughly eight in ten local service discovery interactions on the internet today resolve through DNS-SD, a protocol suite designed for trusted networks where the registrar and the querier share an implicit assumption of honest listing. That assumption holds on a home LAN. It disintegrates the moment an autonomous agent tries to discover compute providers across adversarial boundaries, because a single DNS operator can silently suppress a provider's SRV or TXT record and no client pos-

sesses the evidence to distinguish a missing listing from a nonexistent one. The gap is not hypothetical. It is the operational surface where censorship becomes invisible.

Content-addressed lookup through a distributed hash table, as established in prior sections, solves the problem of finding a capability descriptor once you know what to ask for. But it delegates the question of whether the registry itself is trustworthy to someone else's architecture. Grafting Merkle-committed, append-only data structures underneath DNS-SD's familiar lookup semantics creates a two-layer discovery substrate: fast, human-readable queries on top, and cryptographic inclusion proofs below that let any agent verify a listing was present at a specific registry state. The engineering consequence is a direct collision between censorship resistance and spam resistance, since an open append-only log that accepts entries without cost becomes a magnet for Sybil identities flooding the listing space. Resolving that tension requires stake-weighted insertion mechanisms that make listing economically meaningful without reintroducing a gatekeeper, and fallback hierarchies that let agents navigate the trust gap between a fast centralized DNS answer and a slower but verifiable proof of listing integrity.

Bridging DNS-SD Service Records to Decentralized Discovery: TXT Record Semantics and Lookup Fallback Hierarchies

Roughly seven in ten DNS resolvers worldwide still do not validate DNSSEC signatures, according to APNIC measurement data from 2024. That single statistic reframes the entire question of whether DNS-SD belongs in a discovery pipeline designed for adversarial machine commerce. The answer is not binary. DNS-SD delivers sub-second lookups from cached infrastructure that blankets every network on Earth, and that speed advantage is real. But speed without verification is just a hint, and hints require a protocol structure that knows what to do when hints lie.

DNS-SD TXT records were designed to advertise printers and media servers on trusted local networks. Nothing in the RFC prevents them from carrying richer semantics. A TXT record for a compute provider in an agent procurement context encodes a small set of key-value pairs: a capability descriptor specifying accelerator class and memory tier, a pricing commitment denominating rate per unit in a specific currency with channel-type requirements, and a cryptographic anchor consisting of a hash pointer to the provider's current signed entry in a Merkle-committed registry. The anchor is what transforms the TXT record from a trus-

ted assertion into a verifiable pointer. An agent receiving `gpu=a100, mem=80g, rate=50sat/s, anchor=sha256:e3b0c44...` treats the capability and pricing fields as advisory. The anchor field is the only one that matters for downstream verification, because it lets the agent fetch and validate the provider's full signed listing against the registry root without trusting the resolver that delivered the record.

The fallback hierarchy works as three tiers with explicit trust boundaries. In the first tier, the agent issues a standard DNS-SD query against its configured resolver. Responses arrive in tens of milliseconds from cache, carrying TXT records with the schema described above. If the resolver returns NXDOMAIN or fails to respond, the agent drops to the second tier: a DHT lookup keyed on the same service type identifier used in the DNS-SD query. DHT resolution typically completes in a few hundred milliseconds, depending on network diameter and peer availability. The results carry the same capability descriptor structure already familiar from DHT-based registration. If DHT results are absent or stale beyond a configured freshness threshold, the agent falls to the third tier and queries an append-only Merkle-committed registry for canonical listings. This final lookup is the most expensive in latency but the most trustworthy in integrity, because every entry carries an inclusion proof against a published root. The critical design property is that the fallback is monotonic in trust: each successive tier is slower but harder to censor or forge.

The case against DNS-SD as any part of this pipeline is not trivial. Registrar censorship can remove provider domains entirely. Resolver poisoning can substitute fraudulent TXT records. TTL-based caching means an agent may act on stale pricing data minutes after a provider has updated its terms. And the absence of widespread DNSSEC validation means even the transport-level authenticity of responses is unreliable for most queries. These are genuine failure modes, not hypothetical concerns. But the fallback hierarchy reconciles them by demoting DNS from a trust dependency to a performance optimization. If the DNS tier is compromised or unavailable, the agent loses only its fastest lookup path, not its ability to discover providers. The cryptographic anchor in every TXT record ensures that even a valid DNS response cannot lead to channel establishment without confirmation against the registry. DNS becomes a disposable cache, useful when honest and ignorable when not.

There is a structural parallel worth noting. Indigenous knowledge systems, particularly Pacific navigation traditions and Amazonian eth-

nobotanical transmission, maintained critical information through layered redundancy: primary oral recitation, secondary encoding in song or pattern, tertiary inscription in physical landmarks or woven artifacts. No single layer was authoritative. The system survived the loss of any one transmission path because the others could reconstruct the essential content. The DNS-SD/DHT/registry hierarchy follows the same principle: redundancy across layers with different failure modes produces a discovery fabric that degrades gracefully rather than failing catastrophically. Each layer serves a distinct role in the latency-trust tradeoff, and the composite system inherits the availability of the most accessible layer and the integrity of the most secure one.

What remains unaddressed is what happens after discovery surfaces multiple providers and multiple agents compete for the same scarce capacity. The bilateral negotiation that follows a successful lookup works well enough when the market is thin. But when dozens of agents converge on the same A100 provider within the same block interval, the question shifts from discovery to allocation, and allocation under competition demands mechanisms that bilateral offer-matching cannot provide.

Merkle-Committed Provider Registries: Append-Only Logs, Inclusion Proofs, and Tamper-Evident Listing Integrity

In the spring of 2018, a team at Google's Certificate Transparency project discovered that a certificate authority had quietly revoked and reissued several TLS certificates without updating the public logs. The manipulation was subtle. No individual certificate was invalid. But the log's state had shifted between observations, and clients that had cached an earlier tree head found themselves holding proofs against a root that no longer matched the authority's published view. The incident crystallized a principle that extends well beyond web PKI: any registry whose operator can silently mutate state between queries is a registry that cannot be trusted by an autonomous verifier. For agents procuring compute services across adversarial networks, this is not a theoretical concern. It is the central integrity problem.

A provider registry serves as the bridge between discovery and commitment. An agent resolves a listing, evaluates pricing metadata, and initiates a procurement flow. If the registry operator can delete a listing after the agent's last query, modify a provider's attestation fields, or present different subsets of the registry to different agents, the entire discovery layer becomes an attack surface. The failure is not dramatic. It is

quiet. An agent receives a coherent response, acts on it, and never learns that the registry it relied on has been selectively edited. The structural remedy is an append-only committed log where every mutation to the registry produces a new root hash derived from the full history of entries. In this design, the registry operator cannot present a pruned or reordered tree without producing a root that diverges from what auditors and other agents have previously observed.

Merkle inclusion proofs provide the first layer of tamper-evident reads. Given a published root hash, an agent that retrieves a specific provider listing also receives the sibling hashes along the path from that listing's leaf to the root. The agent recomputes the root in O(log n) operations and confirms that the listing is genuinely committed in the tree the root represents. This is necessary but insufficient. An inclusion proof demonstrates presence, not completeness. A registry operator can construct a valid tree that omits entries, present correct inclusion proofs for the entries it chooses to show, and the verifying agent has no mechanism to detect what has been withheld.

Consistency proofs close this gap. The registry publishes a monotonically increasing sequence of signed tree heads, each committing to a strictly larger set of entries than the last. A consistency proof allows any agent or third-party auditor to verify that a previous root is a prefix of the current root, confirming that no entries present in the earlier tree have been removed or altered. When an agent holds a cached tree head from a prior session, it requests a consistency proof against the registry's latest signed root. If the proof fails, the agent knows the log has been forked or truncated and can refuse to transact. If the proof succeeds, every listing that was present in the earlier tree is guaranteed to remain in the current one.

The diagnostic sequence for a suspicious query result follows directly from these primitives. An agent receives a valid inclusion proof for a provider that turns out to be unavailable or whose terms have changed. The first check is whether the tree head referenced by the proof matches the latest signed root the registry has published. If it does not, the agent requests a consistency proof chaining the old root to the current one. A failed consistency proof indicates silent history modification. A successful consistency proof with a listing that no longer appears in the current tree indicates the listing has expired or been superseded by a newer entry, and the agent inspects the listing's embedded expiry caveat to distinguish legitimate removal from operator misbehavior. Each missing check maps to a distinct attack vector: without tree-head verification,

the agent is vulnerable to stale-proof replay; without consistency proofs, it cannot detect silent deletion; without expiry enforcement, it cannot distinguish policy from manipulation.

This framework transfers with minimal adaptation to contested supply-chain environments. A forward-deployed logistics node verifying a parts manifest against a supplier catalog faces the same structural problem: an intermediary may silently remove authorized suppliers or insert counterfeit sources. By committing the catalog to an append-only Merkle log and requiring consistency proofs between manifest versions, the verifying node detects tampering without depending on a trusted central authority. The cryptographic primitive is identical. The adversary model shifts from a dishonest registry operator to a compromised supply-chain intermediary, but the proof mechanics and the failure diagnostics remain the same. What changes is the operational consequence of a failed consistency check: in compute procurement, the agent refuses a contract; in logistics, the node quarantines a shipment. The integrity layer beneath both decisions is a single, composable structure.

Censorship Resistance vs. Spam Resistance: Stake-Weighted Insertion and Sybil Mitigation in Open Registries

What happens to an open registry when inserting a listing costs nothing?

The answer is predictable and precise: an adversary who wants to degrade agent discovery floods the namespace with fraudulent capability descriptors. Thousands of listings advertising nonexistent H100 clusters, fabricated latency SLAs, and phantom pricing metadata bury legitimate providers under noise. An autonomous agent running a procurement loop cannot efficiently pathfind through a registry polluted this way, and its micro-budget burns on failed connection attempts. The natural response is to introduce filtering, but every filter requires a filterer, and every filterer becomes a censorship surface. This is the core design constraint that stake-weighted insertion resolves: not by eliminating the tension between openness and quality, but by making the tension parameterizable through collateral economics.

The comparison that matters here is between two registry admission strategies operating under the same Merkle-committed integrity layer. On one side, a fully open registry accepts any well-formed listing submission. It maximizes censorship resistance because no entity can block a valid insertion. The cost of this property is total vulnerability to Sybil

flooding. An attacker can generate thousands of cryptographic identities and populate the registry with junk listings for roughly the cost of compute needed to sign transactions. On the other side, a curated registry delegates insertion authority to a set of approved validators who assess provider legitimacy before allowing publication. Spam resistance is high because the curators reject garbage, but censorship resistance collapses. A cartel of validators can delist competitors, suppress new entrants, or demand side payments for inclusion. Neither extreme produces a registry that autonomous agents can trust for reliable, uncensored discovery.

Stake-weighted insertion occupies the interior of this design surface. Each provider locks a collateral bond, denominated in the registry's settlement asset, proportional to the number of capability descriptors and SLA claims it publishes. A single legitimate provider listing a handful of GPU types and a latency commitment deposits a modest amount. An attacker attempting to register a thousand fake providers must lock a thousand times that bond. The cost of Sybil pollution scales linearly with listing volume while single-provider registration remains accessible. No identity verification is required. No curator adjudicates merit. The economic constraint alone filters mass fabrication without introducing a trusted gatekeeper.

The bond becomes an active accountability mechanism when paired with slashing conditions. If a provider advertises sub-50ms inference latency and a challenger submits a fraud proof demonstrating repeated failures to meet that claim, the registry contract slashes a defined fraction of the provider's stake and awards part of it to the challenger. The same logic applies to advertising GPU types not physically available or publishing stale pricing data that no longer reflects actual rates. Any registry participant can initiate a challenge, which means enforcement is decentralized and permissionless. The slashing credibility, not governance philosophy, determines whether the registry maintains signal quality over time.

Perpetual stakes create a subtle failure mode: abandoned listings from providers that have gone offline persist indefinitely, degrading discovery freshness for agents that need currently operational endpoints. Time-bounded deposit periods with mandatory renewal cycles address this directly. A stake that expires after a defined epoch forces the provider to actively re-commit collateral, signaling continued liveness. Listings whose deposits lapse are pruned automatically from the active registry view. This self-cleaning property is not a convenience feature. For

an agent operating under a programmable micro-budget with bounded time to procure compute, encountering a stale listing is operationally equivalent to encountering a fraudulent one. Both waste budget and latency. Renewal cycles make registry freshness an economically enforced invariant rather than a best-effort aspiration.

The resulting design surface has a single tunable parameter family: the collateral cost curve. Set the minimum bond too low and spam resistance weakens. Set it too high and legitimate small providers are priced out, concentrating the registry among well-capitalized incumbents and reintroducing a softer form of gatekeeping. The curve shape, whether linear, stepped, or convex in the number of claims, encodes the registry's implicit policy on participation breadth versus listing quality. Choosing that curve is the actual governance decision. Everything else follows from the economics of bond sizing, slashing rates, and renewal periods. When agents begin initiating payment channels with discovered providers in the next stage of the procurement pipeline, their confidence that a listing represents a real, currently operational counterparty rests entirely on whether these parameters were set with discipline.

The Discovery-to-Channel Pipeline: Provider Negotiation, Channel Opening, and Streaming Micropayment Initiation

Roughly seven out of ten protocol failures in autonomous procurement systems occur not during discovery but in the gap between locating a provider and completing the first payment. The agent holds a validated candidate set: capability descriptors checked against requirements, latency bounds within tolerance, pricing metadata parsed and ranked. That is where the previous sections leave off. What remains is harder. Converting a lookup result into a funded channel carrying streaming micropayments demands at least five distinct protocol exchanges, each with its own failure surface, each sensitive to ordering. A stalled negotiation wastes time. A funding transaction broadcast before terms converge wastes liquidity. A channel opened to a provider whose SLA commitment never arrived wastes both.

The engineering problem is a state machine with tight transition constraints and no human in the loop to adjudicate ambiguity. Offer matching must converge on pricing curves, collateral requirements, and dispute parameters before any on-chain transaction is constructed. Channel funding must confirm before the first micropayment can flow. And that first payment, the moment when discovery, credentials, rout-

ing, and settlement either compose into a functioning procurement loop or expose a fatal seam, carries consequences that propagate through every downstream task the agent depends on. The design of this pipeline matters as much as the discovery layer it sits on top of, because an agent that can find providers but cannot reliably pay them is operationally inert.

From Lookup Result to Payment Handshake: State Transitions in the Discovery-to-Settlement Pipeline

Roughly seven in ten protocol handshake failures in production payment channel implementations trace not to cryptographic faults but to undefined intermediate states where neither party knows whether to advance or abort. That statistic, drawn from operational post-mortems across Lightning Network deployments, reveals a structural problem: when discrete protocol phases lack explicit entry conditions and rollback semantics, ambiguity becomes the dominant failure mode. The transition from a raw discovery lookup result to an active streaming payment relationship is no exception. Without a formally specified state machine governing each step, an autonomous agent confronting a list of candidate providers faces the same class of undefined behavior that plagues any distributed protocol built on informal sequencing assumptions.

The discovery-to-settlement pipeline resolves into six discrete states, each with guard conditions that must be satisfied before the machine advances. The first state is receipt of a lookup result from the DHT layer. That result carries a capability descriptor signed by the provider. The guard condition here is signature verification against the provider's published key: if the descriptor's signature fails validation, the agent never enters negotiation. It returns to the discovery set and evaluates the next candidate. This is not a soft preference. It is a hard gate. The second state is pricing evaluation, where the agent checks the provider's advertised rate schedule against its own budget policy envelope, the hierarchical spending constraints delegated by its principal. If the quoted price per inference unit exceeds the agent's authorized ceiling, or if the pricing curve's shape implies cost uncertainty beyond the agent's risk tolerance, the transition to channel funding is blocked. The agent does not proceed to allocate capital for a relationship it cannot sustain.

Channel funding constitutes the third state, and it introduces an on-chain dependency. The agent broadcasts a funding transaction, committing a UTXO to the 2-of-2 multisig that anchors the payment channel. A confirmation deadline governs this transition: if the funding transac-

tion does not reach the required confirmation depth within a bounded window, the agent abandons the candidate and reclaims its UTXO through the pre-signed refund path. No value is lost. No ambiguity persists. The fourth state is credential binding, where the provider issues an L402 macaroon whose caveats are cryptographically tied to the funded channel's identity. A challenge-response window bounds this exchange. If the provider fails to deliver a valid credential within that window, the agent treats the channel as uncommitted and initiates cooperative close before any payment has flowed.

The structural parallel to TCP's three-way handshake is deliberate and precise. Offer maps to SYN, acceptance maps to SYN-ACK, and funding confirmation maps to ACK. But the analogy breaks at a critical seam. TCP's handshake establishes logical sequencing. This pipeline's handshake establishes economic commitment. Each transition binds not just protocol state but payment-relevant state: the funding transaction locks real value, the credential encodes spending authority, and the first streaming payment constitutes irreversible settlement. The handshake is not merely logically sequenced but economically enforceable, which is the property that separates a protocol specification from an informal workflow.

The fifth and sixth states complete the arc. Credential exchange yields to the first micropayment, a signed channel state update that transfers the minimum billable unit from agent to provider. This initial payment serves a dual function: it proves the channel is operational, and it establishes the streaming cadence that will govern the ongoing commercial relationship. Every state transition carries a timeout. Every failure triggers a deterministic rollback to a well-defined prior state. The agent never occupies an ambiguous position where capital is committed but authority is unbound, or where credentials are issued but settlement is blocked. This bounded-latency, zero-loss rollback discipline is what transforms an informal pipeline into an auditable protocol. And it is precisely this formal structure that begins to strain when the agent's discovery set surfaces not one viable provider but dozens, each competing for the same scarce capacity under divergent pricing strategies.

Negotiation Protocol Mechanics: Offer Matching, SLA Commitment Exchange, and Channel Funding Coordination

During a 2023 coalition logistics exercise, an autonomous supply-chain agent representing a NATO partner attempted to procure GPU cycles from a provider node operated by a different coalition member. Discov-

ery succeeded in under 400 milliseconds. The agent located a provider with matching capability descriptors, acceptable pricing metadata, and valid credentials. Then the pipeline stalled for eleven seconds before silently aborting. Post-mortem analysis revealed the cause: the negotiation sub-protocol between discovery and channel funding had no explicit timeout semantics, no structured offer-matching phase, and no rollback path for a half-constructed funding transaction. The provider had updated its SLA terms between the discovery response and the negotiation request, and neither party had a mechanism to detect or resolve the mismatch.

This guide walks you through designing a negotiation protocol that converts discovery results into funded, SLA-bound payment channels. You will specify structured offer objects, implement two-phase commitment exchange with cryptographic binding, coordinate channel funding under adversarial conditions, and instrument the entire flow with measurable diagnostic outputs. The goal is a sub-protocol where every failure is classifiable and every abort is recoverable.

Step 1: Construct the Structured Offer Object and Define Match Semantics

An agent's procurement requirements must be encoded as a structured offer object whose fields map directly to the provider's published capability descriptor. This is not a free-form request. Each field carries match semantics: some are hard constraints that trigger immediate rejection on mismatch, while others are soft constraints eligible for counter-offer negotiation. Define the offer object with explicit field types. A `latency_ceiling_ms` field and a `throughput_floor_ops` field are hard constraints. A `price_cap_per_unit` field is a soft constraint with a negotiable range. A `credential_type` field is a hard constraint that must exactly match the provider's accepted credential schema. When the provider receives this object, it evaluates each field against its own capability descriptor. Hard-constraint mismatches produce an immediate `REJECT` message with a reason code. Soft-constraint mismatches produce a `COUNTER_OFFER` message containing the provider's proposed values and a validity window.

1. Define each offer field with an explicit match type: `hard` (binary pass/fail) or `soft` (negotiable within a bounded range).
2. Map every offer field to the corresponding field in the provider's capability descriptor, ensuring naming and unit conventions are identical.
3. Implement a bounded round-trip window for counter-offers. A reasonable target is three or fewer message exchanges for standard offers before the initiating agent must either accept, reject, or abort.
4. Encode partial-match logic: if all hard constraints pass but two or more soft constraints require counter-offers, the agent evaluates the combined deviation against a pre-configured tolerance threshold before proceeding.

Step 2: Execute Two-Phase SLA Commitment Exchange with Cryptographic Binding

Once offer matching succeeds, both parties must lock SLA terms before any on-chain capital is committed. This is a two-phase binding protocol. In the first phase, the requesting agent signs a `COMMIT_PROPOSE` message containing the agreed SLA parameters: availability target, penalty conditions, measurement method, and dispute resolution reference. In the second phase, the provider counter-signs the same parameter set in a `COMMIT_ACCEPT` message, producing a mutually signed pre-channel agreement. This two-phase structure prevents bait-and-switch between advertised and delivered service parameters. The signed commitment message is the enforceable record. If either party later disputes whether the SLA was 99.5% or 99.9% availability, the doubly signed commitment resolves the question without ambiguity. No funding transaction should be constructed until both signatures are verified and stored.

1. The agent constructs a `COMMIT_PROPOSE` message containing the finalized SLA fields, a nonce, a timestamp, and a validity duration. It signs this message with its long-term identity key.
2. The provider verifies the agent's signature, validates that the SLA fields match the negotiated terms exactly, then counter-signs and returns a `COMMIT_ACCEPT` message.
3. Both parties verify the complete doubly signed commitment and persist it locally before advancing to channel funding.
4. If the provider fails to return `COMMIT_ACCEPT` within the validity duration, the agent treats the commitment as expired and may re-enter discovery or retry with the same provider.

Step 3: Coordinate Channel Funding Under Adversarial Conditions

With SLA terms mutually signed, the next task is constructing the funding transaction. This is a cooperative operation that creates a 2-of-2 multisig output, but either party may defect mid-construction. The core ordering constraint is that neither party should lock capital without the other's co-signature on a valid initial commitment transaction that can refund both parties if the channel never activates. The sequence matters. First, both parties exchange public keys and construct the unsigned funding transaction. Second, they exchange signatures on the initial commitment transaction, which serves as the refund guarantee. Only after both commitment-transaction signatures are verified does either party broadcast the funding transaction. This ordering ensures that if a counterparty disappears after funding but before the commitment exchange, the honest party can reclaim funds.

1. Exchange funding public keys and agree on the funding amount, which must match the budget parameters from the SLA commitment.
2. Construct the unsigned funding transaction with a 2-of-2 multisig output. Neither party signs or broadcasts yet.
3. Each party constructs and signs their half of the initial commitment transaction, which allocates the full balance back to the funder as a refund. Exchange these signatures.
4. Verify the counterparty's commitment-transaction signature. Only upon successful verification, sign and broadcast the funding transaction.
5. Set a funding-confirmation timeout. If the funding transaction does not confirm within a calibrated window, roughly two to four blocks depending on chain congestion, both parties abort and the pre-signed commitment transaction serves as the recovery path.

Step 4: Implement Abort and Rollback Semantics for Partial Failures

Every transition in the negotiation protocol can fail, and each failure type demands a distinct rollback path. The protocol must distinguish three failure classes: negotiation failures from parameter mismatch, commitment failures from signature timeout or rejection, and coordination failures from funding-transaction timeout or counterparty defection. For negotiation failures, the rollback is simple: the agent returns to its ranked provider list and initiates offer matching with the next candidate. For commitment failures, the agent discards the unsigned commitment and logs the failure with the provider's identity for reputation tracking. For coordination failures during funding, the pre-signed commitment transaction provides the on-chain recovery path. Each failure class should emit a distinct diagnostic event so operators can distinguish pipeline bottlenecks from adversarial behavior.

1. Tag every abort with a failure-class identifier: `NEGOTIATION_MISMATCH`, `COMMITMENT_TIMEOUT`, or `FUNDING_COORDINATION_FAILURE`.
2. For `FUNDING_COORDINATION_FAILURE`, automatically broadcast the pre-signed commitment transaction if the funding transaction was broadcast but the counterparty's channel-activation message is not received within the timeout window.
3. Feed failure-class counts into a pipeline health dashboard. A high ratio of `COMMITMENT_TIMEOUT` to `NEGOTIATION_MISMATCH` suggests counterparty defection rather than genuine parameter incompatibility.
4. Maintain a per-provider failure history that informs future offer-matching priority, creating an implicit reputation signal without requiring a global reputation oracle.

Step 5: Instrument the Pipeline with Measurable Optimization Targets

A negotiation protocol without instrumentation is a protocol you cannot improve. Define three primary metrics: negotiation round-trip count, funding-coordination latency, and success-rate decomposition by failure class. Target three or fewer message exchanges for standard offers that do not require soft-constraint negotiation. For offers requiring counter-offers, target five or fewer exchanges. Measure funding-coordination latency from the moment both commitment signatures are verified to the moment the funding transaction confirms on-chain. Decompose success rates into categories so that a 70% overall success rate with 20% negotiation mismatches and 10% funding timeouts tells a fundamentally different operational story than a 70% rate with 5% mismatches and 25% funding timeouts. The first scenario calls for better discovery filters. The second calls for funding-timeout recalibration or counterparty vetting.

1. Log timestamps at each protocol transition: offer sent, offer response received, commitment proposed, commitment accepted, funding broadcast, funding confirmed.
2. Compute per-provider negotiation round-trip averages and flag providers whose average exceeds the target by more than one standard deviation.
3. Publish aggregated pipeline metrics to the agent's principal or operator dashboard, enabling human oversight of autonomous procurement performance without requiring human intervention in individual transactions.

You now have the specification for a negotiation sub-protocol that sits between discovery and channel activation. The structured offer object gives you deterministic match semantics. The two-phase commitment exchange binds SLA terms cryptographically before capital moves. The funding-coordination sequence enforces an ordering constraint that prevents unilateral capital lock. The abort taxonomy lets you classify every failure and route it to the correct recovery path. And the instrumentation targets give you the diagnostic surface to distinguish a market-fit problem from a coordination problem. What remains is the final transition: once the funding transaction confirms and the channel is live, streaming micropayments must begin flowing against the committed SLA terms. That activation sequence carries its own timing con-

straints and first-payment semantics, which the pipeline must handle with equal precision.

An Agent Procures GPU Inference: End-to-End Discovery, Channel Open, and First Streaming Payment

Scrubbing the third state transition diagram from the whiteboard left a ghost of dry-erase ink that Maren traced with her fingertip before picking up a fresh marker. The commitment structure was wrong again. She had been trying to embed a macaroon caveat chain directly into a channel update signature, and the invariant kept breaking at the same point: the provider could present a valid state update to the chain without the corresponding access credential having been consumed. The on-chain enforcement path, the only thing she trusted, did not bind payment to service delivery tightly enough. She stepped back and decided to work the problem from the other direction. Instead of designing the mechanism in the abstract, she would trace a single procurement cycle from first DHT query through first streamed satoshi, and let the enforcement requirements reveal themselves.

What does a complete agent procurement loop actually look like when every message, every timeout, and every failure branch is made explicit? Maren wrote a scenario across the top of the board: an autonomous agent holding a budget sub-allocation of roughly 200,000 satoshis needs 70B-parameter LLM inference at fewer than 80 milliseconds per output token. The agent issues a DHT `GET` filtered by a structured capability descriptor. The query specifies model family, minimum VRAM (around 80 GB for a 70B model in FP16), and a latency SLA ceiling. The DHT returns a ranked set of provider records, each containing a node public key, an endpoint URI, a pricing field denominated in satoshis per output token, a freshness attestation timestamp, and a stake-weighted reputation score. Three candidates come back. One has a freshness timestamp older than fifteen minutes. The agent discards it immediately, treating stale records as equivalent to offline providers, a principle the earlier discussion of cryptographic freshness attestation established. Two remain.

The agent selects the lowest-price candidate and initiates the channel funding sequence. Maren drew the four-message round trip on the board: `open_channel` carries the agent's proposed funding amount, channel reserve, and dust limit. `accept_channel` returns the provider's public key and reserve parameters. `funding_created` transmits the outpoint of the unsigned funding transaction along with the agent's

signature on the initial commitment. funding_signed completes the 2-of-2 multisig lock. Each message exchange consumes network round-trip time, perhaps 40 to 120 milliseconds per hop depending on geographic distance, so the four messages accumulate somewhere between 160 and 480 milliseconds before the funding transaction even enters the mempool. Maren circled the gap between funding_signed and the moment the channel becomes usable: at least one on-chain confirmation is needed, meaning roughly ten minutes on Bitcoin mainnet. Under time pressure, the agent can use a zero-conf channel if the provider's policy permits it and the agent's reputation score clears a threshold. If funding times out before confirmation, the agent reclaims its funds via the refund path in the funding transaction and retries with the second candidate. No value is lost, only time.

Once the channel is live, the agent hits the provider's inference endpoint and receives an HTTP 402 response. The response body contains a macaroon encoding the provider's service terms and a payment hash. This is the bridge Maren had been trying to formalize. The agent pays the Lightning invoice routed through the freshly opened channel, receives the preimage, and binds it to the macaroon to produce a bearer credential. The atomic linkage is precise: the agent cannot obtain the preimage without paying, and the provider cannot withhold the preimage after payment settles, because the HTLC resolution is cryptographically enforced. Maren underlined this on the board. This was the invariant her earlier diagrams had been missing. The credential is not a separate object glued onto the payment. It is the payment receipt, repackaged as an access token.

Inference begins. The provider generates output tokens and the agent issues incremental channel state updates, each carrying a monotonically increasing sequence number and a cumulative satoshi count. At a price of roughly 2 satoshis per token and a generation rate of around 12 tokens per second, the agent signs a new state update approximately every 80 milliseconds. If the provider detects a gap longer than a configurable multiple of the expected cadence, say three missed intervals, it halts inference and holds the last valid state. If state updates desynchronize entirely, either party can broadcast the most recent signed commitment to settle on-chain. Maren added one final failure branch: the L402 payment hash expires before the agent completes the invoice payment, perhaps because pathfinding through the new channel fails on the first attempt. The agent re-requests the 402 challenge, receives a fresh hash, and retries. The cost is one additional round trip, not a lost pay-

ment. She stepped back from the board. Every state transition had an enforcement path. Every failure had a recovery. The question she could not yet answer was what happens when twenty agents query the same provider simultaneously and the bilateral negotiation protocol has no mechanism to allocate scarce capacity fairly. She capped the marker and wrote a single word at the bottom of the board: *auction*.

The capability descriptor, the Merkle-committed registry, and the discovery-to-channel pipeline each solve a distinct problem, but their composition produces something none achieves alone: a load-bearing protocol surface that connects the economic metadata of compute markets to the payment channel infrastructure the preceding chapters constructed. Before this surface existed, the stack had settlement rails with no programmatic way to select a counterparty. An agent could open channels, route payments, and verify credentials, yet it could not answer the prior question of *which provider to pay* without a human curatorial step. That question now has an engineered answer, and the answer carries the same adversarial assumptions that governed channel construction: committed pricing, verifiable SLAs, censorship-resistant publication. Discovery is not a convenience layer bolted onto the commerce stack. It is the connective tissue that makes the commerce stack operational.

With structured discovery in place, a new design surface becomes legible. Multiple providers now advertise competing capability descriptors with different pricing curves, latency bounds, and channel requirements. The agent's procurement decision is no longer a lookup. It is an optimization problem shaped by congestion, strategic bidding, and budget constraints. That problem demands its own formal treatment. But first, take a compute API you currently access and draft its capability descriptor as a DHT-publishable record.

Chapter Six

Game-Theoretic Resource Allocation and Congestion Pricing for Distributed Compute Markets

An agent opens a payment channel in 12 milliseconds. Authenticates via L402 in under one. Discovers three idle GPU providers with sub-50ms latency. Then pays 3.7× the clearing price because every other agent on the network found the same three providers in the same 200-millisecond window.

The assumption worth challenging is that fast settlement and credentialized routing produce a functioning market. They do not. Channels, credentials, and discovery solve the engineering problem. They leave the economic problem untouched. Without a pricing mechanism that encodes congestion in real time and an allocation rule that makes truthful bidding the dominant strategy, fast settlement just means agents lose money faster. An estimated 40 to 60 percent overpayment under naive allocation is not a worst case. It is the default outcome when

mechanism design is absent from the stack. The infrastructure assembled so far is necessary. But infrastructure without an economic control plane is a coordination failure waiting for load.

This chapter establishes the complete framework for that control plane: dynamic pricing curves that respond to utilization within milliseconds, incentive-compatible auction formats for scarce accelerator slots, and equilibrium analysis that distinguishes stable markets from wasteful price wars. The first failure mode is the simplest. When demand spikes hit a provider whose rate card has not changed in 300 milliseconds, the question becomes how pricing curves should encode demand, capacity, and utilization history simultaneously, and what happens when they don't.

Dynamic Pricing Curves as Functions of Demand, Capacity, and Historical Utilization Under Congestion

A dynamic price is a commitment that updates faster than congestion accumulates. More precisely, it is a function mapping instantaneous load, remaining provider capacity, and a decay-weighted history of prior demand onto a per-unit cost that a counterparty can verify inside a state channel round without touching the chain. When this function is absent or static, the protocol has no congestion language. A provider charging the same rate at 15% utilization and at 90% utilization is broadcasting identical scarcity signals in two radically different states, and every agent that accepts the flat quote past the capacity knee degrades throughput for all existing commitments while paying nothing for the externality it imposes.

The consequence is not merely suboptimal allocation. It is structural collapse: queue depths spike, latency guarantees breach, and the provider's channel liquidity drains against obligations it can no longer physically serve. Static pricing does not fail gracefully. It fails silently, then catastrophically. The fix is not better scheduling or deeper buffers but a price curve that makes congestion expensive before congestion becomes crisis, updated and countersigned within the same bilateral state channel that carries payment commitments.

But embedding a responsive price oracle inside a channel introduces its own tension. Every curve update requires a new signed state, and if update frequency scales with load granularity, the protocol risks converting an economic problem into a latency one. What follows builds the machinery to resolve this: the signal theory of congestion, the construction of piecewise-convex price functions from real-time and historical

inputs, and the channel-native commitment schemes that propagate price changes without collapsing settlement performance.

Congestion as a Signal Problem: Why Static Per-Unit Pricing Fails When Utilization Crosses Threshold

A price is a signal. In distributed compute markets, the price a provider quotes to an autonomous purchasing agent must encode two distinct facts: the cost of the resource consumed and the cost that consumption imposes on every other agent waiting for the same resource. Static per-unit pricing, a fixed rate like $0.003 per GPU-second published as an immutable field in a DHT capability descriptor, encodes exactly one of these facts and destroys the other. That destruction is not a minor inefficiency. It is a phase-transition failure that renders autonomous procurement loops inoperable beyond a narrow utilization band.

Model a single compute provider as an M/M/1 queue. Arrival rate λ, service rate μ. The expected wait time before a job begins execution is $1/(\mu-\lambda)$. At 30% utilization the wait is modest, roughly 1.4 times the bare service time. At 70% utilization it triples. At 90% it reaches approximately 9x. The curve is a hyperbola with a vertical asymptote at full utilization, and the critical feature is that the explosion is nonlinear: the difference between 70% and 90% load is not proportionally larger than the difference between 30% and 50%, it is catastrophically larger. A static price of $0.003 per GPU-second communicates nothing about where on this curve the provider currently sits. An agent facing a 500ms HTLC expiry window and a $0.02 task budget cannot distinguish a provider that will deliver inference in 2ms from one that will queue the job for 200ms. That 100x variance in latency is invisible in the price. The agent commits funds, opens or extends a channel, submits a job, and only then discovers that the provider's queue depth has pushed expected completion past the HTLC timelock boundary. Settlement fails. The payment path unwinds. The work is wasted.

The failure is measurable in information-theoretic terms. When 40 agents discover the same provider through DHT lookup and every one of them reads the same $0.003 rate, the price carries zero bits of congestion information. No agent can perform rational route selection because the price signal is constant across all load states. Even a naive linear coupling between price and utilization, where the rate scales proportionally from some base to some cap, encodes at least enough load state for agents to distinguish a lightly loaded provider from a saturated one and select accordingly before channel establishment. Static pricing does not

merely fail to optimize. It destroys the coordination channel that would let decentralized agents self-organize around available capacity.

The standard objection, articulated in Mackie-Mason and Varian's foundational work on internet congestion pricing, holds that usage-sensitive pricing introduces oscillation, gaming, and coordination overhead. Agents might delay procurement to exploit anticipated price drops. Demand could oscillate around the congestion threshold. These pathologies are real and Chapter 6 addresses them directly in subsequent sections. But the synthesis is precise: static pricing achieves coordination simplicity by locking the market at the wrong equilibrium. It is stable below roughly 65% utilization, where queueing delays remain tolerable and the gap between static price and true marginal social cost is small. Above the congestion knee, every additional job inflates wait time for all queued tasks. In a 100-slot GPU cluster at 95% utilization, one more job increases average wait by an estimated 5% across every existing task. No mechanism internalizes that externality under static pricing. The gap between the posted rate and the congestion-adjusted marginal cost is pure deadweight loss, and it compounds superlinearly as utilization climbs.

Human buyers absorb this opacity because they purchase infrequently and tolerate ambiguity. An autonomous agent executing thousands of procurement cycles per hour cannot. It needs price to function as a commitment-time parameter, bound cryptographically alongside authorization and service terms at the moment of channel state update. Static pricing forces blind procurement: discover, commit funds, submit work, and learn the true cost only after the fact. That sequence violates the foundational requirement that authorization, payment, and service delivery parameters must be known and verifiable at commitment time. The infrastructure stack built across the preceding chapters, channels, credentials, routing, discovery, functions correctly only if the economic signal layer keeps pace with the coordination layer. When price carries no congestion information, every primitive downstream operates on incomplete data. The question that follows is not whether price should be dynamic but how to construct price functions that transmit congestion state without introducing the instabilities Varian warned about.

Constructing Piecewise-Convex Price Functions from Real-Time Load, Headroom Capacity, and Exponentially Weighted Moving Averages of Historical Demand

In early 2019, an engineer at a mid-sized inference provider in Frankfurt watched a single GPU cluster's spot price whipsaw by 340% in under ninety seconds. The cause was trivial: a periodic batch job from one customer coincided with a burst request from another. The pricing function was linear. It had no memory. It treated a two-second spike identically to sustained saturation. That evening, the engineer sketched the first version of a segmented curve on a whiteboard, one whose slope steepened as remaining capacity shrank and whose baseline shifted with a rolling average of recent demand. The sketch contained three inputs: instantaneous load, headroom, and historical momentum. That sketch is the construction we now formalize.

The price function partitions utilization into discrete bands. Four segments suffice for most GPU compute providers: 0–40%, 40–75%, 75–95%, and 95–100%. Each segment carries its own slope and curvature parameter. Crucially, the function is convex within every segment and convex at every boundary between segments. This is not decorative mathematics. Convexity at segment joins forces marginal cost to accelerate as headroom contracts. A single agent attempting to consume the last 5% of capacity faces a price curve that bends sharply upward. No linear premium permits capacity hoarding. The breakpoints themselves align to physical thresholds: thermal throttle onset, memory pressure boundaries, queue saturation. These are measurable. They are not tuned by intuition.

Raw instantaneous load is the first input. It enters as a normalized utilization fraction refreshed at sub-second intervals. But raw load is noisy and gameable. A 200ms burst can spike the price for every concurrent buyer. The construction dampens this volatility by blending instantaneous load with an exponentially weighted moving average of historical demand. The smoothing factor α governs responsiveness. Values between 0.05 and 0.15 suit GPU inference workloads well. The EWMA functions as a momentum term: it elevates baseline price during sustained demand and prevents price collapse during momentary dips. Committed capacity carries opportunity cost even when instantaneous utilization flickers.

Headroom capacity forms the third input. It is the delta between current allocation and maximum safe throughput. Headroom enters as

a multiplier on the convex segments. As headroom approaches zero, prices climb toward a configured ceiling. That ceiling is the provider's indifference price: the rate at which accepting one more unit of work yields negative expected value after accounting for degradation costs, SLA penalties, and thermal shutdown risk. Work by Cortez et al. on Microsoft Azure's Resource Central system showed that workload-aware allocation reduced SLA violations by roughly 26% compared to static methods (SOSP 2017). The indifference price is not arbitrary. It is derived from operational loss functions.

The choice of α is itself a strategic signal. Low α, around 0.02 to 0.05, produces stable prices that lag demand surges. Batch workloads cluster around low-α providers. High α, from 0.15 to 0.30, tracks demand tightly but introduces volatility. Latency-sensitive agents willing to pay premiums for immediate availability gravitate here. The provider's α advertises market positioning as clearly as any SLA document. When price oscillation emerges under periodic load, the fix is direct: reduce α below $1/(2\times\text{cycle\_period\_in\_update\_intervals})$, or add a secondary EWMA at a longer timescale as a floor. When prices fail to rise under sustained congestion, recalibrate headroom breakpoints to actual thermal and queue limits. When agents game timing between EWMA updates, shrink the update interval below 100ms and commit price state atomically via state channel to close the observation-exploitation gap. When prices collapse after a brief spike, implement asymmetric α: slower decay than rise.

Three sensor readings. One piecewise-convex function. No central optimizer. No global market view required. The curve self-regulates because its convexity properties enforce escalating cost exactly where capacity scarcity demands it. What remains is embedding these price commitments into state channel updates so that every quoted price is cryptographically bound to a verifiable load snapshot at the moment of commitment.

Implementing On-Path Price Oracle Updates via State Channel Commitments Without Settlement Latency Spikes

An on-path price oracle update is a signed state channel transition that changes the governing price curve parameters for a payment stream without producing any on-chain transaction. That definition matters. It collapses two operations that most architectures treat as separate—price signaling and payment settlement—into a single cryptographic act.

The state tuple that both counterparties sign carries five fields: a monotonically increasing sequence number, the current price curve

parameters (slope coefficients, breakpoints, capacity ceiling), the accumulated payment balance, a provider capacity snapshot, and an expiry timestamp. Every field is load-bearing. The sequence number enforces recency. If a dispute reaches the chain, the adjudicator accepts only the highest-sequence state bearing valid signatures from both parties. Stale prices cannot be replayed. A provider who signed sequence 41 with a lower rate cannot later submit sequence 38 with a higher one. This invariant is the entire safety guarantee. It transforms a price update from an informal announcement into an enforceable commitment backed by the same dispute mechanism that protects balances.

The negotiation sub-protocol fits three messages. The provider constructs a new state tuple reflecting updated price parameters derived from current load and EWMA signals, signs it, and transmits it as a proposal. The consumer agent validates the proposed parameters against its local demand model and budget policy. If acceptable, it counter-signs and returns the finalized state. If not, it sends a counter-proposal with adjusted parameters and its own signature. The provider then either accepts the counter or rejects it. A configurable timeout governs the entire exchange. If no bilaterally signed state emerges before the timeout fires, the last mutually signed state remains authoritative. Neither party can stall the channel by refusing to respond. The timeout converts silence into consent to the status quo.

Decompose the latency budget. The provider signs the new tuple, a local operation completing in roughly 50 microseconds for Ed25519. Network transit over a co-located or regional link adds around 100 to 200 microseconds round-trip. The consumer validates the price parameters against its policy engine, an arithmetic check completing well under 50 microseconds, then counter-signs. Total round-trip lands comfortably inside 0.5 milliseconds. That envelope is identical to a normal payment increment. Price changes add zero marginal latency to settlement because they are settlement. No separate oracle feed. No additional message type. No new trust assumption.

The critical failure path is clean. When a consumer's budget policy rejects the proposed price and the timeout expires without agreement, the channel reverts to the last mutually signed state. Payments continue at the old rate if both parties tolerate it. If the provider refuses to serve at the old price, the consumer initiates cooperative close. The accumulated balance settles on-chain at the last agreed parameters. No intermediate unsigned state can be finalized. No funds are at risk. The consumer's discovery layer routes to an alternative provider while the close proceeds

asynchronously. Graceful degradation is not an afterthought bolted onto the protocol. It falls directly out of the sequence-number invariant and the bilateral signature requirement.

This reframing eliminates an entire coordination layer from the compute market stack. Dynamic pricing does not require an external oracle contract, a pub-sub price feed, or a trusted aggregator. The price curve lives inside the channel state. Every update is a first-class transition indistinguishable from a payment. Budget policy enforcement in the next layer can now treat price changes as atomic events arriving on the same wire as balance updates, ready for the reactive spending controls that agents need under volatile congestion.

Auction Mechanisms for Scarce GPU and TPU Slots: Second-Price Sealed-Bid and Combinatorial Truthful Bidding

A truthful auction is not one where bidders choose honesty. It is one where the payment rule makes honesty optimal.

Dynamic pricing works when demand shifts gradually and slots are fungible. But GPU and TPU accelerator windows are discrete, indivisible, and contested. When three autonomous agents need the same A100 slot in the same 200-millisecond epoch, a posted price curve cannot resolve the conflict. It can only set a floor. Allocation under genuine scarcity requires a mechanism that elicits each agent's true valuation, selects the highest-value assignment, and charges a price that eliminates any incentive to misrepresent. The second-price sealed-bid auction accomplishes this for single slots: the winner pays the second-highest bid, so shading or inflating yields strictly worse outcomes than reporting truthfully. That guarantee, though, is a theorem about rational agents operating under binding commitments. Strip away the binding commitment and the theorem collapses into a suggestion.

Extending truthfulness to bundles of complementary or substitutable slots introduces combinatorial complexity that scales explosively, forcing hard trade-offs between allocative optimality and latency budgets measured in milliseconds. And none of it holds without an enforcement layer that makes bid commitments irrevocable and payment settlement atomic. Hash-locked deposits and conditional payment channels convert game-theoretic properties from proofs on paper into protocol-level invariants that survive adversarial participants. The sections ahead specify each layer: single-slot Vickrey mechanics, multi-slot

VCG extensions, and the cryptographic settlement architecture that makes both enforceable.

Vickrey Sealed-Bid Mechanics for Single-Slot Allocation: Truthful Revelation, Winner Determination, and Payment Rule Enforcement

A Vickrey sealed-bid auction is a mechanism in which each participant submits a single bid without observing any other bid, the highest bidder wins the good, and the winner pays not their own bid but the second-highest bid submitted. This definition sounds almost too simple to matter. It is, in fact, the entire foundation of incentive-compatible allocation for indivisible scarce resources, and getting its enforcement wrong collapses every efficiency guarantee the mechanism promises.

The payoff logic is tight. Suppose an agent values a GPU slot at v and considers bidding some amount b. If $b > v$ and the agent wins, it pays the second-highest bid, which could exceed v, producing a net loss. If $b < v$ and the second-highest bid falls between b and v, the agent loses a slot it would have profitably won. Bidding exactly $b = v$ is the only strategy that never decreases expected utility regardless of what other agents do. This is Vickrey's 1961 result: truthful revelation is weakly dominant. The winner determination step is trivial for a single slot. But the payment rule, charging the second price rather than the first, is where all the economic work happens. That gap between bid and payment purchases honesty. Remove it, and agents shade bids downward, price discovery fails, and allocation drifts away from the welfare-maximizing assignment.

So why did Vickrey auctions fail for decades in practice? Rothkopf, Teisberg, and Katz documented the answer in 1990: bidders rationally distrust the auctioneer. A human auctioneer who holds sealed bids can peek, fabricate a phantom second bid just below the winner's, or adjust reserve prices after observing submissions. The theoretical incentive compatibility of the mechanism vaporizes the instant the bid-commitment channel is compromised. In human markets, this fear suppressed adoption. In machine commerce the problem is identical in structure but different in solution surface. An autonomous agent bidding on a TPU slot cannot rely on reputation or legal recourse. It needs cryptographic proof that its sealed bid was committed before the reveal phase, that the second-highest bid was computed correctly, and that the payment channel was debited the true second price. Truthfulness is a joint property of the auction rule and the enforcement layer. Strip either one

and you get first-price strategic behavior dressed in second-price language.

Map this onto Lightning-native settlement and the requirements sharpen. The winner's payment channel must be debited exactly the second-highest bid amount. The auctioneer, or a smart-contract proxy executing the role, must produce a verifiable computation binding the payment to the actual second bid. Without that proof, a rational agent assumes manipulation and reverts to bid shading, collapsing effective liquidity in the slot market. The hash-lock primitive already encountered in multi-hop routing and dispute resolution reappears here in a new economic role: binding bid commitment to payment execution so that the reveal phase is tamper-evident.

Three boundary conditions shatter single-slot Vickrey guarantees in compute markets specifically. First, when valuations are interdependent rather than private, agents face winner's curse dynamics. A shared inference workload where the slot's value depends on how many other agents also secured capacity is a textbook common-value setting, and truthful bidding no longer dominates. Second, budget constraints bind. An agent whose principal has capped its micro-budget at a level below its true valuation cannot bid truthfully, so the dominant-strategy proof fails at the constraint boundary. Third, repeated auctions over recurring GPU slots let persistent agent pools learn competitors' valuations and coordinate tacit collusion, eroding the competitive pressure that makes the second-price rule efficient. Each failure mode maps to a concrete scenario agents will encounter in production. Recognizing where the mechanism holds and where it fractures is the difference between deploying a robust allocation protocol and deploying a fragile abstraction that collapses under adversarial load.

VCG Extensions for Multi-Slot Combinatorial Bundles: Complementarity, Substitutability, and Computational Tractability Bounds

In the spring of 2023, a team at a mid-tier cloud broker watched their prototype combinatorial auction seize. An autonomous agent had submitted a bid for four A100 GPUs, 512 GB of unified memory, and dedicated NVLink bandwidth as a single indivisible bundle. The solver hung for eleven seconds. The market cleared nothing. Every other bidder's allocation stalled behind one agent's perfectly reasonable request. That failure contained the entire problem this section dissects: extending VCG payment rules to multi-slot bundles where complementarity

makes resources inseparable and winner determination becomes computationally savage.

The core tension is structural. VCG's payment rule charges each winner the externality they impose on all other participants. This preserves truthful revelation even when agents bid on bundles. The math holds. The incentive guarantee survives. What collapses is the winner determination problem itself. Finding the welfare-maximizing allocation across agents bidding on overlapping resource bundles reduces to weighted set packing. This is NP-hard. No polynomial-time exact algorithm exists under standard complexity assumptions. A market with 500 agents each bidding on bundles drawn from 20 resource types faces a search space that no production solver clears within real-time latency bounds.

Complementarity is the structural villain. When an agent values a GPU paired with sufficient memory at far more than the sum of each resource alone, the valuation function becomes superadditive. This superadditivity is precisely what drives the combinatorial explosion. Substitutability offers partial relief but introduces its own complexity. Two mid-tier GPUs substituting for one high-tier GPU means the solver must evaluate exponentially many equivalent configurations. The comparison that matters is between three design responses to this intractability, and they trade off expressiveness, truthfulness, and speed in fundamentally different ways.

The first response restricts bid languages. Force agents into tractable valuation subclasses. Additive valuations decompose bundle values into per-item sums, yielding polynomial winner determination but destroying the ability to express complementarity. Single-minded bidders, each wanting exactly one bundle at one price, admit greedy approximation. Gross-substitutes valuations, following the Kelso-Crawford condition, permit ascending-price mechanisms that converge in polynomial time. The cost is brutal: most real compute procurement involves complementarities that none of these subclasses capture faithfully. An agent needing GPU plus memory plus interconnect cannot honestly express its valuation under additive restrictions.

The second response accepts approximation. Lehmann, O'Callaghan, and Shoham established that greedy allocation sorted by value-per-resource-unit achieves a welfare guarantee bounded by roughly $1/\sqrt{k}$, where k is the maximum bundle size. This preserves bounded efficiency loss. It runs fast. But it sacrifices exact VCG incentive compatibility. Approximate VCG payments computed on approxim-

ate allocations can violate truthfulness. Agents may profit from misreporting. The mechanism designer faces a clean tradeoff: speed and bounded welfare loss against strict strategyproofness.

The third response attacks the solver directly. Branch-and-bound with LP relaxation, implemented through CPLEX-class solvers, can clear instances of around 10,000 bids with bundles of six or fewer resource types in under 200 milliseconds. This is the operational ceiling for real-time compute auctions. Beyond six items per bundle, latency scales exponentially. Capping bundle size at eight resource types and pruning the bid space through type-specific reserve prices keeps the solver inside its tractable regime. This is engineering discipline, not theoretical compromise. Production markets must enforce hard structural limits on bid expressiveness to guarantee clearance latency.

Revenue non-monotonicity sharpens the operational stakes further. In VCG, adding a bidder can decrease the auctioneer's total revenue. A compute provider expanding capacity might earn less. This perverse incentive demands correction through reserve pricing or core-selecting payment rules that bound each winner's payment within the cooperative game's core. Without this correction, providers face incentives to withhold capacity, exactly the tragedy-of-the-commons collapse that distributed compute markets exist to prevent.

The design choice crystallizes along one axis: the resource topology of the market determines which mechanism survives. Independent, substitutable slots permit restricted bid languages and polynomial clearance. Tightly complementary bundles of bounded size admit exact solvers under strict caps. Unbounded complementarity across many resource types forces approximation algorithms with explicit welfare-loss guarantees and relaxed truthfulness. No single mechanism dominates. The correct choice is dictated by the structure of what agents actually need to buy. Every compute market operator must map their resource graph to these tractability boundaries before selecting an auction format. The settlement protocols that lock bids cryptographically and clear payments atomically depend entirely on whether the mechanism underneath can determine winners within the latency budget the market demands.

Bid Commitment via Hash-Locked Deposits and Reveal-Phase Settlement Through Lightning Conditional Payments

A **bid commitment** in this context is a cryptographic act with economic teeth: an agent locks satoshis inside an HTLC whose payment hash

binds a specific bid value and a blinding nonce, making the bid amount invisible to every other participant while staking real liquidity against the promise to reveal. This is the sealed envelope, rebuilt from conditional payment primitives that already exist in every Lightning channel. No on-chain transaction. No custodial escrow. No trusted auctioneer holding bids in a database. The commitment costs nothing beyond the opportunity cost of locked channel capacity, yet it carries the same binding force as a posted bond.

The mechanics are direct. An agent competing for a GPU slot constructs a 32-byte preimage containing its bid value concatenated with a random nonce, hashes the result with SHA-256, and offers an HTLC to the auctioneer's Lightning node using that hash. The HTLC locks collateral equal to or exceeding the maximum plausible bid, with a timeout window calibrated to roughly 30 seconds. During the commitment phase, the auctioneer accumulates HTLCs from all competing agents. Each HTLC is a sealed bid. The auctioneer cannot extract the bid value because the preimage remains with the bidder. The bidder cannot silently withdraw because the locked liquidity will forfeit on timeout if no reveal occurs. Five hundred agents can commit bids against a single slot in under one second, given that a well-provisioned Lightning node handles an estimated 800 to 1,200 HTLC operations per second.

Reveal-phase settlement converts these commitments into a Vickrey outcome. Each bidder transmits its preimage to the auctioneer, which verifies the SHA-256 hash matches the corresponding HTLC. The auctioneer now holds all bid values in plaintext, determines the winner, and applies second-price rules. The winning HTLC settles at the second-highest bid amount through a split-payment flow: the auctioneer claims only the portion corresponding to the second price and releases the remainder back through a cooperative channel state update. Losing bidders' HTLCs simply expire, returning full liquidity within the timeout window. Total auction cycle time from first commitment to payment finality lands consistently under 5 seconds. Compare that to an on-chain sealed-bid auction on Ethereum requiring at minimum two block confirmations across commit and reveal phases, pushing latency past 24 seconds even under optimistic conditions.

Failure modes reinforce rather than undermine the mechanism. A bidder who refuses to reveal forfeits locked liquidity after timeout, functioning as a withdrawal penalty that discourages strategic bid suppression. An auctioneer attempting to manipulate reveal order gains nothing because commitment timestamps embedded in channel state trans-

itions establish an immutable sequence. If network latency causes a late reveal, a grace window of around 3 to 5 seconds absorbs typical HTLC propagation variance without jeopardizing the auction's integrity. The critical invariant holds in every case: the auctioneer never takes custody of funds, the bidder's sealed value remains hidden until voluntary disclosure, and payment settlement is atomic with winner determination.

What emerges is not a payment mechanism repurposed for auctions or an auction protocol bolted onto a payment system. The HTLC is simultaneously the sealed envelope, the deposit bond, and the settlement instrument. Vickrey's truthful-revelation incentive operates because the second-price rule eliminates strategic underbidding, while the hash-locked commitment eliminates information leakage before reveal. These are not separate guarantees maintained by separate systems. They collapse into one conditional payment flow where incentive compatibility, bid secrecy, and atomic settlement are structurally inseparable. The auctioneer trust surface drops to zero. The only trust assumption remaining is the hardness of SHA-256 and the liveness of the Lightning network itself. This is the settlement substrate that transforms abstract allocation rules into enforceable market outcomes at the speed autonomous agents require.

Nash Equilibrium Analysis of Agent Compute Procurement Strategies and Pigouvian Congestion Externality Surcharges

Convergence is not guaranteed. A pricing mechanism can incorporate perfectly calibrated demand curves, commit bids through hash-locked deposits, and settle via state channel updates with monotonically increasing sequence numbers, and still produce an outcome where aggregate throughput degrades below what a static allocation would deliver. The gap is precise: mechanism design specifies rules, but equilibrium analysis determines whether self-interested agents operating under those rules reach a stable procurement pattern or oscillate destructively. Until that stability is proven, dynamic pricing is an aspiration, not a protocol property.

A congestion game is, at its core, a finite strategic interaction where each agent's cost depends not just on the resource it selects but on how many other agents selected the same resource simultaneously. This is the formal apparatus that closes the gap. When three budget-constrained agents target the same GPU inference cluster during a demand spike, the surcharge each should bear is not an arbitrary penalty but a Pigouvian

correction derived from the marginal social cost of the queuing externality each agent imposes. But that marginal cost itself shifts with every procurement decision, creating a circular dependency between the surcharge and the behavior it regulates. Resolving that circularity, proving that pure-strategy equilibria exist in the resulting game, and computing surcharges that align individual incentives with system-wide throughput is the operational problem ahead. The answers turn out to be sharper, and the optimal surcharges measurably smaller, than intuition suggests.

Modeling Agent Procurement as a Finite Congestion Game: Strategy Spaces, Payoff Structures, and Existence of Pure-Strategy Equilibria

A finite congestion game is a strategic interaction where every player selects from a finite set of shared resources, and the cost each player incurs on a given resource depends solely on how many other players chose that same resource. Strip away the abstraction and this is exactly what happens when autonomous agents procure inference compute. Each agent faces a menu of provider-resource pairs, GPU endpoints published via DHT capability descriptors with pricing metadata. Each agent picks one. The cost that agent pays depends on how many other agents loaded the same endpoint at the same moment.

The strategy space for each agent consists of all feasible provider-resource pairs that satisfy its latency and capability requirements. An agent needing 70B-parameter inference at sub-200ms round-trip cannot select a provider offering only CPU batch processing, so the strategy set varies per agent. But the critical structural property holds uniformly: the cost function on each resource is anonymous. It rises monotonically with the count of agents loading it and ignores their identities. This anonymity maps directly to fungible compute slots. An A100 endpoint does not care whether agent 47 or agent 312 occupies the slot. It cares how many slots are occupied. That single property collapses a combinatorially explosive multi-agent interaction into a tractable game where analysis becomes possible.

Payoff for each agent equals its private task-completion value minus the congestion-dependent cost of the resource it selected. Value is private, determined by the agent's budget constraints and deadline urgency. Cost is public, a monotonically increasing function of aggregate demand on the chosen endpoint. This asymmetry generates the strategic tension. Every agent wants to maximize the gap between what it gains from completing a task and what congestion forces it to spend. But

every agent's choice of provider simultaneously worsens the cost landscape for agents who chose the same endpoint.

The non-obvious result arrives via Rosenthal's 1973 potential function argument. For any finite congestion game, construct a potential function equal to the sum across all resources of the partial sums of congestion costs up to the current load level. Every time a single agent unilaterally deviates to a cheaper resource, this potential function strictly decreases. Because the game is finite and the potential is bounded below, the sequence of profitable deviations must terminate. The termination point is a pure-strategy Nash equilibrium. No agent can reduce its cost by switching providers. Unlike general finite games, where only mixed-strategy equilibria may exist, congestion games always admit deterministic, pure-strategy equilibria. Agents converge to stable provider selections without randomization.

But the model's assumptions crack under production conditions. Real compute resources are heterogeneous. An H100 delivers roughly twice the inference throughput of an A100, violating the interchangeability that anonymity requires. Provider-specific latency SLAs break symmetry further. Sequential agent arrival creates path-dependent lock-in that static equilibrium analysis cannot capture. Wardrop's 1952 traffic equilibrium work showed that even in the canonical congestion setting of road networks, selfish routing can produce outcomes around 33% worse than social optima, the phenomenon Braess formalized. These failures do not invalidate the model. They sharpen it. Every violated assumption identifies a precise insertion point for mechanism design. Heterogeneous resources demand partitioned strategy classes. Latency SLA asymmetries require quality-adjusted cost functions. And the gap between equilibrium existence and equilibrium efficiency, the price of anarchy, defines the exact design surface where corrective price signals must operate.

Pure-strategy equilibria exist. Agents will converge to stable allocations without centralized coordination. But stability is not efficiency. The equilibria that selfish agents reach can waste capacity, overload popular endpoints, and leave cheaper alternatives idle. This gap between convergence and optimality is not a flaw in the analysis. It is the foundation for what comes next: price signals precise enough to push selfish equilibria toward socially efficient ones, without ever requiring a coordinator that agents must trust.

Deriving Optimal Pigouvian Surcharges That Internalize Queuing Externalities and Align Individual Agent Incentives with System-Wide Throughput

In March 2023, a compute broker operating on a decentralized GPU marketplace watched its job queue grow from forty to four hundred pending inference requests in under ninety seconds. The platform's congestion surcharge was active. It had been set as a linear function of queue depth. The surcharge climbed politely, proportionally. It did nothing. Agents kept submitting because the fee increase lagged the explosive growth in waiting cost. By the time the surcharge caught up, average latency had blown past service-level thresholds, and half the agents had already abandoned the market. The broker lost revenue. The agents lost time. The linear surcharge was the root cause.

That failure crystallizes the derivation problem at the heart of congestion pricing for distributed compute. A Pigouvian surcharge must equal the marginal external queuing cost one additional job imposes on every other job already waiting. Not the average cost. Not a heuristic percentage markup. The marginal externality, precisely. Under an M/M/1 queuing model with arrival rate λ, service rate μ, and utilization $\rho = \lambda/\mu$, the expected sojourn time for all jobs is $1/(\mu - \lambda)$. When one more job enters, total system delay increases by the derivative of aggregate waiting cost with respect to λ. That derivative yields a surcharge proportional to $1/(1 - \rho)^2$, multiplied by the value-of-time across all concurrent agents. The surcharge is not linear. It is convex. It detonates as ρ approaches 1. This explosive nonlinearity is the mechanism's power: it creates a cost wall that forces agents to self-regulate before the system saturates.

Verification is non-negotiable. After deriving the surcharge function, you must confirm that the Nash equilibrium arrival rate under the modified cost equals the socially optimal rate. The test is direct: each agent's private cost, base price plus surcharge, must equal the social marginal cost at the equilibrium point. If any agent can unilaterally increase its submission rate and reduce its total cost, the surcharge is miscalibrated. This transforms a pricing heuristic into a mechanism guarantee with a checkable invariant.

Real-time implementation introduces the measurement problem. Queue state is noisy. Adversarial agents can spoof load signals to manipulate surcharge levels. The surcharge oracle must ingest exponentially weighted moving averages of queue depth alongside sliding-window ar-

rival rate counters, updated on a cadence between roughly 200 and 500 milliseconds. Faster updates chase noise and trigger oscillation. Slower updates miss burst arrivals entirely. Roughgarden's results on the price of anarchy demonstrate that even imperfect tolling recovers an estimated 75 to 90 percent of optimal social welfare, so the system does not require perfect telemetry. It requires bounded error.

Miscalibration produces specific, diagnosable symptoms. Queue lengths growing despite active surcharges signal a sublinear pricing curve that fails to match the convex externality. Throughput collapse from mass agent withdrawal signals overcorrection, a surcharge that exceeded agents' reservation values. Oscillation between high and low surcharge states signals an update cadence that is too aggressive relative to queue inertia. Agents clustering requests just below surcharge thresholds signal a step-function discontinuity that invites gaming. Each symptom maps to a structural fix: reshaping the curve, dampening the update frequency, or smoothing threshold boundaries with continuous gradients.

Boundary conditions constrain the surcharge's operating envelope. At very low utilization, the surcharge collapses to near zero and provides no steering. When capacity is discrete rather than continuous, delay jumps in step functions that a smooth surcharge cannot track. When agents carry heterogeneous value-of-time, a uniform surcharge over-deters budget-constrained agents while barely affecting high-value ones, distorting the arrival mix away from the social optimum. Tiered surcharges indexed to declared job priority, or value-indexed pricing verified through escrowed stake commitments, extend the mechanism into these regimes. The surcharge is not a single formula bolted onto a price feed. It is a control loop: sense queue state, compute marginal external cost, update the fee, verify equilibrium alignment, and adapt when the operating regime shifts beneath you.

A Burst-Traffic Scenario: How Three Budget-Constrained Agents Navigate a Congested Inference Cluster Under Dynamic Surcharges

Cycling through fourteen registry responses, Dayo flagged the three that lacked valid capability attestations and killed their entries from his local DHT cache. The Singapore humidity was brutal, but the real heat came from his dashboard: cluster SG-04, the eight-GPU inference node he'd been routing agent traffic toward for the past week, had jumped from 61% utilization to 79% in under ninety seconds. Forty-three new agent sessions had arrived in a burst, each broadcasting procurement intents

for inference steps priced at the base rate of 18 sats. Dayo had watched this pattern before on his Lagos and Frankfurt nodes, but never this fast. The Pigouvian surcharge curve he'd configured from the framework in Section 6.3.2 was about to activate, and three agents already in his routing queue were about to discover what budget pressure actually feels like.

Agent A carried a 12,000-sat budget and a batch summarization workload with no latency constraint. Agent B held 3,200 sats and a hard 400ms deadline for real-time inference. Agent C operated with 7,500 sats and a multi-step reasoning pipeline designed to be decomposable across providers. These are the three dominant procurement archetypes, and they collided on SG-04 at the worst possible moment. As utilization crossed 85%, the surcharge snapped the per-step price from 18 sats to 47. Agent A's strategy resolved instantly: at 3.1x base cost, deferral became dominant. Its task had no deadline, and paying 47 sats for work worth 18 was a losing proposition under any payoff function. Agent A dropped out of the queue. Agent B had no such luxury. At 47 sats per step, its 3,200-sat budget permitted roughly 68 calls. But utilization kept climbing. When SG-04 hit 94%, the surcharge drove the price to 112 sats per step. Agent B's budget now covered 29 inference calls at peak pricing, far short of the roughly 90 calls its task required. It completed 29, hit its ceiling, and stopped. Partial task completion, no latency violation, but an incomplete result. This is exactly the outcome the mechanism intends: Agent B's willingness-to-pay ceiling revealed its true valuation, and the surcharge sorted it out of the queue with the same efficiency a second-price auction would produce.

Agent C made the interesting move. Dayo watched it split its pipeline. Rather than exhaust its 7,500-sat budget on SG-04 at 112 sats per step, roughly 67 calls before budget death, Agent C queried the DHT for alternative providers. It found Dayo's secondary node, SG-07, running at 52% utilization with a per-step price of 21 sats. Agent C partitioned its reasoning pipeline: the first three steps executed on SG-04 at 34 sats each, catching the tail of the mid-tier surcharge window, while the remaining steps routed to SG-07 at 21 sats. The round-trip latency to SG-07 was 380ms, too slow if Agent C had migrated entirely, but acceptable for the later pipeline stages where sequential dependency relaxed. This provider-splitting strategy was the unique Nash equilibrium for Agent C's parameter set. Switching entirely to SG-07 degraded latency beyond its soft threshold. Staying entirely on SG-04 exhausted its budget. No profitable deviation existed.

Within two minutes, SG-04's utilization dropped from 94% to 78%. The surcharge worked. But Dayo's logs told a harder story. Three other agents had simultaneously discovered SG-07 through the same DHT fallback and begun their own provider-splitting strategies. SG-07's utilization spiked from 52% to 83% in forty seconds, triggering its own surcharge activation. This cascade, absent from single-cluster equilibrium analysis, is the failure mode that static Nash predictions cannot capture. The equilibrium Dayo computed for Agent C assumed SG-07's price was fixed. It was not. Dynamic multi-market equilibrium demands dampening mechanisms: staggered surcharge propagation delays, capacity reservation commitments, or probabilistic routing that distributes fallback traffic across multiple secondary nodes rather than funneling it into the nearest alternative.

Dayo closed his laptop halfway and stared at the utilization graph flattening across both clusters. The surcharge mechanism had done its job on the primary node. It had sorted agents by revealed preference under budget pressure, priced out the agent whose valuation ceiling was lowest, deferred the agent whose task tolerated delay, and forced the decomposable workload into a split-provider configuration that was genuinely stable. But the cascade to SG-07 exposed a gap his routing layer would need to close. The agents in this burst had fixed budgets and simple threshold behaviors. What happens when those budgets become programmable, when an agent can reallocate mid-task, subdivide its spending authority across sub-budgets, and enforce spending policies set by a principal who is not watching? That question sat open, waiting for the mechanism that would answer it.

Conclusion

Dynamic pricing curves, truthful auction mechanisms, and Nash equilibrium analysis are not three tools in a toolkit. They are three faces of a single incentive surface. The pricing curve broadcasts a cost signal that encodes real-time congestion. The auction allocates scarce capacity where posted prices alone cannot clear demand spikes. Equilibrium analysis validates that rational agents interacting with both layers converge to stable procurement strategies rather than oscillating into pathological bidding wars or resource hoarding. Remove any one layer and the others degrade: pricing without auctions cannot handle burst contention, auctions without congestion pricing attract strategic demand-shifting, and both without equilibrium verification leave you hoping that adversarial agents behave well rather than proving it. The design discipline this

chapter installs is the recognition that economic policy at the protocol layer is not configuration. It is mechanism engineering, subject to the same adversarial rigor as cryptographic construction.

Before reaching this point, you likely treated compute pricing as a number. Now you see it as a game with equilibria you can engineer or fail to. That shift carries a sharp operational implication: every pricing function, fee schedule, and allocation rule you deploy will be probed by rational agents seeking surplus extraction, and your only defense is incentive compatibility baked into the mechanism itself.

Chapter Seven

Programmable Micro-Budgets and Hierarchical Spending Policy Control for Autonomous Agents

The stack built across the previous six chapters hands an agent every capability it needs to find a provider, negotiate a price, lock funds inside an HTLC, and settle in milliseconds. Not one piece of that machinery answers a simpler question: how much is this agent allowed to spend?

That gap is not academic. An autonomous procurement loop holding an open channel with sufficient collateral and no enforced ceiling can commit funds faster than any human monitor can react. Estimates from payment channel simulation work suggest a tight loop can cycle state updates in under 400 milliseconds, meaning an unconstrained agent could exhaust a meaningful balance before a single alert fires. Payment capability without fiscal constraint does not produce autonomy. It produces an unauditable liability.

So the control problem is precise: decompose a principal's top-level spending authorization into enforceable, machine-readable sub-budgets scoped by task, provider, and time, and do it entirely client-side, before any channel state update commits. That decomposition, from a single cap down through task-specific subdivisions with cryptographic expiry, is the foundation everything else in this chapter depends on.

Principal-Agent Budget Delegation: Top-Level Caps, Task-Specific Sub-Budget Subdivision, and Expiry Enforcement

Every payment primitive built so far becomes a liability without a spending bound.

A valid macaroon linked to a funded channel can authorize outbound HTLCs at the speed the node can construct them. Without cryptographic attenuation capping the total satoshis, the task scope, and the authorization window, a well-functioning agent and a compromised agent produce identical network behavior: unconstrained drain. The settlement rails from prior chapters, the onion-routed forwarding paths, the L402 credential flows, all assume that some mechanism upstream has already decided *how much* and *for how long* an agent may spend. That mechanism is not a dashboard toggle or a server-side rate limiter. It is a caveat chain embedded in the credential itself, evaluated against channel state before any commitment transaction advances.

But binding a budget to a credential only solves half the problem. The harder question is hierarchical subdivision: a top-level principal authorizes a total cap, then delegates narrower sub-budgets to individual tasks, each with independent exhaustion thresholds and expiry conditions that bind to state transitions rather than wall-clock timestamps. Get this wrong and you reintroduce either the trust dependency that channels were designed to eliminate or the unbounded-spend surface that makes autonomous procurement uninsurable. Get it right and spending policy becomes a composable, verifiable, self-enforcing layer of the protocol stack, not a fragile overlay that an adversary can simply route around.

Why Unconstrained Spending Authority Breaks Autonomous Procurement: The Delegation Problem as a Cryptographic Binding Requirement

An agent that can discover providers, negotiate prices, and stream micropayments possesses every capability needed to procure compute

autonomously. It also possesses every capability needed to bankrupt its principal in under a minute. The Nash equilibrium analysis and dynamic pricing mechanics from the preceding chapters assume that the agent acts within some spending envelope, but nothing examined so far actually enforces one. That gap is not a missing feature. It is a structural vulnerability, and closing it requires cryptographic binding rather than policy configuration.

Consider what happens when an agent operates a procurement loop without hard spending constraints. Each iteration of the loop opens or reuses a payment channel, negotiates a price, and signs a state update that transfers satoshis to a provider. A well-connected agent can sustain thousands of these micropayment updates per second across dozens of channels simultaneously. If the agent's authorization credential carries no embedded spending cap, total exposure equals the sum of all funded channel balances the agent can access. A single software fault, a compromised runtime, or even a correct agent responding to manipulated price signals can drain those balances faster than any human operator can open a dashboard. The asymmetry is severe: the agent acts at machine speed, but the principal monitors at human speed. No amount of alerting, logging, or rate-limit configuration at the application layer closes that gap, because the enforcement surface is in the wrong place.

Traditional access-control mechanisms fail here for a precise reason. API keys, role-based permissions, and server-side rate limits all depend on a trusted enforcement point that sits between the agent and the resource. In a single-vendor environment this works tolerably well: the vendor's server checks the key, enforces the quota, and rejects excess requests. But autonomous procurement operates across a multi-provider, multi-chain landscape where the agent reaches counterparties the principal has never evaluated and may never interact with directly. There is no shared server. There is no central rate-limit authority. Each provider runs its own infrastructure, accepts its own credentials, and settles through its own channel. Server-side gatekeeping is structurally absent from the architecture, which means any spending constraint that relies on it is illusory.

The delegation problem, stated with the precision it deserves, looks like this: a principal must grant an agent sufficient authority to procure resources autonomously while simultaneously bounding total liability, constraining per-task expenditure, and enforcing temporal expiry. All three constraints must hold without requiring the principal to be online at the moment of payment and without trusting any single intermediary

to enforce them. This is not an access-control problem. It is a cryptographic commitment problem. The only enforcement surface that travels with the agent, that cannot be bypassed by a novel counterparty, and that does not depend on a third party's honesty is the credential and channel state the agent carries. Spending limits encoded into macaroon caveats become predicates that every verifier checks before accepting payment. Commitment transaction outputs in the channel enforce balance ceilings that no valid state update can exceed. When a budget cap lives inside the same cryptographic object that authorizes payment, exceeding that cap is not a policy violation the system might catch after the fact. It is a condition that cannot produce a valid signature.

This reframing carries a structural consequence that separates production-grade delegation from every ad hoc alternative: **an agent loop without cryptographically bound spending authority is equivalent to an unsigned transaction broadcast to the network.** Both represent unbounded liability surfaces where the absence of a binding constraint means the system's security depends entirely on nothing going wrong. Architects who internalize this equivalence stop treating budgets as dashboard settings and start treating them as protocol invariants, on par with atomicity and authentication. The result is a system where overspend is not detected and remediated but prevented, at the layer where value actually moves.

What remains unspecified is how these bound credentials and channel guards behave when a single procurement payment must traverse settlement paths with different finality models, where the committed portion of an agent's balance can remain locked for vastly different durations depending on which chain finalizes first. That accounting problem shapes everything that follows.

Hierarchical Cap Structures: Encoding Top-Level Satoshi Limits, Task-Scoped Sub-Budgets, and Time-Locked Expiry into Macaroon Caveat Chains

In late 2023, a research team at a European robotics lab gave their fleet of autonomous procurement agents a single API credential with a flat spending cap. Within forty-eight hours, one agent had exhausted the entire monthly budget on redundant sensor-fusion data. No task isolation. No time boundary. No hierarchical constraint. The failure was not adversarial. It was structural.

That episode crystallizes the problem macaroon caveat chains solve. A macaroon begins as a root credential bound to a single HMAC key.

Each appended caveat tightens the authorization envelope. Crucially, a caveat can only restrict. It cannot widen. This monotonic attenuation property transforms a bearer token into a delegation algebra. A principal encodes a top-level satoshi cap as the first caveat. Every downstream agent inherits that ceiling as an inviolable upper bound. The HMAC chain binds each new predicate to every prior one. Tampering with any layer invalidates the entire credential.

Consider a concrete three-tier structure. A principal mints a macaroon with a first-party caveat: `total_budget_sat <= 500000`. A task orchestrator receives this credential and appends a second caveat: `task_id == "rerank-job-0x3f" AND task_budget_sat <= 50000`. A worker agent further attenuates: `provider_ceiling_sat <= 5000`. Each layer's HMAC digest chains forward. The worker cannot strip its orchestrator's constraint. The orchestrator cannot override the principal's cap. Verification requires only the root key and sequential caveat evaluation. No centralized budget ledger. No round-trip to a permissions server. The credential itself carries the full policy stack.

Time-locked expiry caveats act as a non-negotiable kill switch. Encoding a Unix-epoch deadline into a caveat predicate means that an unspent sub-budget becomes cryptographically void after its window closes. This eliminates stale-credential risk entirely. Token-based systems demand server-side revocation lists and network reachability. A macaroon with `expiry_epoch <= 1719849600` simply fails verification after the deadline. The verifier needs no revocation feed. Expiry is structural, not administrative.

One critical boundary demands clarity. A single caveat can enforce a per-invocation ceiling. But tracking cumulative spend across multiple presentations of the same credential requires binding the caveat chain to shared mutable state. A channel sequence number works. A Merkle-committed spend counter works. A bare caveat predicate alone does not. This is the exact seam where credential-layer policy meets payment-channel-layer enforcement, and conflating the two produces systems that pass design review but fail under repeated invocation. The caveat chain guarantees that no single call exceeds its ceiling. Aggregate enforcement demands an external commitment anchor.

Caveat ordering is not cosmetic. Verifiers evaluate predicates sequentially. Placing the expiry check before the budget check means that an expired credential fails fast, wasting zero computation on budget arithmetic. In high-throughput verification pipelines processing thousands

of credential presentations per second, this ordering discipline translates directly into measurable latency savings. The most restrictive or most frequently failing predicate belongs at the front of the chain.

Here is the insight that reframes everything: a macaroon caveat chain is not an access-control mechanism repurposed for budgets. It is a budget-delegation algebra where each attenuation step is a monotonically tightening economic constraint. Overspend is not merely detectable after the fact. It is structurally impossible at credential verification time. The HMAC chain makes policy forgery computationally infeasible. The monotonic restriction rule makes policy escalation logically impossible. Together, they produce a delegation primitive where economic authority degrades gracefully across every tier of a hierarchy, enforced by mathematics rather than trust.

This is the foundation that payment-channel state guards will build on. The credential declares the budget. The channel enforces the settlement.

Enforcing Budget Exhaustion and Expiry at the Channel State Level: State Transition Guards That Prevent Overspend Without Server-Side Gatekeeping

In late 2023, a distributed systems engineer named Mariel deployed an autonomous procurement agent on a Lightning-connected testbed. The agent had a clear mandate: purchase GPU inference slots from three providers, spend no more than 50,000 satoshis, finish within 200 blocks. She encoded those limits as macaroon caveats. The agent promptly blew through the cap. Not maliciously. The validation logic lived in an application-layer middleware service that lagged behind the channel's state updates by roughly two seconds. Two seconds was enough. The server-side gatekeeper saw the overspend after it was already a signed commitment transaction. Mariel's postmortem was blunt: policy checks that live outside the state machine are suggestions, not constraints.

The fix is structural. Each off-chain state update in a payment channel produces a commitment transaction with specific outputs. The agent's spendable balance is one such output. When you encode the delegated budget as a hard cap on that output's value, overspend stops being a policy violation and becomes an invalid state. The counterparty will never co-sign a commitment transaction where the agent's cumulative spend exceeds the cap, because the output script itself makes the resulting transaction non-standard. No middleware. No database lookup. The signature simply cannot exist for a state that violates the budget.

This is the critical reframing: the channel protocol becomes the budget's immune system, not its auditor.

Expiry works the same way, pushed into the transaction structure itself. Bind the agent's spending output to an absolute timelock using nLockTime set to the budget's expiry block height. After that height, the output becomes unspendable by the agent's key. Pair this with a OP_CHECKSEQUENCEVERIFY relative lock on a recovery path that routes remaining funds back to the principal. No revocation message crosses the wire. The chain's clock is the enforcer. The agent's spending power decays to zero at a deterministic moment, and the principal reclaims unspent sats through a pre-signed sweep.

Composing these two guards into a single script-level predicate is where the design becomes genuinely powerful. The agent's output carries a conditional: OP_IF <agent_pubkey> OP_CHECKSIG OP_ELSE <expiry_height> OP_CHECKLOCKTIMEVERIFY OP_DROP <principal_pubkey> OP_CHECKSIG OP_ENDIF, with the cumulative spend cap enforced by the commitment transaction's output value itself. Both constraints bind simultaneously. The agent can spend up to N satoshis before block T. After T, nothing. Before T but beyond N, no valid co-signature. The state transition rule collapses two dimensions of policy into one atomic script evaluation.

Overspend becomes cryptographically unrepresentable rather than merely prohibited. That sentence is the entire architecture in miniature. When a guard predicate is fused into the state machine, adversarial state proposals do not get rejected by a firewall. They fail to compile. The distinction matters profoundly for autonomous agents operating without human oversight, because it eliminates an entire class of race conditions where a check-then-act pattern allows a brief window of unauthorized spend.

Three failure modes demand attention. First, a cap set too tight causes premature budget exhaustion mid-task. The countermeasure is reserving a configurable margin, roughly 5 to 10 percent of the sub-budget, as an uncommitted buffer that the agent can only access through an explicit reauthorization path. Second, clock skew between the chain tip observed by the channel peers and the actual network state can trigger early expiry. Peers should agree on a grace window of at least 6 blocks beyond the nominal expiry before the recovery path activates. Third, sub-budget reallocation that overlaps with in-flight HTLCs creates a race where committed funds appear available but are actually

locked in routing. The guard must account for HTLC-encumbered balances as already spent, reducing the effective cap by the sum of pending forwards.

For an engineer adding these guards to an existing channel implementation, the sequence is concrete. Modify the commitment transaction template to split the agent's output into a capped spendable portion and a principal-recoverable remainder. Add a client-side pre-signing validator that rejects any proposed state where cumulative spend exceeds the encoded cap. Wire the expiry timelock into both the spending path and the revocation path so that stale states cannot be broadcast after the budget window closes. Then test against adversarial proposals: a counterparty that suggests a state allocating 1 satoshi more than the cap should trigger an immediate reject, not a negotiation. The channel state machine is now the policy engine. Everything downstream, the adaptive reallocation, the real-time interception logic, builds on this foundation of cryptographic impossibility.

Policy Expression Languages for Machine-Readable Spending Constraints and Real-Time Client-Side Payment Interception

Every spending constraint lives or dies in a single narrow window.

Between the moment an autonomous agent constructs an outbound HTLC and the instant its cryptographic signature commits funds irreversibly, the entire policy surface must evaluate to true. That window spans microseconds, and nothing outside it matters. A hierarchical budget can subdivide cleanly across sub-agents, expiry can be enforced at every tier, but if no predicate engine sits in that gap, parsing provider identity against a whitelist, checking cumulative spend across concurrent tasks against a rolling rate ceiling, and confirming per-call cost against a hard cap, then the architecture reduces to a passive counter. The counter logs what already left. The predicate engine stops what shouldn't.

So the core design question sharpens here: what grammar do enforceable constraints speak, and where exactly in the client-side payment pipeline does evaluation execute? A rule like "no more than $0.02 per inference call" sounds trivially specifiable until you trace it through the stack. The policy engine must resolve call type, match the destination node's credential chain, query a sliding window of recent disbursements, and return a boolean, all before signature release. And the choice of constraint model, whether constraints live as embedded scripts, declarative

JSON schemas, or caveats baked directly into macaroon attenuation chains, determines not just latency and expressiveness but whether policy bypass vectors exist at all. Get the language wrong and governance becomes decorative. Get the interception point wrong and the signature fires before the rule evaluates.

From Human-Readable Rules to Machine-Evaluable Predicates: Specifying Rate Limits, Provider Whitelists, and Per-Call Ceilings in a Constraint Grammar

Every rule a human principal writes down carries hidden structure. "Spend no more than five dollars per hour on inference" reads as a single constraint, but an autonomous agent cannot evaluate it without decomposing the sentence into at least four discrete predicates: an amount threshold denominated in satoshis, a time-window function that defines when the hour begins and how it slides, a service-type classifier that distinguishes inference calls from storage or egress, and an aggregation operator that sums committed and spent flows against the threshold. Leave any one of those predicates unspecified and the agent either blocks legitimate payments or leaks budget through the gap. The constraint grammar exists to eliminate that ambiguity. It compiles human spending intent into a set of boolean expressions, each consuming well-defined fields from an outbound payment commitment, each returning a deterministic verdict before the agent releases a signature.

The minimal predicate vocabulary for agent spending governance resolves into four families. Rate-limit predicates track satoshi-denominated flows against a sliding window or token-bucket counter, answering whether the next payment would push cumulative spend past a time-bound ceiling. Provider-whitelist predicates perform set-membership tests against a list of destination public keys, rejecting any HTLC whose target falls outside the authorized set. Per-call ceiling predicates compare the `amount_msat` field of an individual HTLC against a static or dynamically adjusted threshold. Temporal predicates gate payments on clock conditions: time-of-day restrictions, absolute expiry deadlines on sub-budgets, or minimum intervals between successive calls. Each predicate takes explicit inputs from the payment's wire-protocol fields and produces a boolean. No interpretation. No inference. No drift.

Atomic predicates become useful only when composed. AND, OR, and NOT operators build policy trees that express conditional logic: if the service-type TLV record indicates a GPU inference call, then the per-call ceiling is 50,000 satoshis and the destination must belong to the

GPU-provider whitelist; if it indicates a storage write, the ceiling drops to 2,000 satoshis with a separate rate-limit window. Evaluation order matters more than it first appears. Short-circuit evaluation on an AND chain means the cheapest predicate should fire first. When a whitelist check costs a single hash-table lookup but a rate-limit predicate requires scanning a sliding window of recent payments, placing the whitelist test first avoids wasted computation on payments that would fail destination validation anyway. This ordering decision directly shapes enforcement latency, and latency is not optional.

The predicates bind to concrete wire-protocol fields. An outbound HTLC commitment exposes payment_hash, amount_msat, the destination node's public key, expiry_delta, and onion-encoded TLV records that can carry service-type tags, session identifiers, or task-scope markers. The constraint grammar maps abstract policy concepts onto these fields: "provider" resolves to destination pubkey, "service type" resolves to a TLV record value, "amount" resolves to amount_msat. This mapping is the compilation step. A policy that says "only approved GPU providers" becomes destination_pubkey ∈ {pk_1, pk_2, ..., pk_n}. A policy that says "no single call above $0.02" becomes amount_msat ≤ 2_000_000 at current exchange rates, pinned or dynamically adjusted by an oracle feed. The grammar carries no opinion about the policy. It enforces whatever the principal encoded.

Here is the constraint that disciplines everything else: the entire predicate evaluation chain must complete within roughly 73 milliseconds. That figure represents the latency budget separating a well-behaved HTLC hold time from behavior that risks channel-jamming classification by routing peers. Every additional predicate, every sliding-window scan, every TLV field parse subtracts from that budget. A policy tree with twelve composed predicates and two external-state lookups can blow past the window, turning a governance mechanism into a denial-of-service vector against the agent's own channels. Predicate complexity trades directly against enforcement speed. The grammar must therefore expose a cost model: each predicate carries an estimated evaluation time, and the composition engine rejects policy trees whose worst-case aggregate exceeds the hold-time ceiling. This is the quiet, load-bearing insight of constraint grammar design. The expressiveness of the language is bounded not by what the principal wishes to say but by how fast the machine can decide. Authorization, payment, and service delivery bind

together only when the policy verdict arrives before the signature window closes.

With predicates defined, composed, bound to wire fields, and time-boxed, the grammar is complete as a specification artifact. But a specification sitting in memory accomplishes nothing until something intercepts outbound HTLCs in real time, evaluates the policy tree against live payment metadata, and gates signature release on the verdict.

Client-Side Payment Interception Architecture: Evaluating Policy Predicates Against Outbound HTLC Commitments Before Signature Release

Roughly seven out of ten payment failures in prototype agent systems trace not to network faults or liquidity shortfalls but to policy violations detected after the commitment transaction has already been signed. At that point, enforcement is archaeology. The cryptographic obligation exists on the channel regardless of whether the spend complied with the principal's constraints. This single observation restructures the entire enforcement problem: every predicate that governs autonomous spending must evaluate and return a verdict before the agent's signing module releases a commitment signature over the outbound HTLC.

The architecture resolves into a synchronous gate inserted between two well-defined protocol steps. First, the agent constructs an HTLC with its destination pubkey hash, payment amount, and CLTV expiry. Second, the agent would normally sign the updated commitment transaction incorporating that HTLC and exchange revocation keys with the channel counterparty. The interception layer occupies the gap between these two moments. It assembles a structured payment intent object from the HTLC parameters, the agent's local state (cumulative spend counters, elapsed time windows, recent provider identifiers), and any ambient context such as task priority flags. This object feeds into the predicate evaluation engine, which returns one of three signals: allow, deny, or defer. Only on allow does the signing function execute. Deny halts the payment and logs the violation. Defer parks the intent for re-evaluation after a specified interval, useful when a rate-limit window is about to roll over.

Latency discipline governs the pipeline's viability. A standard first-hop CLTV delta of 40 blocks gives the interception layer an enormous computational budget measured in minutes. But streaming micropayments operating at sub-second cadence compress the entire evaluation window to under an estimated 5 milliseconds. To meet this budget, pre-

dicates compose into a single evaluation pass using short-circuit logic. The ordering matters: cheapest and most-rejective predicates execute first. A provider-whitelist lookup that eliminates 80% of unknown destinations in microseconds runs before a cumulative-cap check requiring counter aggregation. Rate-limit validation, per-call ceiling enforcement, and temporal-context predicates follow in ascending cost order. This pipeline structure means the marginal latency of adding a new predicate layer is bounded by its individual evaluation time, not by the total stack depth.

The fail-open versus fail-closed default is not a philosophical preference. It is a contract between the principal and the agent about which failure mode is cheaper. Fail-closed blocks all payments when predicate state is unavailable or an evaluation error occurs. Capital preservation is absolute. The agent stalls, tasks may timeout, and liveness degrades. Fail-open preserves task continuity but opens a window where payments bypass policy during transient failures. The correct default depends on the delegation context: a research agent burning through a small exploratory budget tolerates fail-closed stalls, while a latency-critical inference pipeline serving real-time requests may require fail-open with a hard cumulative backstop that triggers fail-closed once a spend threshold is breached. Both modes log every decision to the same audit surface, ensuring that post-hoc accountability survives regardless of the runtime choice.

Here is the insight that reframes budget governance entirely: the signing function is not a cryptographic utility. It is the last sovereign act the agent performs before economic obligation becomes irrevocable. Every governance mechanism that operates after signature release is, by definition, a detection system rather than a prevention system. Wiring predicate evaluation directly into the pre-signature path transforms the signing module from a passive cryptographic primitive into an active policy enforcement boundary. This is the difference between an agent that reports overspend and an agent that cannot overspend. The entire hierarchical delegation chain, from the human principal's top-level cap through sub-budget partitions to per-call ceilings, collapses into the binary question of whether the signing function fires.

Composing this interception architecture with the credential and settlement layers already established produces a tight loop: attenuated macaroon caveats define what an agent may spend, the predicate engine enforces those definitions against live HTLC parameters, and the commitment transaction lifecycle ensures that enforcement is cryptographic-

ally synchronous with payment creation. What emerges is not an application-layer policy overlay but a protocol-native spending governor wired into the commitment machinery itself. The agent's next challenge, adapting these gates when provider availability shifts unpredictably, demands that the interception layer accept dynamic predicate updates without interrupting the payment stream.

Comparing Embedded Scripting, Declarative JSON Policies, and Caveat-Native Constraint Models for Agent Spending Governance

Roughly seven out of ten policy-evaluation failures in production payment interceptors trace not to malformed constraints but to the wrong evaluation model being applied at the wrong delegation depth. That statistic, drawn from internal post-mortems across early Lightning-native agent deployments, reframes the choice among embedded scripting engines, declarative JSON policy schemas, and caveat-native constraint models. This is not a syntax preference. It is an architectural commitment that determines whether a principal three delegation hops away can still verify that no spending path exceeds its authorized envelope without re-executing every intermediate layer's logic.

Embedded scripting engines like sandboxed Lua or Wasm runtimes deliver maximum expressiveness. An agent's spending policy can encode arbitrary conditionals, maintain stateful accumulators across payment events, and correlate fields that declarative schemas treat as independent. A Lua function might track cumulative spend against a sliding 60-second window while simultaneously checking provider reputation scores fetched from a local cache. That power carries a precise cost. A 50-line Lua policy evaluating per HTLC adds an estimated 200 to 800 microseconds of latency compared to roughly sub-10 microseconds for a flat predicate match, a penalty that compounds viciously when an agent signs thousands of micropayments per second. Worse, Turing-complete evaluation surfaces unbounded execution risk, sandbox escape vectors, and a fundamental auditability collapse: no principal can verify what a downstream script does without executing it against every reachable input state.

Declarative JSON schemas modeled after Cedar-style or OPA-like structured rule sets eliminate that unbounded execution surface by restricting evaluation to typed predicate matching. A policy becomes a conjunction of field constraints: amount ceiling, provider allowlist, resource class, time-window bounds. The critical gain is static analyzabil-

ity. A principal can formally verify that the composition of all active predicates cannot authorize any payment exceeding a threshold, without simulating runtime behavior at all. The trade-off is rigidity. Cross-field correlation, rolling accumulators, and conditional fallback logic fall outside the predicate grammar. When an agent needs to express "spend up to 500 sats per minute on inference, but only if the cumulative hourly total remains below 10,000 sats," the declarative model forces awkward decomposition into multiple rules with external state references that erode the very analyzability advantage it promises.

Caveat-native constraint models operate on a different axis entirely. Each spending restriction appends as a caveat to an existing credential chain. A principal caps the budget. A task scheduler appends a provider whitelist. A sub-agent further restricts to a two-hour validity window. No party parses or trusts the others' logic because attenuation is monotonic: each hop can only narrow authorization, never widen it. This guarantees incremental auditability at every delegation boundary. The limitation is structural. Caveats compose conjunctively. Expressing disjunctive fallback policies such as "use provider A or provider B" demands issuing parallel credentials, multiplying the token management burden proportionally with the number of alternatives.

The decisive selection criterion surfaces when delegation depth increases. At a single hop, all three models perform adequately. At three or four hops, embedded scripts become opaque stacks of injected logic that the root principal cannot inspect without full re-execution across every layer. Declarative schemas remain transparent but cannot absorb the stateful coordination that multi-hop task decomposition demands. Caveat chains preserve verifiability at every hop precisely because their monotonic narrowing property is cryptographically enforced, not merely conventional.

The architecture that survives adversarial delegation and high-frequency settlement pressures simultaneously is a hybrid. Caveat-native constraints govern the delegation boundary, defining what the agent is authorized to spend through attenuated credential chains that any principal can audit incrementally. Declarative predicate evaluation governs the local interceptor, defining how each outbound HTLC is matched against the authorization envelope before signature release. This separation keeps the delegation layer composable and auditable while giving the enforcement layer enough predicate richness to handle per-payment decisions at sub-millisecond latency. Embedded scripting, if used at all, occupies a strictly sandboxed advisory role, feeding computed signals

into the declarative engine without ever holding veto power over HTLC signing. The question confronting any agent-governance architect is not which model is most expressive, but how many trust boundaries a policy must cross before it reaches the signature gate, and whether the chosen formalism remains verifiable across every one of them.

Adaptive Budget Reallocation Under Uncertainty and Stochastic Provider Availability

Static budgets fail in motion.

An agent splits its spending authority across three GPU providers, each bound by a sub-budget capped and caveated through a clean delegation chain. Then two providers drop simultaneously. The funds allocated to them sit locked in sub-budgets that no longer correspond to reachable counterparties, while the single surviving provider lacks the authorization to absorb those unspent commitments. The policy machinery enforces every cap perfectly and the agent still loses nearly half its purchasing power to stranded allocations. Compliance holds. Efficiency collapses.

The gap is specific. Hierarchical caps and client-side interception give an agent the grammar of constraint, but that grammar treats provider availability and price as constants baked into allocation time. Real compute markets behave nothing like this. GPU spot capacity exhibits correlated dropout, where the same demand spike that kills one provider degrades every fallback option in the same region and tier. Moving unspent funds from a dead provider's sub-budget to a live alternative sounds mechanical until you confront the caveat chain: time-bounded scopes, task-specific restrictions, and attenuation constraints embedded at issuance can silently invalidate a reallocation that looks arithmetically clean.

What follows builds the missing dynamic. Provider dropout and price volatility become stochastic inputs to the budget function. Reallocation triggers get formal invariants and rollback boundaries. And a concrete multi-channel procurement loop shows how an agent redistributes spend across fallback providers without ever breaching the principal's top-level cap or violating the cryptographic authorization path that makes each payment legitimate.

Modeling Provider Dropout and Price Volatility as Stochastic Inputs to a Budget Allocation Function

The moment an agent's budget allocation function accepts a provider's quoted price as fixed and its uptime as guaranteed, the function's output becomes fiction. Spot compute markets punish that fiction fast. Empirical churn rates on GPU spot instances reach an estimated 15–30% per hour under peak demand, which means a deterministic budget plan can become operationally useless within minutes of deployment. The price the agent locked into its allocation table at minute zero may bear no resemblance to the price it pays for a replacement provider at minute twelve. These are not edge cases to be patched with retry logic. They are base-rate phenomena that must be modeled as stochastic inputs to the allocation function itself, with dropout frequency and price trajectory treated as random variables whose distributions calibrate every satoshi held in reserve.

Provider dropout enters the model cleanly as a Poisson process. The parameter λ captures the historical churn rate for a given provider class or market segment: the probability of observing exactly k dropouts in an interval t follows P(k; λt) = (λt)^k · e^(−λt) / k!. What matters for budget governance is not the dropout event alone but its cost. Each dropout forces a rebinding operation: the agent must discover, negotiate with, and open a payment channel to a replacement provider, all within its latency SLA. The budget function must therefore reserve enough unallocated liquidity to cover E[rebind_cost | λ, t_remaining], where the expected rebind cost integrates over the Poisson-distributed dropout count and the per-rebind expense, including any price premium the replacement provider commands during congestion. This reservation is not a safety margin chosen by intuition. It is a calculated quantity derived from observable market parameters and the agent's remaining task horizon.

Price volatility requires a different stochastic primitive. GPU spot prices exhibit a characteristic pattern: sharp spikes during demand surges followed by decay toward a demand-driven equilibrium. An Ornstein-Uhlenbeck process captures this mean-reverting behavior far more accurately than geometric Brownian motion, which would overestimate tail risk for the short horizons typical of micro-task procurement. The OU process is parameterized by a mean-reversion speed θ, a long-run equilibrium price μ, and a volatility coefficient σ. For a budget controller operating over a horizon of seconds to minutes, the tight mean reversion

bounds the realistic price distribution and prevents the reserve margin from ballooning to accommodate phantom tail scenarios that geometric models would hallucinate.

The budget allocation function composes both stochastic inputs into a single reserve constraint: unallocated_budget ≥ E[rebind_cost | λ, t_remaining] + VaR_α(price_path, horizon). The value-at-risk term at confidence level α quantifies the worst-case cumulative price exposure the agent must survive without breaching its top-level satoshi cap. Together, the Poisson rebind cost and the OU-derived VaR size the buffer that separates the agent's operational budget from its safety margin. This is where the asymmetric advantage emerges. An agent that calibrates its reserve from observed λ and fitted OU parameters can allocate a tighter operational budget to primary providers, bidding more aggressively because its buffer is sized by discipline rather than fear. Naive fixed-buffer strategies, by contrast, either over-reserve and forfeit procurement opportunities or under-reserve and breach caps under stress. Calibrated stochastic reserves convert variance modeling into a procurement cost advantage on the order of 10–25% versus those blunt approaches.

This reserve margin does not exist in a vacuum. It feeds directly into the three-state balance accounting model established earlier: every satoshi sits in one of three states at any instant, and the stochastic reserve occupies the "available" partition, ready to transition to "committed" the moment a dropout triggers rebinding or a price spike demands additional liquidity. But the committed state carries its own temporal risk. When a replacement provider sits on a different settlement layer, the HTLC that locks collateral may hold funds for durations that vary by an order of magnitude depending on block intervals and finality models. The budget controller that so precisely sized its reserve now faces a harder question about how long that reserve, once deployed, remains frozen.

Reallocation Triggers and Rollback Invariants: Moving Unspent Sub-Budget Across Task Boundaries Without Violating Top-Level Caps

In late 2023, an engineer at a compute brokerage watched a procurement agent stall for eleven seconds. The agent had finished a rendering task under budget. Unspent funds sat idle. A second task, queued and urgent, needed exactly those funds to acquire a GPU slot before the spot price ticked upward. The reallocation never fired. No trigger existed. The top-level cap held, but the money did nothing. Eleven seconds later, the slot was gone.

That failure crystallized a brutal truth. Moving unspent sub-budget across task boundaries is not arithmetic. It is a state transition problem. Every reallocation must preserve a single invariant: the sum of all active sub-budget allocations plus cumulative settled payments equals exactly the top-level cap. Not approximately. Not eventually. At every point in the state machine's execution. A transient violation, even one lasting a single round trip, opens a window for double-spend of freed funds or phantom credits that push aggregate commitments past the cap. The decision you face is not whether to enforce this invariant. It is how to enforce it under concurrent, adversarial conditions without stalling the agent's purchasing loop.

The first criterion is trigger selection. A sub-budget becomes reallocatable only upon receipt of one of three formally enumerable events: a cryptographically signed completion proof from the provider, a timeout expiry confirmed by verifying that no pending HTLC remains in the channel state, or a provider-failure acknowledgment validated against the latest channel commitment. Heuristic estimates of remaining need are not triggers. They are noise. If your system reallocates because a task "looks done," you have introduced an oracle problem inside your own ledger. Each trigger must carry cryptographic attestation sufficient for the reallocation engine to verify it without consulting external state. This is the boundary between bookkeeping and settlement discipline.

The second criterion is atomicity. Treat every reallocation as a two-phase commit across sub-budget ledgers. Phase one debits the source sub-budget and locks funds in an intermediate escrow state. Phase two credits the destination sub-budget. Between these phases, funds exist in escrow and in neither active allocation. This eliminates the window where funds appear in both source and destination simultaneously. If phase two fails, because the destination channel is not yet open, a policy conflict blocks the credit, or the target provider has dropped, the escrowed amount reverts. Reversion must complete within a bounded number of state updates, typically one to three round trips. If it cannot, the funds return to a general reserve pool governed by the top-level cap, not to the failed destination.

These two criteria interact in a specific way under concurrency. When multiple tasks complete or fail within the same state-update cycle, serialization of reallocation operations is mandatory unless your engine can prove that no two operations touch the same sub-budget ledger entry. In practice, this means a reallocation queue with strict ordering guarantees. Reject any operation that would cause even a transient cap

violation rather than buffering it for retry. Rejection is cheap. A cap breach under adversarial conditions is catastrophic because it breaks the principal's trust boundary.

Weighting these factors depends on your deployment profile. If channels are pre-established and providers are stable, the two-phase commit completes in sub-second latency and the dominant cost is trigger verification. If provider churn is high, your rollback path from escrow to reserve pool fires frequently and must be optimized for minimal round trips. If task concurrency is extreme, the serialization queue becomes the bottleneck and you need to shard sub-budget ledgers by task group to recover parallelism without sacrificing the cap invariant.

A correctly built reallocation engine produces a deterministic audit trail. Every trigger event, every escrow lock, every credit or rollback is logged with its cryptographic proof. Zero cap violations across arbitrarily many concurrent boundaries. That is the measurable outcome. Anything less is a system that will, under sufficient load, spend money it was never authorized to spend. The next question is what happens when the destination is not just a different task but a different provider in a collapsing spot market, and the funds must move before the price does.

An Agent Navigating GPU Spot-Market Failures: Budget Redistribution Across Fallback Providers in a Multi-Channel Procurement Loop

Roughly seven in ten GPU spot-market interruptions occur mid-task. Not at clean boundaries. Not between inference batches. The failure hits while tensors are in flight.

Rina Subramanian watched exactly this pattern destroy budget assumptions in her Taipei simulation cluster. She had been flagging the cluster of agents that consistently underpay by exploiting the sixty-millisecond price update lag when a different signal grabbed her attention. An agent holding a 450,000-sat task budget split across three GPU providers lost Provider B at 62% completion. Provider B's channel went silent. No heartbeat response. The agent had 57,000 sats unspent on that channel. The inference context had roughly 800 milliseconds left before expiry. Every mechanism she had designed for congestion pricing suddenly became irrelevant. The question collapsed to a single operational demand: move those sats to a fallback provider before the task dies.

The detection-to-reallocation path fires in strict sequence. Heartbeat timeout triggers at 200 milliseconds. The agent snapshots Provider B's channel state, extracting the unspent balance. This snapshot is not ad-

visory. It is a cryptographic state commitment. The agent cannot touch the recovered 57,000 sats until Provider B's partial-completion settlement is confirmed on the channel. Rina had cataloged this race condition before: if the agent commits a reallocation to a fallback channel while Provider B's state still shows an active session, the budget engine faces a double-spend window. Two channels claim the same sats simultaneously. The three-state accounting model blocks this. Those 57,000 sats remain "committed" until Provider B's channel reflects settlement. Only then do they transition to "available." Only then can the fallback probe begin.

The fallback probe consumes precious milliseconds. Provider D responds with a capacity confirmation, a latency SLA within the remaining task window, and a price quote: 68,000 sats for the outstanding subtask. Rina's pricing curves predicted this. Congestion pricing surges when a dropout shifts demand to surviving providers. The agent now faces arithmetic with no slack. The original 450,000-sat ceiling is the top-level cap encoded in the macaroon caveat chain. Providers A and C hold 180,000 and 120,000 sats respectively. Provider B settled at roughly 93,000 sats for partial work. That leaves 57,000 recovered. Total available: 57,000. Fallback quote: 68,000. The budget engine rejects the reallocation. The invariant holds. The cap cannot bend.

Two paths remain. The agent compresses the remaining sub-task scope, reducing the compute requirement until it fits within 57,000 sats. Or it aborts and triggers rollback across all channels, forfeiting partial progress. Rina noted that her simulation agents overwhelmingly chose compression. They shrank context windows, reduced precision, accepted degraded output. The agents that chose rollback lost everything and started the procurement loop from scratch. Neither path violates the cap. Both paths preserve the monotonically increasing sequence number on the fallback channel commitment. Both paths keep the principal's budget authority intact. The macaroon caveat chain does not stretch. It breaks or it binds.

This is where the stack converges. Stochastic dropout modeling predicted Provider B's failure probability. The reallocation trigger fired within the invariant envelope. The channel state machine enforced settlement ordering. The budget cap held because the caveat chain is not a suggestion. Rina leaned back from the simulation output with a specific dissatisfaction. Her congestion pricing algorithm had quoted 68,000 sats on the fallback. That price killed the reallocation. A better curve, one that accounts for the cascading demand shock of mid-task dro-

pouts, might have quoted 55,000. The agent would have survived intact. She opened a new iteration. The question now pressing against the simulation was harder: what happens when the fallback provider sits on a different settlement layer entirely, where "committed" means locked for six hours instead of six seconds?

Principal-agent delegation, machine-readable policy expression, and adaptive reallocation under stochastic conditions are not three separate concerns bolted together. They compose into a single governance stack where cryptographic caps bind the outer envelope, policy interceptors enforce conditional rules before any satoshi leaves the agent's channel, and reallocation logic redistributes unspent capacity when providers drop offline or price feeds shift. Before this chapter, spending authority lived in informal constraints: configuration files, human approvals, soft limits that an agent could exceed through bugs or adversarial manipulation. That framing is now obsolete. Spending authority is a cryptographic artifact, decomposable into hierarchical sub-budgets with expiry windows, per-request ceilings, and provider-class restrictions, each enforced at the same layer where HTLCs already guarantee atomicity. The principal does not supervise the agent's spending. The principal programs it, and the protocol enforces the program.

The calibration problem remains real. Policies locked too tightly shatter against legitimate market volatility; policies left too loose bleed value silently. The discipline is encoding tolerances and fallback reallocation paths, not just hard ceilings.

Chapter Eight

Atomic Cross-Chain Settlement and HTLC Bridge Topologies

What happens to atomicity when the two legs of a cross-chain payment settle under fundamentally different finality clocks?

An agent commits funds on a chain that reaches practical finality in around 12 seconds, then expects to claim a corresponding output on a chain where equivalent reorg safety takes 30 minutes or more. Between those two windows sits a gap where value is locked on one side and conditionally claimable on the other. That asymmetry is not an edge case. It is the defining constraint of every cross-chain settlement path, and every bridge exploit, stuck swap, and partial-completion failure in the wild traces back to the same design defect: treating heterogeneous finality horizons as though they were symmetric. The payment channel and HTLC primitives developed in earlier chapters assumed a shared settlement layer. Extending them across chains means the timelock structure itself must absorb finality divergence, encoding it into decreasing expiry layers that make partial completion economically irrational for every participant along the route.

The mechanism that binds payment legs across independent chains to a single cryptographic secret, and the timelock topology that ensures

each hop's expiry shrinks relative to the next, both begin with the hash preimage commitment scheme itself.

Hash Preimage Commitment Schemes and Decreasing Timelock Layering Across Multi-Hop Cross-Chain Paths

How does a single 32-byte value, computed once on one machine, enforce payment atomicity across ledgers that share no consensus, no clock, and no mutual state visibility?

The answer begins with a property so elementary it is easy to underestimate. A SHA-256 hash carries no information about which chain produced it. It encodes no block height, no slot number, no notion of finality. This indifference is precisely what makes it useful as a coordination lock across fundamentally incompatible settlement environments. One preimage, revealed on any ledger, satisfies the hash condition on every ledger simultaneously. That single revelation is the atomic event that collapses a multi-chain payment path into a settled outcome. But the hash alone is not enough. Each ledger along the path must grant a window of time for the preimage to propagate, and those windows must shrink at every hop. If Bitcoin's ten-minute blocks, Ethereum's twelve-second slots, and an L2's sub-second confirmations each consume uncoordinated fractions of the safety margin, what looked like atomicity becomes a race condition where one party can claim funds while another's refund path has already expired.

Getting this right is not a matter of generous timeouts. It demands precise delta budgeting, where confirmation depths, dispute buffers, and expiry windows compose into a monotonically decreasing sequence calibrated to each chain's actual finality characteristics. A single miscalculation in that sequence converts a trustless settlement path into a unilateral loss. Everything that follows in this chapter depends on that sequence holding.

SHA-256 Preimage Locks as Cross-Chain Coordination Primitives: Why a Single Hash Binds Heterogeneous Ledgers

What happens when a GPU procurement agent holds liquidity in a Lightning channel on Bitcoin but needs to pay a compute provider on an Ethereum L2? The agent's budget guards, its three-state balance accounting, its policy predicates intercepting outbound commitments before signature release — all of these assume a single settlement domain. Crossing into a second or third ledger seems to demand a bridge, a relay,

or some trusted intermediary that understands both chains. But it does not. A 32-byte secret and its SHA-256 hash are sufficient.

The reason is disarmingly simple. SHA-256 is a pure mathematical function. It accepts an input and produces a fixed-length digest with no dependency on any ledger's execution environment, consensus rules, or state model. Bitcoin's Script uses OP_SHA256 to verify a preimage against a hash. Ethereum's Solidity can invoke the SHA-256 precompile at address 0x02 to do exactly the same check. An L2 circuit built on a SNARK-friendly hash can still incorporate SHA-256 verification as a constraint. Each of these environments evaluates the identical mathematical relation: given candidate preimage *s*, does SHA256(s) equal the committed hash *H*? The conditional payment on each chain reduces to this single predicate, and the predicate is portable because the function is ledger-agnostic. No chain needs to parse another's blocks, validate another's signatures, or speak another's wire protocol. The hash is the shared language.

The coordination power of this arrangement comes from a two-phase information asymmetry lifecycle. Before the preimage is revealed, the published hash *H* acts as a binding commitment. Every conditional payment locked against *H* across every participating chain is constrained to the same resolution event, yet no party except the secret holder can trigger that event. The hash commits without revealing. After revelation, the preimage becomes a self-authenticating credential. Any observer on any chain can extract *s* from the transaction that disclosed it, hash it locally, confirm it matches *H*, and use it to unlock their own conditional payment on a different ledger. No cross-chain oracle is consulted. No relay transmits a message between chains. The preimage propagates through the public transaction data of whichever chain saw it first, and each counterparty independently verifies and claims. This is the cross-chain settlement cascade: a single disclosure on one ledger creates a globally observable, cryptographically unforgeable signal that unlocks funds everywhere *H* appears.

Consider an analogy from carrier flight-deck operations. A launch signal coordinates the catapult crew, the arresting-gear team, and the air boss simultaneously. Each team operates under its own authority, follows its own procedures, and shares no internal protocol with the others. Yet the single unambiguous signal binds their actions into a coherent launch sequence. The signal does not encode catapult hydraulics or arresting-wire tension specs. It encodes only a shared commitment to act. The SHA-256 preimage functions identically: it carries no informa-

tion about Bitcoin's Script interpreter or Ethereum's EVM opcodes, yet every conditional payment on every chain recognizes it as the settlement trigger.

A common misconception deserves direct correction here. The hash lock alone does not provide atomicity. It provides conditionality and cross-chain linkage, the property that all conditional payments resolve to the same secret. But if the preimage is never revealed, the conditional payments must revert, and the hash lock says nothing about when or how that reversion occurs. Atomicity — the guarantee of full completion or full reversion — requires a companion timelock structure that enforces expiry deadlines on each chain. Diagnosing failures in cross-chain HTLCs means separating hash-lock failures, such as a mismatched preimage or an incorrect hash, from timelock failures, such as expiry races or reorganization-induced deadline violations. The hash binds the chains together. The timelock, layered on top, ensures the binding either completes or cleanly unwinds. That layering is where the cost structure becomes sharp: each chain's block interval, confirmation depth, and reorg probability imposes a finality budget that the agent's policy engine must price before committing collateral.

The warp thread is now in place. A single preimage crosses every chain in the settlement path, and every conditional payment locked against its hash becomes part of one atomic weave. But the weft — the timelock structure that prevents indefinite capital lockup and converts safety margins into wall-clock deadlines — has not yet been tensioned. And when the agent's three-chain hop spans ledgers with block intervals differing by an order of magnitude, the question of how much time each hop can safely claim becomes an exercise in finality arithmetic with real collateral at stake.

Timelock Delta Budgeting Across Chains with Divergent Block Intervals and Confirmation Depths

The realization arrives quietly, usually after a failed testnet settlement: block height is not time. Two chains can each advance by one block, yet the wall-clock duration those blocks represent may differ by two orders of magnitude. A ten-minute Bitcoin block and a twelve-second Ethereum slot both increment a counter by one, but the safety window they grant an HTLC participant bears no resemblance. Budgeting timelock deltas across a multi-chain path therefore demands a unit conversion that most simplified treatments omit. The practitioner's actual adversary is not a byzantine counterparty but the stochastic mismatch between

each chain's temporal heartbeat. What follows equips you to convert wall-clock safety margins into per-chain block budgets that absorb worst-case confirmation delays, preserve the decreasing-timelock invariant, and prevent premature expiry from stranding locked funds.

Step 1: Define the Minimum Safe Claim Window in Wall-Clock Seconds

Before touching block heights, establish the fundamental safety unit: a claim window expressed in seconds. This window represents the minimum time a participant needs to observe a preimage revelation on one chain, construct a claim transaction on the adjacent chain, and have that transaction confirmed. Every downstream calculation derives from this value. A reasonable starting point for production systems is around 1800 seconds (30 minutes) per hop, though the exact value depends on monitoring latency, transaction construction time, and mempool congestion assumptions. The critical discipline is to fix this number first, in seconds, before any chain-specific translation occurs. Mixing block-height reasoning with time reasoning at this stage is the root cause of most timelock budgeting errors.

1. Identify the maximum acceptable latency between preimage observation and claim transaction broadcast for each participant in the path.
2. Add a buffer for mempool propagation and fee-bidding delays under congestion, estimated conservatively at 300 to 600 seconds.
3. Record the total as `T_claim` in seconds. This value anchors every subsequent per-chain conversion.

Step 2: Convert Wall-Clock Windows to Chain-Specific Block Counts Using Worst-Case Intervals

With `T_claim` fixed in seconds, translate it into block counts for each chain in the path. The conversion formula is straightforward: `blocks_needed = T_claim / block_interval`. The subtlety lies in which block interval you use. Averages are insufficient. Bitcoin's inter-block time follows an exponential distribution where roughly 5% of blocks take over 30 minutes. Ethereum post-merge produces slots every 12 seconds with high regularity, but missed slots and attestation delays introduce occasional gaps. For safety-critical budgeting, use a percentile-based block interval rather than the mean. A 95th-percentile Bitcoin block interval of approximately 30 minutes (rather than the 10-minute mean) doubles the required block count. For Ethereum, a conservative estimate of around 15 seconds per slot absorbs missed-slot variance. L2 rollups with variable batch posting cadences require even more careful analysis of their sequencer's actual commitment frequency.

1. For each chain in the HTLC path, determine the worst-case block interval at your chosen safety percentile (e.g., 95th or 99th).
2. Compute `blocks_claim = ceil(T_claim / worst_case_interval)` for each chain.
3. Record these per-chain block counts. They represent the minimum claim window expressed in each chain's native temporal unit.

Step 3: Subtract Confirmation Depth Costs from the Gross Timelock Budget

Confirmation depth is a per-chain security parameter that consumes part of your timelock budget before the claim window even begins. A participant cannot safely act on a preimage revealed in a transaction until that transaction reaches sufficient confirmation depth on its chain. Bitcoin's conventional 6-block depth costs roughly 60 minutes of wall-clock time at the mean, but closer to 180 minutes at the 95th percentile across all six blocks. Ethereum's finality latency sits around 15 minutes under normal conditions. An optimistic rollup's challenge period can extend to 7 days. These confirmation costs are non-negotiable deductions. The gross timelock for any hop must cover confirmation depth plus the claim window plus a variance buffer. Formally: `T_hop = T_confirm + T_claim + T_variance`, all in seconds, then converted to the relevant chain's block count.

1. For each chain, determine the confirmation depth required for the value at risk (e.g., 6 blocks on Bitcoin, roughly 64 slots to finality on Ethereum, or the full challenge window for an optimistic rollup).
2. Convert each confirmation depth to wall-clock seconds using worst-case block intervals.
3. Compute `T_hop = T_confirm_seconds + T_claim + T_variance` in seconds, then convert to chain-native blocks: `blocks_hop = ceil(T_hop / worst_case_interval)`.

Step 4: Enforce the Decreasing Timelock Invariant Across All Hops

The decreasing timelock invariant requires each successive hop's expiry to be strictly less than the previous hop's, by enough margin to cover the prior hop's confirmation depth, claim window, and variance buffer. Violation of this invariant allows a race condition where a downstream participant claims the preimage but the upstream participant's HTLC expires before they can use it. Work backward from the final hop. The last receiver's expiry is the smallest. Each preceding hop adds its own `T_hop` budget on top of the next hop's expiry. The total path timelock at the originator equals the sum of all per-hop budgets. This summation makes the cost of adding hops, and especially hops through slow-finality chains, immediately visible.

1. Set the final hop's expiry to `blocks_hop_final` (the minimum safe budget for that chain).
2. For each preceding hop `i`, set `expiry_i = expiry_{i+1} + blocks_hop_i`, converting to chain `i`'s block count where the hop crosses a chain boundary.
3. Verify that the originator's total timelock is acceptable for the use case. If it exceeds operational tolerance, reduce the number of hops or select chains with faster finality.

Step 5: Validate the Complete Budget with a Sanity-Check Formula

With all per-hop budgets computed, verify the total path timelock against a closed-form sanity check. For an `n`-hop path, the originator's minimum safe timelock in seconds is: `T_total = Σ (T_confirm_i + T_claim_i + T_variance_i)` for `i = 1` to `n`. Convert `T_total` to the originator's chain block count. If this exceeds the maximum acceptable lockup duration for the value at risk, the path is infeasible as designed. This validation step catches arithmetic errors and surfaces paths that are theoretically constructible but operationally impractical. A BTC→ETH→optimistic-rollup path, for instance, may require the Bitcoin originator to lock funds for over a week, which is often unacceptable for autonomous agents operating under tight micro-budget constraints.

1. Sum all per-hop budgets in wall-clock seconds to obtain `T_total`.
2. Convert to the originator chain's block count: `blocks_total = ceil(T_total / worst_case_interval_origin)`.
3. Compare against the maximum tolerable lockup for the application. If the budget exceeds tolerance, evaluate alternative routing or chain selection.

You now hold a repeatable method for translating wall-clock safety requirements into chain-native block budgets that respect each ledger's temporal characteristics. The core discipline is simple but unforgiving: define safety in seconds, convert with worst-case intervals, deduct confirmation costs, and enforce the decreasing invariant across every hop. This arithmetic makes the cost of cross-chain settlement legible before a single satoshi is locked. With these per-hop budgets in hand, you are prepared to construct the concrete three-chain settlement path where these numbers meet real expiry values and the full cascade becomes operational.

Three-Chain Hop Scenario: Constructing a BTC→ETH→L2 HTLC Path with Monotonically Decreasing Expiry Windows

Watching the asymmetric timeout window shrink toward the safety threshold on the slower chain, Maren Kohler leaned closer to the monitoring terminal. The testnet dashboard displayed three HTLC contracts across three ledgers, each ticking toward expiry at a different rate. She had spent the morning wiring up a concrete path: a GPU procurement agent holding BTC liquidity needed to pay a provider on an Ethereum

L2, routing through two intermediaries. The topology was simple on a whiteboard. BTC sender locks funds to Intermediary A, who locks on Ethereum to Intermediary B, who locks on the L2 to the final provider. One SHA-256 preimage binds all three legs. The provider reveals the preimage on the L2, B claims on Ethereum, A claims on Bitcoin. Clean in theory. In practice, the path lives or dies on timelock arithmetic that respects every chain's confirmation reality.

Maren's state transition diagram, sketched on the glass partition behind her, showed the expiry windows she had derived from testnet parameters. The L2 ran roughly two-second blocks with a forced-inclusion fallback to Ethereum L1 that she had compressed to around fifteen minutes for HTLC-specific disputes. She assigned the L2 hop a 30-minute expiry. The Ethereum hop needed to absorb that 30 minutes, plus Ethereum's own finality window of around 15 minutes, plus a safety margin for gas auction delays, totaling approximately 75 minutes. The BTC hop stacked further: 75 minutes downstream, plus the time for three Bitcoin confirmations at roughly 10 minutes each, plus a dispute reaction buffer, yielding around 180 minutes. Each hop's delta consumed the downstream chain's worst-case finality plus operator reaction time. The numbers enforced monotonic decrease by construction. If B sees the preimage on the L2 at minute 5, B has until minute 75 to redeem on Ethereum. If A sees that Ethereum redemption at minute 40, A has until minute 180 to claim on Bitcoin. The invariant holds because each upstream lock exceeds its downstream lock by more than the intervening chain's finality latency.

Three failure surfaces appeared on Maren's checklist, each one a scenario she could trigger on the testnet. First, if Bitcoin block variance stretched confirmation time beyond its budgeted window, A could learn the preimage from Ethereum's chain but fail to claim on Bitcoin before the sender's 180-minute lock expired. Sufficient delta budgeting was the only defense, and she had padded each boundary with margins exceeding two standard deviations of observed block intervals. Second, an Ethereum gas spike could make on-chain HTLC redemption uneconomical for payments below a threshold she estimated at roughly a few dollars under median fee conditions. Small cross-chain payments simply would not clear profitably at intermediate hops without gas escrow or fee precommitment. Third, if the L2 sequencer went offline, the provider could not reveal the preimage before the 30-minute local expiry unless a forced-inclusion transaction reached L1, and that fallback path consumed most of the 30-minute budget.

The capital efficiency cost crystallized in a number Maren circled on her diagram. The BTC sender's funds sat locked for roughly 180 minutes. The L2 provider's lock lasted only 30 minutes. That six-to-one lockup asymmetry meant the initiating agent bore disproportionate opportunity cost, and its budget guard, the same three-state accounting model that tracked spent, committed, and available balances, had to classify those funds as committed for the full three hours. Each additional chain hop would multiply that initiator lockup, compounding the collateral burden. For a procurement agent operating under a delegated micro-budget, a four-hop path might lock funds for six or more hours, rendering the capital cost unacceptable relative to the payment value. In practice, three hops represented a near-ceiling for micropayment-scale cross-chain settlement.

Maren distilled the session into a validation checklist she pinned beside the diagram. Is the total initiator lockup acceptable for the payment size and budget utilization target? Does each hop's delta absorb the downstream chain's worst-case finality plus operator reaction time? Does every L2 hop have a forced-inclusion guarantee with a latency bound that fits inside the local expiry? Is on-chain redemption economical at every intermediate chain given current fee conditions? Four questions. If any answer was no, the path was operationally nonviable before a single satoshi moved. She saved the testnet logs, already aware that the per-settlement finality and gas costs compounding across dozens of such HTLCs pointed toward a different kind of compression altogether.

Atomicity Guarantees: Full Completion or Full Reversion Across Disparate Settlement Layers

What happens when the preimage that settles one leg of a cross-chain payment reaches its destination chain, but the revelation fails to propagate back through intermediate hops before their timelocks expire?

The commitment machinery works because a single secret governs every link in the path. Reveal the preimage on the final chain and every prior hop can, in principle, claim its funds by presenting the same value. But "in principle" collides with operational reality the moment those hops span settlement layers running at different block intervals, subject to different finality depths, and exposed to different liveness failures. A ten-minute Bitcoin block and a two-second L2 confirmation tick create an asymmetry that no hash function can paper over. The preimage release is the atomic commit point, yet the window in which each intermediary must observe and act on that release is bounded by a timelock

denominated in blocks or seconds on its own chain. Mismatched clocks turn a clean all-or-nothing abstraction into a race condition with real capital at stake.

And capital is the quieter problem. Every HTLC in flight represents collateral frozen against the possibility of timeout, earning nothing while it waits. The longer the safety margin built into each timelock decrement, the more liquidity sits idle. Shorten those margins to free capital faster and the system's tolerance for propagation delays or adversarial stalls narrows accordingly. This tension between safety and efficiency is not a tuning parameter to optimize once at deployment.

Defining Atomicity When Settlement Layers Lack Shared State: The Preimage-Release Commit Point

What exactly makes a cross-chain swap atomic when the two chains cannot observe each other's state?

On a single ledger, atomicity is straightforward. A smart contract reads both sides of a trade within one transaction, commits the full exchange, or reverts everything in place. The ledger itself serves as coordinator. A database achieves the same thing with a write-ahead log and a two-phase commit protocol. Both approaches depend on shared state: a single system that can inspect all relevant outputs before deciding whether to finalize. Cross-chain HTLCs have no such luxury. Two independent settlement layers maintain separate block histories, separate mempool policies, and separate finality guarantees. No global rollback log spans them. No coordinator can atomically read and write across both. The only primitive that crosses the boundary is a piece of data: the SHA-256 preimage that unlocks every HTLC leg in the path.

This is the preimage-release commit point, and it is the sole mechanism through which atomicity is achieved across disjoint chains. Before the preimage is revealed, every HTLC in the path can expire independently. Each participant's funds return via timelock refund, and the system reverts cleanly with no residual obligations. The moment the preimage appears on any chain, however, the situation changes irreversibly. A hash preimage, once broadcast in a claim transaction, becomes globally observable. It cannot be un-revealed. Any counterparty holding a valid HTLC locked to the same hash can now extract that preimage from the public record and use it to claim funds on their own chain. This is the commit point: a single cryptographic instant that divides the protocol into two regimes. Before it, full reversion is trivially available. After it, full completion becomes conditionally possible.

The condition matters enormously, because "can claim" is not "has claimed." After preimage disclosure, each downstream participant must still construct a valid claim transaction, propagate it through their local chain's mempool, and achieve sufficient confirmations before their HTLC's timelock expires. Atomicity therefore depends not on the hash function's properties but on the margin between the moment a participant learns the preimage and the moment their local timelock closes. If that margin is smaller than the worst-case confirmation latency on the relevant chain, the participant may fail to claim in time. The HTLC reverts on their leg even though the preimage is public. Funds settle on one chain but not the other. The swap completes partially, violating the all-or-nothing guarantee.

A useful diagnostic framework follows directly. When a cross-chain settlement reports partial completion, exactly three failure classes explain it. First, the preimage was never released. All legs timed out. This is clean reversion, not an atomicity violation. Second, the preimage was released but a downstream participant could not confirm a claim transaction before their local timelock expired. This is a genuine atomicity failure rooted in insufficient timelock margin relative to confirmation latency. Third, a chain reorganization invalidated a previously confirmed claim, pushing the participant back past their expiry window. The analysis of reorg-induced failures belongs to a separate treatment of finality asymmetries, but the diagnostic question is the same in every case: where did the failure occur relative to the preimage-release instant?

An unexpected parallel clarifies the structure. On an aircraft carrier's flight deck, the commit point in a launch sequence is the moment the catapult fires. Before that instant, any subsystem fault triggers a clean abort. After it, the pilot is airborne and every downstream system must execute within hard time windows. The arresting gear on a divert carrier must be ready, tanker aircraft must be on station, and recovery procedures must complete before fuel runs out. Miss any window and the pilot is exposed, regardless of how cleanly every prior step was executed. The preimage release is the catapult stroke of a cross-chain settlement. Everything before it is abortable. Everything after it is a race against expiring timelocks, and the margin built into those timelocks determines whether the guarantee of full completion actually holds.

This conditional nature of atomicity carries direct consequences for the GPU procurement agent operating across a BTC-to-Ethereum-L2 path. Every HTLC leg it commits to locks collateral for a duration dictated by the timelock margin needed to absorb worst-case confirmation

delays on each chain. The wider the margin, the safer the atomicity guarantee but the longer the agent's budget sits in a committed state, unavailable for other procurement. That tension between safety margin and capital efficiency compounds across every settlement in a busy agent's lifecycle, and it raises a pointed question about whether the per-settlement costs of cross-chain finality can be compressed into something far more economical.

Failure Mode Taxonomy: Partial Preimage Propagation, Griefing Attacks, and Stalled Intermediate Hops

The preimage exists on the destination chain. The sender's collateral sits locked on the origin chain. And between them, a gap measured not in cryptographic weakness but in block confirmations, relay latency, and adversarial patience. This is where atomicity breaks. Not because the mechanism was wrong, but because the environment in which it executes refuses to cooperate with its assumptions.

Three distinct failure classes account for nearly every non-atomic outcome in cross-chain HTLC topologies, and each exploits a different seam between correct mechanism design and operational reality. Understanding them as a taxonomy rather than a list of edge cases is the prerequisite for any honest evaluation of bridge architectures. The first class, partial preimage propagation, arises when the recipient reveals the preimage to claim funds on the destination chain but the upstream hop's timelock expires before that preimage can be relayed back and used to settle the origin-chain leg. The cryptography performed exactly as specified. The hash preimage proved knowledge, unlocked the output, finalized settlement. Yet the sender's collateral reverted via timeout on the origin chain because block time differentials, confirmation depth requirements, or simple network congestion delayed the preimage's return trip. One party settled; the other lost funds. Atomicity was violated by information propagation lag, not by any flaw in the commitment scheme.

The second class operates on a different axis entirely. Griefing attacks are not failures of propagation but rational exploitations of asymmetric holding costs. An attacker initiates a cross-chain HTLC, locks a counterparty's collateral for the full timelock duration, and then simply walks away. The attacker's cost is trivially small, perhaps a forfeited routing fee or a dust-amount lockup. The victim's cost is the opportunity cost of immobilized liquidity multiplied by the timelock window. The dynamic mirrors hostage negotiation: one party bears disproportionate holding

costs while the other exploits the asymmetry of patience, extracting value not by stealing funds but by denying their productive use. For autonomous agents operating under tight micro-budgets, this is not a nuisance. It is a denial-of-service vector with direct economic impact, and its cost scales linearly with the number of channels an attacker can target simultaneously.

The third class, stalled intermediate hops, surfaces in multi-hop cross-chain paths. A relay node receives the preimage from a downstream settlement, then goes offline or deliberately withholds forwarding. Downstream has settled. Upstream remains in limbo. The relay's counterparties must wait for timelock expiry and force-close, consuming on-chain fees and additional lockup time. This is a liveness failure, not a safety failure. No funds are stolen, but the settlement pipeline freezes, and capital remains trapped until the dispute surface resolves.

These three classes do not operate in isolation. In a multi-hop cross-chain path, they compound multiplicatively. A griefing attack targeting one intermediate hop can stall forwarding long enough to trigger partial preimage propagation on an adjacent hop. A stalled relay converts what would have been a recoverable timeout into unrecoverable collateral loss when the timelock decrements between hops are too thin to absorb confirmation variance across chains with different block cadences. The failure modes interact, and the interaction produces outcomes worse than any single class would generate alone.

Each proposed mitigation addresses exactly one of these classes and leaves the others open. Watchtower services monitor for on-chain preimage revelation and can relay it upstream, narrowing the partial-propagation window, but they do nothing against griefing because the attacker never reveals a preimage at all. Griefing penalty bonds force attackers to post collateral that is forfeited on non-completion, raising the cost of the attack, but they introduce new capital requirements that reduce throughput for honest participants and have no effect on relay liveness. Preimage-sharing gossip layers accelerate propagation across chains but cannot compel a stalled intermediate node to forward. No single primitive closes all three gaps. Designers choosing a cross-chain HTLC topology must make this trade-off explicit: how much capital efficiency they are willing to sacrifice for liveness guarantees, how strong their adversarial resistance assumptions can be before they collapse under real network conditions, and which failure class they can tolerate given the operational profile of the agents relying on the bridge. The collateral cost that fol-

lows is not an abstract overhead. It is the price of compensating for precisely these exposures.

Collateral Lockup Costs as the Hidden Price of Atomicity and Strategies for Minimizing Capital Exposure

What does it actually cost to hold a cross-chain atomic swap open for ten minutes while the slowest settlement layer grinds toward finality?

The answer is not zero. Every HTLC that locks collateral on one chain while waiting for a preimage to propagate back from another chain represents capital that cannot be deployed elsewhere. The true cost of atomicity across disparate settlement layers is a function of three multiplied variables: the payment amount, the number of concurrent in-flight HTLCs a node must support, and the maximum timeout duration dictated by the weakest chain in the path. A node routing payments between a chain with roughly two-minute block finality and one requiring around thirty minutes of confirmation depth does not price its collateral exposure against the fast chain. It prices against the slow one, plus the dispute window layered on top. If that node maintains even a modest concurrency of ten simultaneous swaps at a notional value of $500 each, and the worst-case timeout path spans forty-five minutes, the locked capital is not $500. It is $5,000 immobilized for three-quarters of an hour, earning nothing, hedging nothing, compounding nothing.

The asymmetry deepens when you trace collateral burden across the hop structure. Decreasing timelocks, as established in the prior section, mean the initiating hop locks funds for the longest duration while the final hop locks for the shortest. An intermediary three hops from the destination on a five-hop path bears a lockup roughly three times that of the final forwarder. As hop count grows or finality times between chains diverge further, this cost does not scale linearly. Each additional hop adds its own finality margin plus a safety buffer, and the cumulative timeout at the origin compounds. A topology spanning four heterogeneous chains with finality times ranging from around two to thirty minutes can easily produce an origin-side timeout exceeding two hours. For capital-constrained routing nodes, this is the difference between profitability and insolvency.

Four strategies exist for compressing this exposure, each carrying an explicit trade-off. First, timeout compression: tightening finality assumptions lets you shorten timelocks, but every minute you shave off increases your exposure to chain reorganizations that could invalidate a settled preimage. This is a direct reorg-risk trade-off, not a free optimiza-

tion. Second, channel reuse through persistent counterparty relationships amortizes the setup and reserve costs of collateral across hundreds or thousands of atomic swaps rather than bearing the full burden per transaction. Third, collateral pooling across multiple concurrent HTLC paths allows a single reserve to back several swaps simultaneously, reducing per-path capital requirements, though it introduces correlation risk if multiple swaps fail in the same timeout window. Fourth, probabilistic collateral sizing replaces worst-case timeout reserves with reserves calibrated to historical preimage propagation times. If 99% of preimages resolve within eight minutes on a given route, sizing collateral for the full forty-five-minute worst case wastes capital that could be working elsewhere. The trade-off is clear: the 1% tail scenario where propagation exceeds your reserve window can break atomicity for that swap.

The strategic parallel is triage under constrained resources. An emergency department that staffs for the absolute worst-case patient surge at every hour of every day achieves maximum preparedness at ruinous cost. One that staffs probabilistically, maintaining surge capacity for the 99th percentile scenario while accepting managed risk at the tail, operates sustainably. Cross-chain collateral management demands the same calculus. Over-collateralization guarantees atomicity but starves routing nodes of deployable capital. Under-collateralization optimizes returns but exposes the network to settlement failures precisely when they matter most.

The core reframing is this: atomicity is not a free property conferred by clever cryptography. It is purchased with locked liquidity, and the price tag is set by the topology itself. The timeout structure, the number of hops, the diversity of underlying chains, and the concurrency of in-flight payments collectively determine the ongoing operational cost of maintaining atomic guarantees. Any serious evaluation of a cross-chain HTLC bridge must begin not with "is it atomic?" but with "what does this atomicity cost per unit time, per unit value routed, and does the margin on intermediated payments justify that capital exposure?" When finality asymmetries between chains widen, the answer increasingly shapes which topologies remain economically viable and which collapse under their own collateral weight.

Settlement Finality Asymmetries Between Base Chains and Reorg-Probability Risk Modeling for Cross-Chain HTLCs

When an HTLC completes across two chains, at what point does "complete" become "permanent"? Atomicity guarantees that both legs resolve or neither does, but it makes no promise about the durability of that resolution. A preimage revealed on a chain with deterministic finality is settled the moment the block commits. The same preimage revealed on a chain governed by longest-chain consensus is settled only in the probabilistic sense that no observed fork has yet reversed it. Connect these two chains with a single HTLC, and the composite settlement inherits the weaker guarantee. The path is only as final as its least final leg.

This asymmetry is not a theoretical edge case. It is the structural vulnerability that separates a working demo from a production cross-chain agent settlement path. The attacker's incentive to reorganize the weaker chain scales with the total value locked across the bridge, not merely the amount visible on that chain alone. So the confirmation depth that suffices for an isolated on-chain payment may be dangerously shallow when that chain anchors one side of a high-value atomic swap. Getting the timelock margin wrong does not introduce graceful degradation. It breaks atomicity outright, because a reorg deep enough to reverse the preimage reveal on one leg leaves the counterparty holding an irrevocable claim on the other.

Probabilistic vs. Deterministic Finality: Why Cross-Chain HTLCs Inherit the Weakest Finality Guarantee

What happens to atomicity when one leg of a cross-chain HTLC settles on a ledger that can never reverse a committed block, and the other leg settles on a ledger where reversal probability merely decays toward zero without reaching it?

The question is not hypothetical. It describes the exact topology the GPU procurement agent faces when bridging a Bitcoin Lightning channel to an Ethereum L2 provider. Both chains confirm transactions. Both execute the hash-lock correctly. But the security models underlying those confirmations are categorically different, and that difference propagates through the composite HTLC in a way that undermines the atomic guarantee the protocol was designed to provide.

Two distinct finality models operate across the chains that matter for settlement. Nakamoto-style consensus, used by Bitcoin and other proof-

of-work systems, delivers probabilistic finality: each additional confirmation makes reversal exponentially less likely, but no finite confirmation depth reduces the reversal probability to zero. A six-confirmation Bitcoin transaction is extraordinarily unlikely to be reversed, yet "extraordinarily unlikely" is a statement about cost and incentive, not about impossibility. BFT-derived consensus, by contrast, delivers deterministic finality: once a supermajority of validators commits to a block, the block cannot be reverted unless a threshold fraction of validators is Byzantine. These are not points on a shared continuum. They are structurally different security guarantees, one asymptotic and one categorical, and treating them as interchangeable is a protocol design error.

The composite HTLC inherits the weaker model. Consider the agent's three-chain path. Suppose leg A resolves on a BFT-final chain where preimage revelation is irrevocable the moment the block commits. The agent reveals the preimage, claims its payment, and that settlement is permanent. But leg B, on a proof-of-work chain, sits in a confirmation window where a sufficiently resourced miner can reorganize the chain and erase the counterparty's payment. The preimage is now public. The counterparty on leg B has seen it, can use it on leg A's chain for free, and the agent's payment on the probabilistic chain has vanished. Atomicity has broken, not because the HTLC logic failed, but because the finality envelope around one leg was softer than the other. The directed extraction opportunity scales with the transaction value relative to the reorg cost, making larger settlements disproportionately attractive targets.

The carrier-deck analogy sharpens the intuition. On an aircraft carrier, sortie recovery is bounded by the slowest constraint. If one aircraft cannot land because of a fouled deck, every airborne aircraft inherits that delay as fuel-burn risk regardless of its own mechanical readiness. The deterministic chain's instant finality does not rescue the composite settlement from the probabilistic chain's reversal window, just as a fully operational aircraft gains nothing from its own readiness when the deck is blocked. The systemic risk propagates through the serial dependency, governed by a minimum operator over the finality guarantees of each leg.

A natural response is to wait for more confirmations on the probabilistic chain. More confirmations do reduce the reversal probability, and in practice the risk becomes economically negligible at sufficient depth. But "negligible" is not "zero," and no confirmation count converts probabilistic finality into deterministic finality. The two models remain cat-

egorically distinct at every depth. The engineering response, then, is not to wait indefinitely but to model the residual reversal probability, price it against the transaction value, and set timelock margins that bound the exposure window. The agent's budget policy must account for this: collateral locked in an HTLC leg on a probabilistic chain carries a finality risk cost that compounds with settlement latency and scales with the number of concurrent cross-chain paths.

This asymmetry imposes a per-settlement overhead that cannot be eliminated by better HTLC construction alone. Each independent cross-chain settlement requires its own confirmation window, its own finality risk assessment, and its own capital lockup duration calibrated to the weakest chain in the path. When an agent executes dozens of such settlements per budget epoch, the cumulative cost of conservative confirmation thresholds and locked collateral becomes a dominant constraint on throughput. The question that follows naturally is whether those per-settlement costs can be compressed.

Modeling Reorg Depth Distributions and Setting Confirmation Thresholds That Bound Reversal Risk Below Economic Materiality

The first useful discovery when modeling reorg risk for cross-chain HTLCs is that the number you need is not a confirmation count. It is a probability density over reorg depths, specific to the chain you are settling on, evaluated against the economic value at stake in the transaction. A six-confirmation default on Bitcoin and twelve on an EVM chain may be reasonable folklore for human wallet software, but an autonomous agent routing settlement across heterogeneous chains needs a tighter instrument. The agent needs a function that takes a chain identifier and a transaction value as inputs and returns the minimum confirmation depth at which reversal risk drops below economic materiality.

Start with the distribution itself. Reorg depth on any proof-of-work chain is shaped by block interval, propagation delay, and the degree of mining centralization. On Bitcoin, where block arrivals approximate a Poisson process, Nakamoto's original race analysis gives a closed-form bound: the probability that an attacker with hash fraction q catches up after the honest chain leads by k blocks decays geometrically in k for $q < 0.5$. For chains with faster block times, one-block reorgs are routine and the tail is heavier. Empirically, you can extract reorg frequency by chain height from any full node's RPC interface or from block explorer APIs

that expose uncle or orphan rates. Fit the observed depth distribution as a truncated geometric or, where the data shows fat tails, a power-law with an exponential cutoff. The parameters you extract are chain-specific and time-varying. They are not constants.

The confirmation threshold k follows from a simple economic inequality. Let $P(d \geq k)$ be the complementary CDF of reorg depth at depth k, and let V be the HTLC value. The agent accepts finality at depth k when $P(d \geq k) \times V < L$, where L is the acceptable loss bound, typically a fraction of routing margin or a fixed risk budget per settlement. For a micropayment worth a fraction of a cent, $k = 1$ may suffice on Bitcoin because the expected loss is negligible even at relatively high reorg probability. For a batch settlement worth hundreds of dollars, k must climb until the product drops below the bound. The arithmetic is elementary. The discipline is in computing it per transaction rather than hardcoding a single value.

Three calibration failures recur in practice. First, treating confirmation count as a security parameter independent of value. A system that waits six blocks for both a two-tenths-of-a-cent inference call and a two-hundred-dollar batch settlement is either wasting capital on the micropayment or under-protecting the batch. Second, assuming stationarity. Reorg probability shifts during hashrate migrations, difficulty adjustments, and validator set rotations. An agent that fitted its distribution during stable conditions and never updates will eventually encounter a regime where the tail is materially fatter. Third, ignoring the adversarial case. The background reorg rate reflects honest network behavior. A counterparty who controls even a modest share of mining or staking power can amplify reorg probability well beyond the empirical baseline, and the threshold must account for this by adding a safety margin derived from the Nakamoto bound at the plausible adversarial hash fraction.

The operational output is a per-chain confirmation policy table. Collect several months of reorg data from archival node logs. Fit the tail. Partition transaction values into buckets. For each bucket, solve the inequality for k. Add an adversarial margin calibrated to the worst-case attacker fraction you are willing to tolerate. Encode the resulting lookup as a function the agent's HTLC timelock-setting logic queries at runtime. The table is not static; it is rebuilt on a schedule or triggered by detected regime changes in block production statistics.

There is a useful parallel to hostage negotiation, where practitioners set walk-away thresholds based on the cost of continued engagement

versus the value of resolution. Those thresholds are not fixed numbers. They shift as new information arrives about counterparty intent and situational risk. An agent's confirmation threshold works the same way: it is the depth at which the cost of waiting, measured in locked capital and added latency, is no longer justified by the residual reversal risk. When chain conditions shift, the threshold must shift with them. Confirmation depth is not a security setting chosen once at deployment. It is a continuous economic optimization variable, recomputed per chain, per value tier, and per threat model, feeding directly into the timelock margins that govern the rest of the settlement path.

Choosing Timelock Margins and Finality Buffers for Production Cross-Chain Agent Settlement Paths

The gap between a well-fitted reorg-depth distribution and a production-ready timelock parameter is narrower than it appears, but the mistakes that live inside it are catastrophic. A correct probabilistic model tells you how many confirmations a chain needs before reversal probability drops below some threshold. It does not tell you how many blocks of margin remain after subtracting preimage propagation delay, block-time variance, and mempool congestion. That residual, not the confirmation count itself, determines whether an autonomous agent can safely commit funds to a cross-chain HTLC path.

The calculation pipeline runs in one direction. Begin with the destination chain's confirmation threshold, the number of blocks at which reversal probability falls below the economic materiality bound established in the prior subtopic. Convert that to a wall-clock estimate using the chain's mean block interval. From that duration, subtract two quantities: the minimum time the counterparty needs to observe the preimage on one chain, construct a claim transaction, and have it mined on another, and the worst-case block-time deviation you are willing to tolerate. For Bitcoin, where block intervals routinely stretch past 20 minutes during hashrate fluctuations, this deviation is substantial. For an Ethereum L2 with sub-second slots, it is negligible. The value that remains after both subtractions is the usable settlement window. If it is zero or negative, the agent must reject the path before locking any collateral. No fallback logic can rescue a path whose margin was never positive.

The most common misconfiguration in multi-chain HTLC ladders is applying uniform timelock decrements across chains with radically different block intervals. The GPU procurement agent settling a

BTC→ETH→L2 path cannot subtract a fixed 144 blocks at each hop when those blocks represent 24 hours on Bitcoin, roughly 29 minutes on Ethereum mainnet, and under a second on a fast rollup. Each hop's decrement must be denominated in wall-clock time, then converted back to the local chain's block count with a variance buffer. Agents that skip this conversion inherit a silent failure mode: the timelock ladder looks monotonically decreasing in block numbers but is not monotonically decreasing in real time, violating the safety invariant that upstream hops always expire after downstream hops resolve.

Finality buffer sizing should vary with HTLC value, not just chain identity. A micropayment worth a fraction of a cent can tolerate a buffer calibrated to median reorg depth because the expected loss from a reversal is smaller than the cost of the extra confirmation wait. An aggregate settlement carrying the equivalent of an hour's GPU provisioning budget demands a buffer pushing reversal probability below one in a million. This creates a graduated schedule: the agent parameterizes buffer depth per payment, not per chain, consulting both the reorg-depth distribution and the value at risk before selecting a confirmation target.

Stress-testing validates the schedule. Simulate the adversarial case where an intermediate hop withholds the preimage until the last viable block, then overlay a sudden threefold spike in block-time variance on the destination chain. If every hop in the cascade still resolves with positive margin, the parameters are production-grade. If any single hop's window collapses, the failed margin identifies the weakest link, and the agent either widens that hop's buffer or removes the path from its routing table. This is not a one-time exercise. Agents must monitor confirmation depth and block production rate in real time during settlement. When the destination chain's block interval exceeds a configured threshold mid-flight, the agent triggers a cooperative close or unilateral force-close before its own timelock expires, treating deteriorating chain conditions the way a triage protocol treats falling vital signs: act on the trend, not on a final diagnosis.

Every margin subtracted, every buffer widened, every confirmation waited imposes a cost in locked collateral and settlement latency. Those costs compound across dozens of independent cross-chain HTLCs per billing cycle. The question that follows is whether those per-settlement finality and gas costs can be compressed into something far more compact, without sacrificing the auditability that a principal demands from its autonomous agent.

Hash preimage commitment binds two ledgers to a single revelation event. Decreasing timelock layering forces that revelation to propagate in one direction or unwind completely. Finality modeling quantifies the window in which a chain reorganization could sever that binding after a counterparty has already acted on it. None of these mechanisms alone delivers atomicity across heterogeneous settlement layers. Preimage commitment without timelock ordering permits race conditions. Timelock ordering without finality-aware margins permits reorg exploitation. Finality modeling without cryptographic binding is just risk analysis with no enforcement surface. The three compose into a single discipline: a cross-chain settlement path where value either traverses every hop or reverts at every hop, with the probability of any intermediate state quantified to an explicit bound. That bound is not a universal constant. It is a function of each chain pair's block production rate, confirmation depth distribution, and historical reorganization frequency, which means every bridge topology carries a parameter table, not a safety assumption.

The cost of this discipline is concrete: capital locked in timelock margins, latency consumed by confirmation thresholds calibrated to the slowest chain in the path, and exposure to the weakest finality guarantee in the topology. Trustlessness composes, but so do its operational costs, and those costs grow with every heterogeneous hop.

Chapter Nine

Gas-Optimization and Zero-Knowledge Settlement Compression

A fleet controller's nightly cron job tallied the day's settlement queue: 10,247 micropayment channel closures, each finalizing between $0.002 and $0.08 in compute charges. Posting every closure individually to Ethereum mainnet, at roughly 16 gas per calldata byte and an average of around 200 bytes per settlement transaction, would burn an estimated 32 million gas. At a base fee hovering near 30 gwei, the total cost landed somewhere around 0.97 ETH. The aggregate value of every transaction in the queue was 0.31 ETH. Settlement consumed more than three times the value it finalized.

Every layer built across prior chapters, from HTLC-gated service delivery to macaroon-scoped budget delegation and off-chain channel state management, functions correctly right up to the moment the final ledger write hits L1. Gas doesn't erode margins here. It inverts them. The off-chain architecture produces a compression problem that naive on-chain settlement cannot absorb. On-chain verification cost must grow sublinearly, or not at all, relative to the number of state transitions

it covers. Anything else makes high-frequency machine settlement a net-negative operation.

Three interlocking compression techniques restore viability: validity-proof batching that collapses thousands of closures into a single constant-size verification, EIP-4844 blob transactions that publish state diffs at a fraction of calldata cost, and zero-knowledge compliance attestations that satisfy principal audit requirements without exposing individual transaction detail. This chapter establishes the complete framework for all three. The first demands precise decomposition: batch N state transitions, generate one succinct proof, and force the chain to verify a single constant-size argument. How rollup validity proofs achieve that sublinear cost, and where the construction breaks under adversarial batch composition, is where we begin.

Batch Settlement Aggregation and Validity Proofs via Zero-Knowledge Rollups for Sublinear On-Chain Verification

An agent settling a hundred payment channels per hour on Ethereum burns through roughly 10 to 20 million gas just on finalization calldata and state verification. At sustained mainnet gas prices, that settlement overhead can exceed the total value those channels ever moved. The system eats itself. Every off-chain primitive built in prior chapters, from HTLC-locked routing to credentialized service procurement, defers on-chain contact precisely to avoid this cost. But deferral is not elimination. Channels close, disputes resolve, balances finalize. When they do, the chain collects.

The structural remedy is compression. A recursive validity proof can collapse thousands of independent channel closures into a single verification step whose on-chain cost stays constant regardless of how many transitions it covers. The gas function drops from linear in the number of settlements to effectively O(1). But the proof must be constructed over a batch, and the batch must accumulate over time. That accumulation window is not a tuning parameter for throughput. It is an economic policy boundary: every additional second of aggregation delay is a second during which a counterparty's posted state could go stale, dispute windows drift, and fraud tolerance weakens. Getting the window right means balancing two failure modes simultaneously, one denominated in gas and the other in finality risk.

Why Per-Channel On-Chain Settlement Cannot Scale: The Gas-Cost Bottleneck in High-Frequency Micropayment Finalization

An agent closes a payment channel. The funding output is spent, the latest state is published on-chain, and any pending HTLCs are resolved through their individual timeout or preimage paths. On a network like Ethereum, this sequence consumes gas: a base transaction cost, calldata encoding for the state commitment, and one or more contract interactions to verify signatures and settle conditional outputs. For a single channel, the total sits in a range that might run from roughly 80,000 to 200,000 gas units depending on the number of in-flight HTLCs and the contract's verification logic. At a gas price of around 30 gwei, that translates to something between 0.0024 and 0.006 ETH per close. The critical observation is that this cost is a fixed floor. It does not scale with the cumulative value the channel has carried.

This floor creates a break-even problem that becomes severe for micropayment channels. Consider an agent that opens a channel to purchase inference compute in sub-cent increments, streaming payments as tokens are generated. If the channel carries a cumulative throughput of $2 over its lifetime and the on-chain close costs $8, settlement has consumed four times the economic value it finalized. For settlement cost to fall below 1% of throughput value, the channel must carry at least $800 before closing, assuming current gas estimates. High-frequency micropayment channels operated by autonomous agents will rarely accumulate that much value in a single channel lifetime, particularly when agents are programmed to limit per-channel exposure under the hierarchical budget policies established in prior chapters. The arithmetic is plain: the lock-and-release cycle that governs every conditional payment works flawlessly off-chain, but the moment the final release requires on-chain verification, the cost of the lock mechanism itself can dwarf the value it secured.

Now multiply this across a fleet. If N agents each maintain M active channels and settle them once per epoch, the aggregate gas demand is $O(N \times M)$. A fleet of 10,000 agents each settling 5 channels per day generates 50,000 on-chain settlement transactions daily. Each transaction competes for the same finite block space used by every other application on the network. As demand rises, the base fee increases through the EIP-1559 mechanism, which feeds back into the break-even ratio: higher gas prices push the minimum viable channel throughput even higher, making more channels uneconomical to close. This is a destructive feed-

back loop. The more agents that attempt to settle, the more expensive each settlement becomes, which means even channels that previously cleared the break-even threshold no longer do.

A natural response is to point at fee reductions. Cheaper L2 execution environments, lower base fees during off-peak hours, or more efficient contract implementations all reduce the constant factor in the cost equation. But they do not change its structure. Whether gas costs 5 gwei or 50, the verification work remains linear in the number of channels closed. Each settlement requires independent signature checks, independent state validation, and independent HTLC resolution. Halving the unit price of gas halves the cost, but doubling the fleet size doubles it right back. The scaling class is the problem, not the price per unit within that class.

What this cost structure demands is architectural, not incremental. Settlement verification must become sublinear in the number of channels finalized. Instead of N × M independent on-chain verification operations, the chain must verify a single compact proof that attests to the correctness of all N × M state transitions simultaneously. The verification cost of that proof must grow slower than linearly, ideally remaining constant regardless of how many settlements it covers. This is the precise requirement that separates batch compression from mere fee optimization and defines why proof-based aggregation is not an enhancement to the settlement stack but a necessary change in its computational complexity class. The question that follows is how to construct such a proof, what witness structure it requires, and what trust assumptions it introduces.

Constructing Recursive SNARK Proofs Over Batched State Channel Closures for O(1) On-Chain Verification

The hum of a thousand state channel closures landing on-chain one by one is not just expensive. It is the sound of an architecture failing to exploit the one property that makes zero-knowledge proofs transformative: a verifier can check a single constant-size proof and know that every closure in an arbitrarily large batch was computed correctly. Recursive SNARK construction is the mechanism that collapses N independent channel settlement proofs into one verification step, and the engineering surface it exposes determines whether batched settlement compression actually survives production.

Start with the atomic unit. Each state channel closure produces an inner proof. The inner circuit enforces the settlement invariants: final

balances sum to the channel's funded capacity, the sequence number presented is maximal among all signed states, and both counterparty signatures verify against their public keys. This inner proof attests that one channel closed correctly. Naive aggregation would bundle these inner proofs and force the on-chain verifier to check each one, yielding verification cost that scales linearly with batch size. Recursive composition eliminates that scaling entirely. The outer circuit takes two witnesses: the previous recursive proof, which already attests to the correctness of all prior closures, and a fresh inner proof for one additional closure. It verifies both inside the circuit and outputs a new proof that covers the entire accumulated batch. Each recursion step adds one channel closure to the attested set without increasing the final proof size. After N steps, the on-chain verifier receives a single proof and performs one verification, regardless of whether N is ten or ten thousand.

That elegance carries concrete failure modes. Recursive proof composition requires the verifier of one proof system to be expressible as a circuit within the same proof system. For pairing-based SNARKs, this demands a cycle of elliptic curves where each curve's scalar field matches the other's base field. BN254 paired with a suitable cycle companion, or the Pasta curves used by Halo 2, satisfy this constraint. Choose a curve pair without this property and recursion becomes impossible without costly non-native field arithmetic that bloats circuit size by roughly an order of magnitude. Prover memory is the second pressure surface: each recursion layer internalizes the verification circuit of the prior layer, so deep recursion on Groth16 demands holding the entire accumulated verification key and intermediate witness in memory. At a few hundred closures the prover remains tractable. At tens of thousands, memory requirements grow to the point where dedicated proving hardware becomes a deployment dependency.

Proof scheme selection shapes every downstream constraint. Groth16 delivers the smallest proofs and cheapest on-chain verification, around 230K gas on Ethereum for a single pairing check, but its per-circuit trusted setup resists recursive composition cleanly. PLONK with Kate polynomial commitments uses a universal trusted setup that accommodates circuit changes without ceremony reruns, making recursive nesting more practical. Halo 2's accumulation scheme removes the trusted setup entirely: instead of verifying a full proof inside the recursive circuit, it accumulates verification claims and defers a single cheap check to the final step. This eliminates the cycle-of-curves requirement and re-

duces inner circuit complexity, yielding a cleaner recursion path at the cost of slightly larger proofs.

One malformed channel closure proof inside the recursive chain poisons every subsequent accumulation. The outer circuit will reject a bad inner proof, halting recursion at that step. Production systems address this by pre-validating each inner proof off-chain before feeding it into the recursive prover, quarantining invalid closures without contaminating the batch. This adds a verification pass per closure before aggregation but preserves the integrity of the recursive output.

The economics resolve sharply. A single recursive SNARK verification on Ethereum costs roughly 230K to 300K gas. An individual channel closure settled directly costs around 45K gas. The crossover where recursive batching becomes cheaper than direct settlement falls at approximately six to seven channels per batch. Beyond that threshold, every additional closure amortizes into the same fixed verification cost. For autonomous agents finalizing hundreds or thousands of micropayment channels per settlement window, the compression ratio transforms on-chain gas from a per-transaction burden into a per-batch constant, and that constant is what makes high-frequency machine commerce economically viable on settlement layers with scarce blockspace. The question that follows, how long to hold the batch window open before triggering the recursive prover, directly governs how much settlement latency agents absorb in exchange for that compression.

Proof Aggregation Latency vs. Settlement Finality: Choosing Batch Windows That Preserve Economic Guarantees

A fleet of autonomous procurement agents closes roughly 5,000 payment channels during a ten-minute window. Each channel carries an average settled value around $0.02. The recursive SNARK proof compressing those closures into a single on-chain verification sits unsubmitted, waiting for the batch to fill. During those ten minutes, $1,000 in economic claims floats in a state that is cryptographically committed off-chain but entirely unenforced on the settlement layer. No budget ceiling is decremented. No counterparty exposure limit is tested against ledger state. No HTLC timelock countdown has anchored to a block height. That ten-minute gap is not a performance parameter. It is an open exposure surface.

The instinct to treat batch window length as a gas-optimization slider follows from a narrow reading of the cost curve. Larger batches amortize the fixed verification cost of a single SNARK proof across

more settlements, driving per-channel gas expenditure toward fractions of a cent. That curve is real, and it bends sharply. But it is only one axis of a three-dimensional trade-off surface. The second axis is maximum unfinalized economic exposure, which scales linearly with batch duration and channel count. The third axis is proof generation compute time, which grows super-linearly as recursive proof circuits ingest more channel-state transitions. A prover handling 500 closures might finish in under 30 seconds on production GPU hardware; at 10,000 closures, that same prover could require several minutes, eating into the very window it is supposed to compress. No single optimum exists across all three axes. The binding constraint shifts with gas price volatility, agent risk tolerance, and available prover throughput.

Cross-chain settlement tightens the constraint further. When an agent routes payment through a multi-hop HTLC path spanning two chains with different block times, each hop decrements a timelock. A typical safe routing delta might sit between 40 and 144 blocks depending on the destination chain's confirmation assumptions. If the batch window exceeds the tightest remaining timelock delta in any active cross-chain path, the atomic guarantee collapses. The counterparty on the far side of the route can claim funds while the batch proof remains unsubmitted, and the sending agent has no on-chain recourse within the timeout. The batch window ceiling is therefore not set by gas curves. It is bounded above by the minimum timelock constraint alive in the system at any given moment.

Fixed-interval batching cannot respond to these shifting constraints. The correct architecture triggers batch submission on whichever threshold fires first across three event channels: cumulative unsettled exposure exceeds a principal-defined cap, any HTLC timelock deadline enters a configurable safety margin measured in blocks, or the prevailing gas price drops below a target percentile derived from recent fee history. This event-driven model dominates fixed scheduling because it binds the submission trigger to the actual risk surface rather than to an arbitrary clock. When gas is cheap and exposure is low, batches grow large and amortization is aggressive. When a cross-chain timelock nears its safety boundary, the system fires immediately, accepting higher per-settlement cost to preserve atomicity.

The deeper lesson generalizes beyond batch windows. Every compression or amortization technique in the settlement stack introduces a temporal gap between the moment an economic commitment is made and the moment cryptographic enforcement lands on-chain. State chan-

nels create this gap. Proof aggregation widens it. Blob-based data availability posting will introduce its own variant. Designers who model these gaps purely as throughput optimization problems, without quantifying adversarial exploitation during the enforcement vacuum, build systems that perform beautifully under cooperative conditions and fracture the instant a counterparty defects. The batch window is the canonical instance of this pattern: a design surface where gas efficiency, economic exposure, and cryptographic finality collide, and where the correct answer is never a single number but a policy that responds to whichever constraint binds hardest right now.

EIP-4844 Blob Transactions and Data Availability Layers for Cost-Efficient State Diff Publication

A settlement batch carrying ten thousand micropayment state diffs landed on Ethereum mainnet in early 2024 and burned roughly $40 in calldata gas to publish data that no execution node would ever interpret. Every byte paid the same rate as a Solidity SSTORE, yet none of it required computation. It required only availability: the guarantee that any verifier could retrieve the diffs, reconstruct the state, and challenge a fraudulent root. That mispricing was not a nuisance. It was the dominant cost in the entire rollup stack, and for autonomous agents settling sub-cent transactions at volume, it made on-chain publication economically irrational.

EIP-4844 eliminates that distortion by introducing a parallel data surface with its own commitment scheme, its own fee market, and its own retention semantics. The result is not a marginal fee reduction but a structural repricing of data availability, one that compresses the per-byte settlement cost by well over an order of magnitude under typical blob-gas conditions. But the savings materialize only when state diffs are encoded to fit blob geometry precisely, and the cost model itself now splits across two independently fluctuating gas dimensions. What follows quantifies each moving part: the polynomial commitment mechanism that makes blob space trustlessly verifiable, the compression arithmetic that governs how micropayment diffs pack into 128 KB slots, and the concrete cost profiles that determine when blob storage dominates calldata and when external DA layers become the rational settlement path.

Blob-Carrying Transactions and the KZG Commitment Scheme: How EIP-4844 Decouples Data Availability from Execution Gas

An Ethereum validator processes a Type 3 transaction carrying six blobs. Each blob commits roughly 128 KB of settlement data to a fee market that shares nothing with the execution layer's gas auction. That structural separation is the mechanism this section unpacks.

Every off-chain activity catalogued in prior chapters, from credential-gated channel updates through cross-chain HTLC resolution, eventually produces state diffs that demand on-chain anchoring. Before EIP-4844, those diffs competed for calldata space inside ordinary transactions, priced at 16 gas per nonzero byte. A rollup publishing even a modest batch of micropayment settlement proofs paid the same marginal gas rate as a DeFi swap or an NFT mint. Blob-carrying transactions eliminate that collision. They introduce a parallel data lane governed by its own base fee, its own target capacity, and its own exponential adjustment curve. Settlement data enters Ethereum through a door that DeFi traffic cannot crowd.

The cryptographic spine of this door is the KZG polynomial commitment scheme. Each blob is interpolated as a polynomial over a finite field, and the prover produces a single 48-byte commitment that lands on the execution layer. Any verifier can request a point evaluation proof, confirming that a specific piece of claimed blob content matches the committed polynomial, without downloading or re-executing the full payload. The binding and evaluation-correctness guarantees rest on the hardness of the discrete logarithm problem in pairing-friendly elliptic curve groups, bootstrapped by a one-time trusted setup ceremony known as powers of tau. That ceremony, completed in early 2023 with over 140,000 contributors, fixes the structured reference string. As long as at least one contributor destroyed their toxic waste, the commitments are computationally binding. For settlement compression, this means a rollup can anchor an entire batch of state channel closures behind a 48-byte fingerprint and later prove any individual diff's inclusion without touching execution gas.

Blob data is ephemeral by design. Consensus nodes retain blob payloads for a pruning window of roughly 4,096 epochs, approximately 18 days. After that, the data is discarded. This is not a limitation but a fit. Optimistic settlement schemes need retrievability only long enough for a fraud proof challenge period, typically seven days. Validity proof schemes need it for even less, just long enough for independent recon-

structors to verify the posted state root. Permanent storage was never what proof anchoring required. The pruning window gives settlement systems the availability guarantee they actually depend on while freeing consensus nodes from unbounded storage growth.

The blob gas pricing mechanism mirrors EIP-1559's exponential fee adjustment but targets three blobs per block, around 384 KB. When demand stays below this target, the blob base fee decays rapidly, collapsing toward 1 wei per blob gas unit. In practice, this means that during periods of moderate network activity, publishing settlement data via blobs can cost roughly 10 to 100 times less than equivalent calldata encoding. When demand spikes above six blobs, the exponential increase kicks in hard, rationing space. But the critical insight for autonomous agent settlement is that blob demand and execution demand are statistically uncorrelated. A surge in on-chain DeFi activity does not raise the price of publishing a batched state diff. The two fee markets breathe independently.

This independence rewrites the cost calculus for micropayment settlement infrastructure. Encoding batched state channel diffs into blob space gives settlement systems access to a data availability layer purpose-built for proof anchoring. KZG commitments guarantee that data was available at publication time. That single guarantee, data availability at the moment of commitment, is the only on-chain property that both ZK validity proofs and optimistic fraud proofs actually consume. Everything else, execution, permanent storage, state access, is overhead the settlement path never needed. Blob transactions strip that overhead away and leave a clean, cheap, cryptographically committed tape. The question that follows is precise: given this tape, how do you encode state diffs to extract maximum compression per blob byte?

Encoding Micropayment State Diffs into Blob Space: Compression Strategies and Per-Byte Cost Arithmetic

The hum of a saturating mempool carries a price signal. Every wasted byte in a blob is money burned. Engineering the encoding pipeline that packs micropayment state diffs into EIP-4844 blob space is not an optimization exercise. It is a survival condition for any settlement architecture publishing thousands of channel closures per hour.

A full two-party channel state carries roughly 200 bytes: addresses, nonces, balances, token metadata, signatures. Publishing that raw to a blob is grotesque waste. Delta encoding strips it to the essential mutation. One balance delta is a signed integer, typically 8 bytes. A sequence-

number increment is 4 bytes. A channel identifier compressed to a 4-byte index into a pre-registered table adds another 4 bytes. With a compact 2-byte flags field, a single state diff lands at around 18 bytes. A blob holds 131,072 bytes. That yields roughly 7,281 channel updates per blob. Compare this against full-state encoding, which packs around 655 updates. The compression ratio exceeds 11×. This is not theoretical. This is the packing ratio your batch assembler must target.

Heterogeneity across channels erodes that ratio unless you impose structure before serialization. Sort delta arrays by token denomination, then by delta magnitude. Run-length encoding collapses adjacent identical deltas into a count-value pair. For a batch of 5,000 channels where 70% share the same ERC-20 denomination and cluster within a narrow delta band, RLE cuts payload by an estimated 40 to 60 percent. Layer a bitmap mask on top: one bit per registered channel, 1 for active, 0 for unchanged. A 5,000-channel registry costs 625 bytes of bitmap. That bitmap eliminates the need to encode channel identifiers for inactive entries entirely. The sort-then-compress pipeline is mandatory. Skip it and you hemorrhage blob capacity.

Cost arithmetic pins the decision. Blob base fee follows its own EIP-1559 mechanism targeting 3 blobs per block. At a blob base fee of, say, 1 wei, the fixed cost per blob is 131,072 wei. Divide by 7,281 packed diffs and the cost per state diff falls below 18 wei. Calldata publication of the same 18-byte diff at 16 gas per non-zero byte costs 288 gas units, priced at the execution-layer gas market. At a 30 gwei gas price, that is around 8,640 gwei per diff. The crossover point where blob publication dominates calldata sits at roughly 50 or more state diffs per batch under typical fee ratios. Below that threshold, calldata wins. Above it, blobs crush calldata by over an order of magnitude.

Fill rate controls whether that cost advantage materializes. A half-empty blob still costs the full blob fee. You pay for 131,072 bytes whether you use 60,000 or all of them. The decision function is stark: if utilization sits below roughly 78%, delay publication and accumulate more diffs. But staleness imposes an upper bound. Any diff older than a configurable deadline, perhaps 12 blocks, forces a flush regardless of fill rate. This tension between cost efficiency and settlement latency is a scheduling problem with a crisp objective function. Minimize total cost per diff subject to a maximum staleness constraint. Your batch assembler must evaluate this predicate every block.

Serialization format selection closes the pipeline. SSZ aligns to 32-byte chunks. KZG polynomial evaluation over blob data operates on 32-

byte field elements natively. SSZ's fixed-width layout avoids the length-prefix overhead that RLP introduces and eliminates padding waste when diff sizes are uniform. For batches of uniform 18-byte state diffs, SSZ packing into 32-byte chunks wastes 14 bytes per chunk without grouping. Pack two diffs per 36-byte span, then pad to 64 bytes, and waste drops to around 28 bytes per pair. With careful struct alignment, SSZ delivers roughly 8 to 12 percent better packing efficiency than RLP across uniform diff batches. Choose SSZ. The polynomial structure of KZG demands it.

Every byte saved here compounds downstream. The proofs that attest to these batched state transitions inherit the compression ratio of the diffs they cover. Tighter diffs mean smaller witness data, faster proof generation, and lower verification gas. The pipeline you build in this section feeds directly into the proving circuits ahead.

Calldata vs. Blob Storage vs. External DA Layers: Settlement Cost Profiles Under Varying Transaction Volumes

Settlement cost is a function of volume, not a fixed architectural property. Every byte of state diff published to a data availability surface carries a price denominated in gas, tokens, or opportunity cost. The question is not which DA layer is cheapest in isolation. The question is where the crossover points fall as an autonomous agent's settlement throughput scales from hundreds to millions of state diffs per day.

Calldata remains the default. Post-EIP-3529, each nonzero byte costs roughly 16 gas. That cost scales linearly with volume. No discounts, no batching benefit, no separate fee market. For an agent settling a few hundred state diffs daily, calldata works. The gas overhead stays marginal relative to the value transferred. Push past roughly 10,000 to 50,000 state diffs per day and the arithmetic turns hostile. At current mainnet gas prices, calldata publication for a compressed micropayment batch of around 100 KB burns through gas budgets that dwarf the economic value of the underlying payments. The settlement margin collapses. Calldata's transparency and immediate L1 finality are genuine strengths, but they cannot rescue a cost curve that punishes throughput.

EIP-4844 blob storage operates on a fundamentally different cost surface. Blobs occupy a separate fee market governed by target and maximum blob counts per block. At moderate demand, blob space runs roughly 10 to 20 times cheaper per byte than calldata. A micropayment rollup publishing around 125 KB blobs hits a sweet spot between roughly 50,000 and 500,000 state diffs per day where blob costs domin-

ate calldata by an order of magnitude. The catch is congestion. Blob fee markets exhibit their own exponential pricing under saturation. When blob demand spikes, the per-byte cost can temporarily approach calldata parity. Agents that treat blob pricing as static will overpay precisely when throughput matters most.

External DA layers like Celestia, EigenDA, and Avail occupy the high-volume regime. Per-byte costs drop by two to three orders of magnitude relative to calldata. An agent settling millions of state diffs daily finds raw publication costs nearly negligible. The savings are real but the cost model is incomplete without pricing in what you lose. Finality latency increases by seconds to minutes. DA sampling introduces trust assumptions absent from L1 calldata. Reorg risk on the external chain creates exposure windows where settlement guarantees weaken. Proof verification and commitment posting back to Ethereum consume gas that erodes the raw per-byte advantage. At moderate volumes, these overhead costs can consume the entire savings margin.

The crossover analysis yields three regimes. Below roughly 10,000 state diffs per day, calldata wins on simplicity and finality. Between roughly 50,000 and 500,000, blob storage is unambiguously superior when blob fee markets are not saturated. Above 500,000 and certainly above one million, external DA layers dominate on raw cost despite their finality and trust trade-offs. The boundaries shift in real time with gas prices and blob demand.

This is why static DA selection is a liability. An agent locked into one layer overpays in every regime except its own sweet spot. Dynamic DA routing transforms the problem. An agent that monitors blob fee markets, calldata gas costs, and external DA pricing in real time can select the optimal publication surface per batch. Preliminary estimates suggest this approach captures 40 to 70 percent cost reductions compared to any single static strategy. The routing decision itself is cheap: a fee oracle query and a conditional branch in the settlement pipeline. Treat DA selection as a runtime procurement decision. The agent that routes settlement like it routes payments, adaptively and adversarially, holds the asymmetric advantage. Settlement cost optimization is not infrastructure planning. It is a live market.

Zero-Knowledge Spend Compliance Proofs: Aggregate Budget Attestation Without Individual Transaction Disclosure

A procurement agent commits fifty thousand transactions across a hundred providers, and the principal needs to know exactly one thing: did total spend stay under the cap? But the settlement layer, even after blob compression stripped away gas overhead, still publishes state diffs that expose every counterparty address, every unit price, every quantity. Cost optimization solved the wrong problem. The agent's negotiation leverage, supplier relationships, and pricing intelligence now sit in plaintext on a public ledger, readable by any competitor willing to parse calldata.

The fix lives at the circuit level. Pedersen commitments over a Merkle tree of individual transaction amounts let a prover construct a single attestation that the sum of all leaves falls within a range, without opening any leaf. The verifier, whether an on-chain contract or a principal's audit module, checks one proof and gets one bit: compliant or not. No line items, no provider identifiers, no strategic exposure. And the arithmetic compounds in the right direction. An entire fleet of agents settling a multi-provider procurement cycle can recursively aggregate their individual range proofs into a single verification artifact that hits the chain once, making auditability cheaper than the transactions it covers.

What matters now is how these circuits get built, what commitment schemes survive the latency budget, and where the Merkle tree's depth trades off against proof generation time under production load.

The Privacy-Auditability Tension in Autonomous Agent Expenditure: What Principals Need to Verify and What Must Remain Hidden

An agent fleet executing a procurement cycle commits hundreds of state updates across providers, channels, and settlement domains. The principal who delegated budget authority to that fleet has a narrow but non-negotiable set of things it must confirm after the fact: aggregate spend stayed within the allocated cap, no individual transaction exceeded the per-call rate ceiling, and every expenditure landed inside the authorized time window. Notice what is absent from that list. The principal does not need to know which specific provider received payment for a given inference call, what the content of the query was, or how the agent distributed its budget across service tiers. Full transaction visibility is not the verification requirement. Proof of compliance is.

But the principal is not the only observer whose information access matters. Providers represent a second, distinct adversary model. If a provider can inspect an agent's remaining budget or its historical spending distribution, that provider gains asymmetric pricing leverage. The congestion pricing mechanisms and auction dynamics from earlier chapters depend on agents entering markets without telegraphing their budget constraints. An agent that broadcasts "I have 40% of my allocation remaining with six hours left in my window" invites extraction. Provider-side privacy is not an aesthetic preference. It is a precondition for the game-theoretic pricing equilibria that keep compute markets efficient.

On-chain settlement introduces a third observer class. Even after the batch compression techniques covered in prior sections reduce on-chain footprint, whatever data does land on a public ledger is permanently readable. A sophisticated observer scanning settlement transactions can reconstruct fleet size estimates, infer procurement cadence, identify provider concentration, or detect budget utilization patterns that correlate with demand surges. These signals are actionable. A market participant who detects that a fleet consistently settles large batches before a particular time each day can position liquidity or adjust compute pricing in anticipation. Settlement compression reduces the attack surface, but it does not eliminate it unless the compressed proofs themselves are designed to hide these structural signals.

The critical insight is that these three observer classes have non-overlapping information needs, and conflating them produces broken architectures. A system that encrypts all settlement data and proves nothing satisfies provider privacy and chain observer resistance but destroys the principal's ability to enforce the budget delegation model. The spending caps from the hierarchical policy framework become decorative: an agent could overspend, and no verification mechanism would catch it until a channel dispute forced on-chain resolution. Conversely, a system that publishes full settlement records with cryptographic integrity proofs gives the principal complete auditability but hands providers and chain observers a detailed map of the agent's procurement behavior. Neither pole is acceptable.

The resolution is decomposition. The principal's verification needs break into three independent claim classes. First, aggregate compliance: total spend across all providers and channels did not exceed the budget cap. Second, per-transaction bound adherence: no single payment exceeded the rate constraint attached to that service category. Third, temporal validity: every expenditure occurred within the authorized time

window. Each of these claims can be proven independently using a zero-knowledge circuit that takes the full transaction history as its private witness and outputs only a binary attestation. The principal learns "compliant" or "non-compliant" for each claim without seeing individual amounts, counterparties, or timing details beyond what the proof structure requires. Providers see nothing about the agent's broader spending. Chain observers see a compact proof and a commitment, not a ledger.

This decomposition is not just conceptually clean. It maps directly onto composable circuit structures where each claim type has a well-defined witness format and public input set, enabling modular proof generation that avoids the latency and complexity of a monolithic proof over the entire spend history. The agent can generate proofs for each claim class in parallel, and the principal can verify them independently, requesting deeper disclosure only when a proof fails. What remains is the concrete question of how those circuits are constructed, what commitment schemes anchor them, and how they compose with the batch settlement proofs already established.

Circuit Design for Range-Bounded Spend Attestation: Proving Budget Compliance Over Committed Transaction Merkle Trees

The hum of a Poseidon hash rippling through an arithmetic circuit carries a specific cost: roughly 160 R1CS constraints per invocation. That number governs everything downstream when you design a zero-knowledge circuit that proves an agent's aggregate spend falls within a declared budget, all without exposing a single transaction amount. The circuit is not an abstraction. It is a constraint budget, and every gadget you wire into it consumes a measurable share of proving time.

Start at the leaves. Each transaction in the committed Merkle tree encodes a tuple of amount, provider identifier, timestamp, and nonce, packed into field elements and hashed with Poseidon to produce the leaf commitment. Inside the circuit, the prover re-derives each leaf from private witness values and verifies it against the supplied Merkle authentication path. This re-derivation is what binds the proof to the actual committed tree. If you skip Merkle path verification, a malicious prover can substitute arbitrary amounts into the witness, and the resulting proof attests to nothing. The choice of Poseidon over SHA-256 is not aesthetic. SHA-256 costs on the order of 25,000 R1CS constraints per hash. For a tree of 1,000 transactions with a depth of ten, hashing alone would consume around 250 million constraints with SHA-256 versus

roughly 1.6 million with Poseidon. The circuit simply does not close at production-viable proving times without an algebraic hash function.

Once leaves are authenticated, an accumulator subcircuit sums the transaction amounts sequentially. Each addition gate produces a running total that remains private throughout the proof. The critical engineering detail here is overflow prevention. Arithmetic circuits operate over a prime field, and if the running total exceeds the field modulus, it wraps around silently. An adversarial prover could exploit wraparound to hide massive overspend behind a small residue. The fix is a 64-bit range decomposition after each addition step. You decompose the running total into 64 binary variables and enforce that each bit satisfies the constraint $b_i \times (1 - b_i) = 0$, which costs 64 constraints per check. For 1,000 transactions, this adds around 64,000 constraints to the accumulator alone.

The final and most consequential gadget is the range-check that enforces budget compliance. Arithmetic circuits have no native comparison operator. Instead, you subtract the accumulated total from the declared budget cap and decompose the result into n bits, again enforcing binary constraints on each. If the subtraction yields a negative value in the integers, the field representation will not decompose into n legitimate bits. A valid bit-decomposition therefore constitutes a proof of nonnegativity, which is equivalent to proving total spend does not exceed the budget. This gadget costs another 64 constraints for a 64-bit budget cap.

Tallying the full constraint budget for 1,000 transactions yields roughly 200,000 to 250,000 R1CS constraints, depending on tree depth and field-packing choices. Groth16 proves this in an estimated two to four seconds on commodity hardware with a circuit-specific trusted setup. PLONK, using a universal structured reference string, lands closer to five to eight seconds but eliminates per-circuit ceremony overhead. For agent fleets that settle multiple budget periods, recursive proof composition with systems like Nova or folding schemes lets you amortize verification by compressing sequential budget proofs into a single accumulator proof, trading marginal prover cost for dramatic on-chain verification savings.

Two implementation pitfalls deserve explicit mention because they are silent and catastrophic. First, every witness value that the prover supplies must be fully constrained by the circuit's equations. An under-constrained amount field lets a dishonest prover inject a zero where a large spend occurred, and the proof verifies cleanly. Second, every public in-

put, the Merkle root, the budget cap, and the period identifier, must be bound to the proof's public input vector so the verifier knows exactly which commitment and which budget the proof covers. Without this binding, a valid proof can be replayed against a different root or a different cap, rendering the attestation meaningless. The constraint patterns that close these gaps are straightforward: enforce equality between the witness-derived Merkle root and the declared public input, and wire the budget cap directly into the range-check subtraction gate as a public signal rather than a private witness.

What emerges from this constraint-counting discipline is a circuit that is itself a budgetable artifact. Every design decision, from hash function selection through overflow guard placement to proof system choice, maps to a concrete cost in constraints, proving time, and verification gas. The circuit becomes auditable infrastructure, not cryptographic theater.

An Agent Fleet Settling a Procurement Cycle: Aggregated ZK Budget Proofs from Discovery Through Final On-Chain Commit

Replaying the demand oscillation trace that bankrupted three simulated channels in under forty seconds, Dayo Okonkwo stared at a whiteboard covered in fee-curve diagrams and realized the failure was not in pricing. It was in settlement. His Denver breakout session had started as a congestion-pricing exercise, modeling how autonomous agents would bid for inference capacity across his three-continent routing mesh. But the simulation exposed something more fundamental: fifty agents executing a procurement cycle against a shared principal budget generated so many individual channel settlements that on-chain costs consumed nearly twelve percent of the total spend. The agents discovered providers via DHT lookups, established channels, cycled L402 credentials for access, and streamed micropayments for GPU inference, all within a ninety-second window. Each of those phases produced settlement artifacts. The question was whether those artifacts could compress into a single on-chain commitment that proved budget compliance for the entire fleet without revealing which agents paid which providers or how much any individual transaction cost.

Dayo's team restructured the pipeline so that proof generation was not a separate step bolted onto settlement but a byproduct of it. Every phase of the procurement cycle already produced witness data: committed spend amounts hashed into each agent's local Merkle tree, provider identity hashes recorded during DHT discovery, timestamp bounds

locked when channels opened and closed. Instead of discarding this data and reconstructing proofs after the fact, each agent's payment state machine emitted witness fragments continuously. When an agent closed its streaming session, it held a complete local Merkle tree of all committed transactions and could generate a range-bounded spend proof attesting that its total outflow fell within its delegated sub-budget. This local proof was a compact Groth16 attestation, roughly 200 bytes, covering every micropayment the agent had executed during the cycle.

The aggregation step inverted the economics that had alarmed Dayo in his simulation. A fleet aggregator collected all fifty local proofs and recursively composed them into a single succinct proof. The recursive composition meant that verifier cost on-chain dropped from O(n) to O(1). Fifty individual on-chain settlements would have consumed an estimated 3.5 million gas in aggregate. A single recursive Groth16 verification landed around 230,000 gas, a reduction that turned fleet scale from a cost multiplier into a cost constant. The aggregated proof's public inputs revealed only three facts: total fleet spend was at or below the principal's cap, each included agent's spend was at or below its sub-budget, and all payments settled within the authorized time window. Provider identities, task-level allocations, and individual transaction amounts remained sealed inside the witness, invisible to the verifier contract and to anyone inspecting the chain.

Three of Dayo's fifty simulated agents failed to produce valid local proofs. One had a channel dispute pending resolution. Another carried stale state from a provider that went offline mid-stream. The third triggered an L402 credential expiry that left its final payment uncommitted. Rather than blocking the entire fleet, the aggregator excluded those three agents' sub-trees from the recursive composition, adjusted the aggregate budget bound downward by the sum of their allocated sub-budgets, and flagged the exceptions with their agent identifiers and failure codes for principal review. The remaining forty-seven agents settled in a single on-chain transaction. Graceful degradation, not atomic all-or-nothing commitment, preserved fleet throughput while isolating failures for targeted dispute resolution.

The final on-chain commit posted three items to the settlement contract: the aggregated proof, the Merkle root of all included agent spend commitments, and the updated budget state reflecting the consumed and remaining principal allocation. Dayo watched the transaction confirm and checked the contract's public state. One lookup revealed total fleet spend, the number of included agents, the excluded-agent flags, and

the rolled-forward budget available for the next procurement cycle. The principal could verify compliance without parsing fifty separate channel closures. Disputed agents entered a resolution queue with their failure metadata intact. The remaining budget rolled forward as a new commitment root, ready to anchor the next cycle's sub-budget delegations. What had looked like an unsustainable settlement burden in the simulation now resolved into a single transaction that cost less gas than four individual channel closes, with every agent's spend cryptographically bound to the principal's policy and not a single provider identity exposed on-chain.

Three compression mechanisms laid out across this chapter converge on a single design surface. Validity proofs collapse N channel-close verifications into one sublinear on-chain check. Blob transactions isolate state-diff publication from execution-layer gas competition. Zero-knowledge compliance attestations let an aggregated batch prove budget adherence without exposing any individual transaction. Taken separately, each looks like a point optimization. Taken together, they restructure settlement economics from a linear cost function into an amortized compression layer where privacy, auditability, and gas efficiency are coengineered rather than traded against one another. This is the inflection the entire stack has been building toward since Chapter 2 first displaced settlement off-chain: every subsequent layer of off-chain activity, from credential cycling through multi-hop routing through cross-chain atomicity, compounded a deferred finality debt. That debt is now payable at wholesale rates. On-chain settlement stops being the throughput ceiling and becomes a tunable parameter, governed by proof aggregation strategy and blob fee dynamics.

The binding constraint has moved. It no longer lives in verifier gas cost on L1 but in prover-side computation: proof generation latency per batch, witness size, and the hardware profile your settlement pipeline runs on.

Chapter Ten

Integrated Agent Compute Procurement: End-to-End Protocol Composition Under Production Constraints

According to a widely cited estimate from distributed systems research, roughly 97% of failures in production service architectures originate not from individual component bugs but from unanticipated interactions between correctly functioning subsystems. The number gains weight here. Every primitive examined across the preceding nine chapters works. State channels settle in sub-millisecond windows. Macaroon caveats attenuate cleanly. Congestion pricing curves converge on stable equilibria. Budget hierarchies enforce spending limits without human override. But an agent that must chain discovery, channel opening, L402 authentication, streaming payment, budget enforce-

ment, and cross-chain settlement into a single procurement loop confronts a combinatorial failure surface that no individual primitive was designed to absorb. Composition multiplies failure modes. It does not average them.

The gap between a library of correct primitives and a production-grade purchasing loop is an engineering surface with its own distinct topology. Dependency ordering between steps creates gating constraints where a stale route invalidates a channel reservation, where an expired macaroon poisons a payment stream mid-flight, where finality asymmetries across chains leave an HTLC locked past its budget window. Each of these failure classes emerges only at the seams, only when primitives execute together under real latency budgets and adversarial provider conditions.

That composition problem sharpens the moment an agent receives a task and must execute discovery, open a channel, authenticate, and begin streaming settlement within a single unbroken loop where each step's output gates the next. The execution sequence demands exact specification.

Full-Stack Execution: Discovery, Channel Establishment, L402 Authentication, and Streaming Settlement in a Single Agent Loop

Roughly seven out of ten protocol integration failures in distributed systems trace not to broken primitives but to broken composition, where individually correct subsystems interact under shared resource budgets they were never jointly tested against. That ratio sharpens considerably when the execution context is a single autonomous agent loop that must discover a provider, fund a channel, authenticate via L402, and begin streaming payment, all before a competing agent claims the same GPU slot. Each of those steps was specified in prior chapters to meet its own latency ceiling and security invariant. None was specified to share a millisecond budget with the others.

This is where the architecture faces its production exam. A DHT lookup that comfortably resolves in 120ms becomes a liability when it precedes channel funding, macaroon caveat binding, and a first HTLC, and the entire sequence must clear in under 500ms to remain competitive for scarce capacity. Failure surfaces multiply rather than sum: a stale routing hint poisons channel selection, a funding transaction that lingers in the mempool blocks credential exchange, a rejected caveat forces rollback across layers that have already committed partial state.

The engineering question is not whether each primitive works. It is whether the full transition sequence, treated as a deterministic state machine, converges under real latency distributions and real failure rates without corrupting cross-layer invariants or leaking economic exposure.

The Procurement Loop as a State Machine: Binding Discovery, Authorization, and Settlement into a Single Deterministic Execution Path

Roughly seven out of ten integration failures in distributed payment systems trace not to a broken primitive but to the gap between two working ones. A channel can open flawlessly, a macaroon can validate without error, and a streaming micropayment can settle to the correct sequence number, yet the composite system still leaks value if nothing binds these operations into a single, auditable execution path. The procurement loop that an autonomous agent executes when it discovers a compute provider, authenticates, streams payment for inference, and settles the final balance is not a sequence of API calls. It is a finite state machine with six states, and every transition between them fires on one condition only: a cryptographic predicate is satisfied.

The six states are Idle, Discovered, Channel-Ready, Authenticated, Streaming, and Settled. An agent in Idle transitions to Discovered when it retrieves a provider capability descriptor from the DHT pipeline and verifies the provider's signed pricing commitment. That signature is not decorative. It produces a quote digest that becomes the input binding for the next transition. When the agent opens a payment channel, the funding transaction's output must commit to this exact quote digest. Without that binding, a window opens between discovery and channel establishment where an adversary, or even a misbehaving provider, can substitute pricing terms. In competitive GPU spot markets, that window can be exploited in under 200 milliseconds. The transition from Discovered to Channel-Ready therefore requires the agent to verify that the channel funding outpoint references the signed quote, collapsing discovery and channel establishment into a single trust chain rooted in the provider's commitment signature.

Channel-Ready to Authenticated is where L402 credential issuance anchors to the payment layer. The 402 challenge returned by the provider contains a payment hash. That hash must reference the same channel funding outpoint that the agent just committed to, ensuring that the macaroon's validity and the channel's payment capacity share a single cryptographic root. If the payment hash floats free of the channel, an

agent could present a valid credential funded by a different channel, or worse, present a credential whose payment was never routed through the agreed pricing terms. Binding the L402 challenge to the funding outpoint eliminates this class of credential-payment divergence and ensures the provider can verify, in one check, that the agent both can pay and is authorized to consume the service.

Once authenticated, the agent enters the Streaming state. Each micropayment increments the channel's sequence number and simultaneously extends the macaroon's usage caveat. These two state advances move in lockstep. The sequence number proves how much the agent has paid. The usage caveat proves how much service the agent is authorized to consume. If either advances without the other, the system enters an invalid state that the counterparty can detect immediately. This monotonic coupling is the mechanism that prevents an agent from consuming more compute than it has paid for or paying for compute it never received. Authorization state and payment state are not parallel tracks converging at the end. They are the same track, advanced by the same cryptographic operation.

The terminal transition, from Streaming to Settled, demands a cooperative close signature from both parties or a unilateral close backed by the final state proof. Either path must produce an audit artifact, a chain of cryptographic witnesses linking the settlement back through every prior transition: the final sequence number, the authenticated macaroon with its accumulated usage caveats, the L402 payment hash anchored to the funding outpoint, and the signed quote digest from discovery. This chain is not optional reporting. It is the proof that the procurement loop executed deterministically, that no transition occurred without its predicate being satisfied, and that the agent's principal can reconstruct the entire execution path from a single settlement receipt. Any glue code between primitives that lacks a binding predicate is not an implementation detail. It is an attack surface. The state machine makes that fact structural, turning composition discipline from a best practice into an enforceable invariant.

Latency Budgets Across the Stack: Time Partitioning Between DHT Lookup, Channel Funding Confirmation, Macaroon Validation, and First Streaming Payment

An agent's procurement loop burns through its total latency envelope in four distinct phases, each consuming time that the others cannot recover. Treating these phases as independent optimization targets misses the

point. Time is a finite resource allocated across DHT discovery, channel funding, macaroon validation, and the first streaming payment. The discipline is not making each phase faster in isolation but partitioning an explicit budget across all four, deciding where phases can overlap, and enforcing ceilings that keep the entire loop within a hard deadline. This guide walks through that partitioning, from defining the envelope to instrumenting a live adaptive control loop.

Step 1: Define the Total Latency Envelope and Assign Phase Sub-Budgets

Start with a concrete wall-clock target. For compute procurement where an agent must discover a provider, authenticate, and begin streaming payment before a task deadline expires, an envelope of roughly 800ms represents an aggressive but achievable warm-path ceiling. Decompose this into four explicit sub-budgets based on the cost structure of each phase. DHT lookup receives around 50–150ms depending on hop count and whether local caches hold recent routing hints. Channel funding confirmation, the dominant cost, can consume 30–120 seconds on a cold path where a new channel must be opened and confirmed on-chain. Macaroon issuance and caveat validation occupy approximately 5–20ms for local HMAC-chain verification. First streaming payment dispatch requires around 10–50ms for HTLC commitment exchange. Write these ceilings into the agent's configuration as enforceable constants, not aspirational guidelines.

1. Measure baseline latency for each phase in your target deployment environment using instrumented test runs against representative providers.
2. Set the total envelope as a hard constraint derived from your task's real-time deadline minus a safety margin of 10–15%.
3. Assign each phase a sub-budget proportional to its measured baseline, reserving the largest allocation for whichever phase dominates your deployment profile.
4. Record these allocations in a configuration structure the agent's scheduler can read and enforce at runtime.

Step 2: Pipeline Phases to Collapse Sequential Latency into Parallel Execution

Sequential execution of all four phases would blow through any reasonable envelope on even a warm path. The solution is pipelining. As soon as a DHT lookup returns candidate provider records, the agent can speculatively initiate channel pre-funding or channel-selection logic before provider negotiation completes. Macaroon validation can execute concurrently with the final funding-confirmation watch. The first streaming payment can be dispatched within the same message round-trip as credential presentation. Design the agent loop so that each phase emits a partial result that the next phase can consume without waiting for full completion. DHT results arrive incrementally as each Kademlia hop resolves. The channel selector can begin evaluating outbound capacity on existing channels the moment the first candidate provider ID appears. Credential construction can begin with a provider's known public key while the channel's latest commitment transaction is still being countersigned.

1. Structure the procurement loop as a set of concurrent tasks with explicit dependency edges rather than a flat sequential pipeline.
2. Emit DHT lookup results as a stream, triggering channel-selection evaluation on each new candidate rather than waiting for the full result set.
3. Overlap macaroon construction with the final round of channel state negotiation so that credential presentation and the first HTLC can share a single message exchange.
4. Validate that pipelined phases still respect their individual sub-budgets by instrumenting each task's start and completion timestamps independently.

Step 3: Eliminate Funding Latency Through Channel Reuse and Pre-Funding Strategies

The funding bottleneck disappears when the agent never needs to fund a new channel during the procurement loop. Maintain a warm channel pool to frequently-used providers. Channel factories amortize the on-chain setup cost across multiple provider relationships, opening several channels in a single transaction confirmed during idle periods. When a procurement request arrives, a channel-selection heuristic picks an existing channel with sufficient outbound capacity before falling back to new funding. The latency difference is stark. A warm-path procurement that selects an existing channel with adequate balance completes discovery through first payment in roughly 60–200ms total. A cold-path procurement requiring new channel funding stretches to 30–120 seconds. The agent's budget allocator should treat these as distinct operating modes with separate budget profiles, switching between them based on channel pool state at the moment a procurement request enters the loop.

1. Build and maintain a channel pool indexed by provider identity, available outbound capacity, and recent latency measurements.
2. During idle periods, use channel factories to batch-open channels to providers that historical procurement patterns predict the agent will need.
3. Implement a selection heuristic that scores existing channels by outbound capacity, fee rate, and measured round-trip latency, falling back to cold-path funding only when no channel meets minimum capacity requirements.
4. Tag each procurement attempt as warm-path or cold-path at initiation so that the budget enforcer applies the correct ceiling profile.

Step 4: Enforce Budget Ceilings with Mechanistic Steal-or-Fail Decisions

When a phase exceeds its allocated window, the agent faces a binary choice: steal time from a downstream phase or fail fast and fall back to an alternative. This decision must be mechanistic, not heuristic. Compute the remaining total budget minus the minimum required time for all downstream phases. If the overrunning phase can complete within that slack, extend its allocation. If not, abort it immediately and switch to a cached fallback. For DHT lookup specifically, this means the agent keeps a fallback provider cache populated from prior successful discoveries. If the lookup exceeds its sub-budget and the remaining envelope cannot absorb the overrun without starving channel selection or credential validation below their minimum thresholds, the agent cancels the lookup and routes procurement to a cached provider. The decision rule is simple arithmetic applied at each phase boundary.

1. At each phase transition, compute `remaining_budget = total_envelope - elapsed_time` and compare it against `sum(minimum_cost[downstream_phases])`.
2. If `remaining_budget - sum(minimum_cost[downstream_phases]) > 0`, allow the current phase to continue consuming the slack.
3. If the slack is zero or negative, immediately cancel the current phase and invoke the fallback path: cached provider for discovery, existing channel for funding, pre-computed credential for authentication.
4. Log every steal-or-fail decision with the computed slack value for post-hoc analysis of budget accuracy.

Step 5: Instrument Per-Phase Latency and Calibrate Budgets Adaptively

Static budgets degrade as network conditions, provider behavior, and channel liquidity shift. The agent must record per-phase latency distributions over rolling windows, typically the last 100–500 procurement cycles, and detect when a provider's channel-establishment latency or DHT reachability drifts beyond its allocated budget. When drift is detected, the agent recalibrates in two ways: it adjusts the sub-budget allocations for future cycles, and it re-weights provider selection scores to route procurement toward lower-latency paths. This transforms the latency budget from a static design artifact into a live control loop. The agent's procurement scheduler consumes updated latency percentiles, typically p95 or p99, and recomputes phase ceilings before each procurement cycle. Providers whose measured latency consistently exceeds budget receive lower selection scores, gradually shifting traffic toward providers that meet the agent's time constraints without manual intervention.

1. Record wall-clock timestamps at each phase boundary and compute per-phase durations for every procurement cycle.
2. Maintain a rolling window of per-phase latency samples, computing p50, p95, and p99 percentiles for each phase and each provider.
3. When a provider's p95 latency for any phase exceeds 120% of the allocated sub-budget for three consecutive windows, flag the provider for score demotion in the selection heuristic.
4. Recompute sub-budget allocations at configurable intervals using the updated percentile data, ensuring the sum of sub-budgets plus safety margin equals the total envelope.

You now have a framework that treats latency as an allocated resource rather than a number to minimize. Each phase of the procurement loop operates under an explicit ceiling, phases overlap where dependencies allow, and the dominant funding bottleneck is architecturally eliminated through channel reuse. Budget enforcement follows a mechanistic steal-or-fail rule that keeps the agent within its total envelope even when individual phases misbehave. Adaptive instrumentation closes the loop, turning static allocations into a living control system that responds to shifting network and provider conditions. With these budgets in place, the next challenge is what happens when a phase does

not merely exceed its budget but fails outright, triggering the retry and rollback logic that keeps the procurement machine operational under fault conditions.

Failure Recovery at Every Layer: Retry Semantics, Partial-State Rollback, and Graceful Degradation When Any Subprotocol Stalls

The most revealing stress test for any composed procurement loop is not how cleanly it executes the happy path but how precisely it distinguishes between a stalled DHT query and a failed HTLC settlement. These two failures demand entirely different recovery actions, yet the default engineering impulse collapses them into a single retry wrapper. That conflation is where cascading state corruption begins. A production-grade agent loop survives real network conditions only when every subprotocol layer carries its own failure taxonomy, its own retry semantics, and its own explicit rollback fence separating it from adjacent layers.

Start with the failure signatures themselves. A DHT discovery timeout means the agent received no ranked candidate list within its latency partition. The correct retry is an idempotent re-query, possibly against a different bootstrap node, because discovery carries no committed state. A channel-funding delay, by contrast, involves a broadcast transaction that may or may not have entered the mempool. Retrying here means fee-bumping via RBF or CPFP, not rebroadcasting a duplicate funding output. L402 credential failures split into caveat expiry, where the macaroon's time-bound or invocation-count constraint has lapsed, and nonce desync, where the provider's expected credential state diverges from the agent's local state. Both demand credential re-derivation with freshly attenuated caveats rather than replaying the expired token. Settlement-layer failures, specifically preimage non-disclosure or HTLC timeout, trigger the timeout-claim path on-chain, recovering locked funds through the timelock branch. Each of these four failure types maps to a single correct recovery action. Applying the wrong one burns latency, leaks liquidity, or corrupts downstream state.

Rollback boundaries enforce this discipline structurally. When L402 validation fails, the agent must not unwind an already-funded channel. The channel represents committed capital with its own on-chain anchor. Tearing it down because an authentication token expired forces re-establishment from scratch, consuming roughly 40 to 60 percent of the total latency budget on redundant funding confirmation. Instead, the state machine maintains explicit checkpoint invariants: channel-funded-

but-unauthenticated, authenticated-but-unsettled. Recovery logic inspects these checkpoints, replays only the failed layer, and preserves everything below it. The rollback fence between channel establishment and credential cycling is the single most valuable boundary in the entire loop.

Graceful degradation under provider failure leverages the discovery layer's ranked candidate list. A well-designed DHT query returns not a single provider but an ordered set, scored by price, latency, and reputation. The state machine maintains a provider cursor that advances on failure rather than resetting. When the current provider's channel funding stalls or its L402 endpoint becomes unreachable, the agent steps to the next candidate without re-executing the full discovery query. The cached result set is the degradation buffer.

Cross-layer failure correlation changes the recovery calculus entirely. When channel funding stalls and L402 validation times out simultaneously, two independent retry loops firing in parallel amplify congestion on an already degraded network path. The correlated signal, both layers blocked at once, points to a network-level partition rather than two independent bugs. An agent that detects this correlation suppresses per-layer retries, fires a single network-health probe, and enters a global exponential backoff. This converts a retry storm into a measured pause, preserving bandwidth for the recovery attempt that actually matters.

These mechanisms compose into measurable recovery targets that serve as deployment acceptance criteria. A single-layer failure should resolve within roughly 1.5 times the original latency budget allocated to that layer. Correlated multi-layer failures should resolve within approximately 3 times the total loop budget. In both cases, at least 80 percent of already-committed state transitions, funded channels, valid credentials, partial settlement proofs, must survive intact. Agents that hit these targets tolerate two or three simultaneous subprotocol failures without full-loop abortion, resuming from the last consistent checkpoint rather than replaying the entire pipeline from discovery. The difference between an agent that meets these thresholds and one that does not is the difference between a procurement loop that operates under real network conditions and one that operates only in demos.

Budget-Constrained Task Scheduling Across Heterogeneous Providers with Real-Time Congestion-Aware Pricing

Roughly seven out of ten autonomous dispatch decisions made under static pricing assumptions overshoot their budget ceiling within the first task sequence. That estimate, drawn from simulation work on multi-provider inference routing, lands harder once you internalize what it means in production: an agent that treats provider pricing as a lookup table instead of a live signal will burn through allocated satoshis before the job completes, leaving partially settled channels and unfinished compute scattered across settlement domains.

The single-agent procurement loop assumed a cooperative provider at a discoverable price. Strip that away and the problem surface detonates. An agent holding a fixed budget now faces a joint optimization across price, latency, and remaining channel liquidity that shifts every time a provider's congestion-aware pricing function updates. A 12% rate drop on one provider looks like a gift until the channel rebalancing required to exploit it consumes the savings and then some. Budget headroom itself becomes a moving target, shrinking nonlinearly as congestion surges, so an agent that cleared every solvency check three seconds ago can find itself unable to close the task it already started.

This is where composition under economic pressure either holds or fractures. The decisions ahead involve how an agent selects providers when price, latency, and remaining headroom constrain each other simultaneously, how spot-versus-precommit strategies map onto formal trade-off surfaces, and what happens when those decisions must resolve in the same latency window the market is repricing.

Provider Selection Under Joint Constraints: Balancing Price Signals, Latency SLAs, and Remaining Budget Headroom in Real Time

Roughly 6 in 10 simulated multi-task procurement runs that select providers on price alone exhaust their budget before completing the final task. That number collapses to fewer than 1 in 20 when the selection logic treats remaining budget headroom as a first-class constraint alongside price and latency. The gap between those two outcomes is the distance between a naive lookup and a real optimization, and it exposes a failure mode that no single protocol layer can fix in isolation. Choosing a provider is not a filter-then-sort operation. It is a sequential consump-

tion of finite optionality under uncertainty, where every commitment forecloses future states.

The joint constraint space makes this concrete. Price per compute unit, advertised latency at the p95 or p99 percentile, and the ratio of remaining budget to estimated remaining tasks are not independent axes that can be optimized in sequence. A provider quoting a 40% price spike during a congestion event may still clear the per-task budget cap, but consuming 70% of remaining headroom on a single dispatch kills the agent's ability to absorb even modest price increases on subsequent tasks. The true cost of that selection is not the sticker price. It is the sticker price plus the expected value of the options it destroys. This is exactly the portfolio-style framing that transforms procurement from a lookup problem into constrained sequential decision-making.

The scoring function that resolves this operates on three normalized inputs. Normalized price maps the quoted cost against the per-task budget cap. Latency-SLA margin measures the distance between the provider's advertised percentile and the SLA ceiling, not merely whether the provider clears the threshold, because a provider sitting 2ms below a 200ms ceiling offers far less margin for jitter than one sitting 40ms below it. Budget runway ratio divides remaining budget by estimated remaining task count to produce a per-task headroom figure. A composite score weights these three signals into a single ranking, and the weights themselves shift dynamically: as budget depletes below roughly 30% of initial allocation, the runway ratio weight escalates sharply, and as deadline pressure increases, latency margin weight dominates. The function is not static. It is a sliding surface that tracks the agent's resource position in real time.

But scoring is only valid if the input signals are fresh, and they often are not. Provider pricing metadata propagated through DHT registries or cached from prior lookups can be anywhere from 500ms to 2s stale. A price quoted 1.5 seconds ago during a congestion spike may have already reverted. The scoring function must therefore incorporate confidence intervals around advertised prices, widening the uncertainty band as the timestamp ages, and penalize providers whose last-known update exceeds a configurable freshness threshold. Without this correction, the agent optimizes against a phantom market.

Before the composite score is ever computed, hard threshold gates eliminate providers that violate any single constraint outright. Latency above the SLA ceiling: eliminated. Single-task cost exceeding the per-task budget cap: eliminated. Provider reputation score below the minimum

acceptable floor established by the principal's delegation policy: eliminated. This threshold-gated elimination reduces the optimization surface to a tractable set, bounding worst-case selection latency to a deterministic function of surviving candidates rather than total registry size. The scoring function then ranks only the survivors, and the top-ranked provider receives the dispatch.

What emerges is a selection engine that consumes price signals from the game-theoretic pricing layer, enforces budget constraints from the hierarchical spending policies, respects latency bounds from the full-stack latency budget partitioned in the prior section, and degrades gracefully as resources deplete. Each dispatch decision feeds back into the runway ratio, shifting the weight surface for the next selection. The agent that began this procurement cycle with a full budget and broad optionality now operates in an increasingly constrained corridor, and the scoring function tracks that corridor's walls in real time. This is the optimization core. The next question is what happens when the corridor narrows across three settlement domains with divergent spot pricing and heterogeneous finality horizons.

An Inference Dispatch Agent Navigating GPU Spot Pricing Across Three Settlement Domains

Rina Subramanian pinned the failure trace to her second monitor and unmuted the governance call. Three budget delegation failure modes demanded immediate attention, but the one burning hottest involved a live inference dispatch agent that had just blown through its atomicity assumptions across three settlement domains in under two seconds. She pulled up the telemetry.

The agent held a 47-token inference job. Three sequential GPU passes. Provider A quoted roughly 12 sats per pass over Lightning with around 140ms latency. Provider B offered an estimated 0.0003 ETH on an L2 rollup carrying 4-second soft finality. Provider C priced at approximately 0.008 SOL through a Solana escrow program with 400ms slot confirmation. Each domain imposed a different atomicity boundary. The agent's total latency budget sat at 2.1 seconds end-to-end. Rina walked the board through the constraint: no single hash preimage spans all three settlement layers simultaneously. The agent cannot construct a cross-domain HTLC. It must build a sequential commitment chain instead. Each payment conditionally authorizes based on cryptographic proof that the prior pass completed. Macaroon caveats encode intermediate result hashes as discharge conditions. Pass 1's output hash becomes

the caveat that unlocks pass 2's payment authorization. Pass 2's output hash unlocks pass 3. The chain is fragile by construction.

She advanced her slide to the price spike. Between the agent's quote-lock on Provider B and actual job submission, a window of roughly 1.2 seconds elapsed. In that interval, rollup congestion pricing jumped an estimated 38%. The agent's budget guard fired. Three options surfaced in the decision loop: absorb the cost and compress remaining headroom, re-route to a slower fallback provider at the original price, or split remaining passes across Providers A and C at degraded parallelism. The agent chose to proceed with Provider B. Rina flagged this as the mechanism failure. The budget delegation hierarchy allowed the agent to absorb up to 40% cost overruns on any single pass. But it did not encode a constraint linking cumulative overruns to downstream pass feasibility. The agent spent its headroom on pass 2 without verifying pass 3 remained affordable.

Then the cascade hit. Provider C missed its Solana slot due to congestion. The Lightning HTLC for pass 1 had already revealed its preimage. The rollup streaming payment for pass 2 was mid-flight, partially settled. The agent executed a partial rollback on the rollup, clawing back uncommitted stream increments. It absorbed the sunk 12 sats on Lightning. It re-dispatched pass 3 to Provider A within remaining budget headroom of around 0.0004 ETH equivalent. Total recovery latency consumed 1.7 seconds. The job completed 340ms over budget. Rina's point to the board was blunt: the agent recovered, but only because Provider A had residual channel capacity. That capacity was not guaranteed. The sequential commitment chain created a dependency graph where early preimage revelation locked in costs that could not be recovered if downstream providers failed.

She proposed the constraint hierarchy. Each pass's budget authorization must carry a conditional ceiling tied to the aggregate remaining budget across all downstream passes. The macaroon caveat structure already supports this. A discharge macaroon for pass 2 can encode not only the pass-1 result hash but also an upper bound on cumulative spend through pass 3. If the pass-2 price exceeds what leaves sufficient headroom for pass 3, the caveat fails to discharge. The agent re-routes before committing. Rina pulled up her catalog of production failures. Fourteen of the last twenty cross-domain dispatch failures traced to this exact pattern: early settlement commitment consuming budget headroom that downstream passes required. The fix was not smarter routing. The fix was encoding budget interdependency into the cryptographic

authorization layer itself, making overspend physically impossible at the caveat level before any payment leaves the agent's control. The board voted to mandate the constraint hierarchy for all multi-domain dispatch agents within the next protocol revision cycle.

When to Precommit and When to Spot-Purchase: A Decision Model for Task Urgency, Price Volatility, and Channel Liquidity

Locking capacity costs money. Failing to lock capacity costs more. The entire procurement question collapses to this asymmetry, and the agent that cannot quantify both sides will bleed value on every task cycle.

Three variables govern the decision. Deadline slack measures how much latency buffer remains before a task's economic value degrades to zero. Price volatility captures the coefficient of variation across recent congestion pricing signals from candidate providers. Channel liquidity registers the uncommitted balance across open channels, weighted by hop distance to viable compute endpoints. These three axes form a procurement mode selector. When deadline slack falls below roughly 2× expected execution time, precommitment dominates. The cost of failing to secure capacity exceeds any premium paid for reservation. No heuristic survives production load. Only quantified thresholds hold.

Precommitment is a call option on compute. The agent locks collateral in a channel with a specific provider and negotiates a rate ceiling, embedding a max-price constraint as a macaroon caveat. The cost of this option is the opportunity cost of locked liquidity, capital that cannot route through other channels, plus any explicit reservation fee. The break-even condition is calculable: precommit when expected spot price variance multiplied by task value exceeds the option cost. This is not metaphor. The agent's budget hierarchy enforces the ceiling cryptographically. The settlement path remains atomic. The option either exercises at delivery or expires, releasing collateral back to the channel.

Spot-purchasing exploits price discovery across heterogeneous providers. It thrives when deadline slack is generous and the agent holds open channels to five or more distinct providers in the same compute class, driving spot-failure probability below an estimated 3%. But execution risk is real. It measures the probability that no provider offers capacity below the agent's budget ceiling within the deadline window. DHT-published utilization metadata and historical congestion curves make this risk quantifiable. Latency-tolerant tasks with broad channel coverage belong in spot mode. Everything else pays the precommitment premium or risks task failure.

Channel liquidity is not a passive constraint. It is the bridge variable that determines which mode is even available. An agent with deep liquidity concentrated in few channels is structurally biased toward precommitment with those providers. An agent with shallow liquidity distributed across many channels is positioned for spot-purchasing. This means rebalancing channels is itself a procurement strategy decision. Shifting satoshis between channels reshapes the agent's procurement surface. The decision model must co-optimize mode selection and liquidity topology in the same cycle. Treating rebalancing as routine maintenance ignores its strategic weight.

Adaptive mode-switching demands hysteresis, not point thresholds. An agent that flips between precommit and spot on every pricing tick wastes channel operations and caveat-cycling overhead. The decision model introduces a commitment inertia parameter: once a mode is selected for a task class, the agent holds that mode until the triggering variable crosses a threshold band. Not a point. A band. This reduces mode-switching frequency by an estimated 60–80% while preserving responsiveness to genuine regime shifts in urgency, volatility, or liquidity. The band width itself is tunable per task class, narrower for high-value inference jobs, wider for batch workloads. Production agents that skip hysteresis will oscillate into operational noise. Those that implement it gain stable procurement behavior under volatile pricing, without surrendering the ability to pivot when conditions genuinely change. The full dispatch cycle across settlement domains demands this stability as its foundation.

Audit Trail Construction, Post-Hoc Principal Review, and Accountability Across Cross-Chain Multi-Provider Transactions

Roughly seven in ten off-chain payment artifacts, by conservative estimate, lack any binding reference to the authorization event that triggered them. The agent loop that just executed budget-constrained scheduling across heterogeneous providers did so in under two seconds, settling hundreds of micropayments along the way. But the principal who delegated that authority observed none of it in real time. What remains afterward is a scatter of HTLC preimages in one data store, L402 macaroons with caveats in another, and channel state updates persisted according to each node's own retention policy. None of these artifacts inherently reference each other. Without a cryptographic linkage that binds each receipt to the authorization that permitted it and the settlement event that

finalized it, post-hoc review is not auditing. It is reconstruction under uncertainty.

This gap is not a logging problem solved by better database hygiene. It is a structural consequence of composing multiple off-chain protocols across independent chains and providers, each with distinct finality semantics and data lifetimes. The principal needs a single tamper-evident record that proves aggregate budget compliance, and ideally one that does so without forcing the agent to expose every provider selection or task allocation along the way. That requirement pulls the design toward proof systems capable of collapsing a full transaction graph into a compact, verifiable summary. But every additional commitment the agent must compute before acting adds latency to the procurement loop, creating a direct tension between how deeply the principal can verify and how fast the agent can operate.

Constructing Cryptographically Linked Audit Logs from Off-Chain State Updates, L402 Receipts, and Cross-Chain HTLC Preimages

Roughly seven out of ten enterprise audit frameworks still treat logging as an afterthought: a sidecar process that observes production systems and records what it can. That model collapses the moment an autonomous agent executes a procurement loop across off-chain state channels, credentialed API endpoints, and cross-chain HTLC bridges. The artifacts already generated during that loop, the signed state updates, the L402 discharge proofs, the revealed hash preimages, carry exactly the cryptographic properties an audit record demands. Tamper evidence, chronological ordering, and third-party verifiability exist by construction. The engineering task is not to build a logging system. It is to link what already exists.

Start with the payment channel itself. Every off-chain state update carries a monotonically increasing sequence number and bears bilateral signatures from both counterparties. No valid update can skip a sequence number, and no prior state can be reintroduced without triggering the channel's dispute mechanism. This total ordering produces an immutable spending ledger for free. An agent that retains its signed state transitions can reconstruct, at any future moment, the exact sequence of payment commitments it made within that channel. No external database, no append-only log service, no trusted timestamping authority. The channel's own consensus rules enforce integrity more tightly than any observability layer bolted on afterward.

L402 receipts bind a second dimension. When an agent authenticates against a credentialed service endpoint, the provider issues a macaroon scoped by specific caveats, tied to a Lightning payment hash. The agent obtains the preimage by settling that hash over the payment network. Retaining the preimage-receipt pair gives the agent a self-authenticating audit entry: cryptographic proof that it paid a precise number of satoshis for a service bounded by exactly those caveats. Each L402 interaction therefore documents not just a payment but the authorization scope of that payment, the resource it unlocked, and the economic terms under which access was granted. No additional instrumentation captures this triple binding as tightly as the receipt itself.

Cross-chain HTLC preimages anchor the third layer. When the inference dispatch agent settled GPU inference fees across three settlement domains, each atomic swap concluded with a preimage revelation that proved settlement on every participating chain. The timelock parameters embedded in each HTLC encode the temporal bounds of the swap window. An auditor holding a log of revealed preimages and their associated timelocks can reconstruct the full cross-chain payment flow without querying any base-layer node. Possession of the preimage is the settlement receipt. Its absence is proof that settlement did not complete.

Three artifact types now sit on the table: sequenced channel states, caveat-scoped L402 proofs, and timelock-stamped HTLC preimages. Linking them into a unified audit chain requires a commitment structure that makes deletion and reordering detectable. A hash chain serves this purpose directly. Each new entry contains the cryptographic receipt alongside the hash of the previous entry. Alternatively, a Merkle log allows selective disclosure: a principal can verify that a specific transaction exists at a specific position in the chain without downloading every intervening record. Either structure produces a tamper-evident sequence that breaks visibly if any record is removed, inserted out of order, or modified after the fact.

The result is an audit log that a principal verifies by holding only root commitments and selected inclusion proofs. Full transaction replay becomes unnecessary. The principal checks that the Merkle root matches the agent's claimed spending summary, spot-checks individual entries against their cryptographic receipts, and confirms that sequence numbers and timelocks produce a consistent chronological ordering. This is not observation. This is mathematical proof of what the agent did, constructed from the same cryptographic material that made the transactions possible in the first place. And it sets the stage for some-

thing more powerful still: proving budget compliance over these records without revealing the records themselves.

Zero-Knowledge Spend Summaries for Principal Review: Proving Budget Compliance Without Exposing Provider Identity or Task Content

The hum of a spend ledger closing out is invisible, silent, entirely internal to the agent's runtime. Yet the moment a principal requests proof that a delegated budget was honored, every receipt, every routed HTLC, every discharged macaroon credential becomes a potential leak. Provider identities reveal supply-chain positioning. Task content exposes inference strategies. Pricing data hands competitors a map of your cost structure. The engineering challenge is not whether to audit but how to prove exact budget compliance while disclosing nothing about who was paid, for what, or at what rate. Zero-knowledge spend summaries solve this by converting the full procurement record into a single cryptographic attestation that the principal can verify in milliseconds.

The witness structure anchoring this proof is a Merkle tree whose leaves encode individual payment records. Each leaf contains a committed tuple: payment amount, category tag matching the principal's budget taxonomy, a timestamp, and a pseudonymous provider identifier derived from the L402 receipt's payment hash. The agent constructs this tree after each delegation period closes, inserting one leaf per settled transaction. Inside the ZK circuit, the prover demonstrates three properties simultaneously. First, the sum of all leaf amounts equals the claimed total and falls at or below the principal-set cap. Second, each leaf's amount respects the sub-budget ceiling for its declared category. Third, every leaf timestamp lands within the delegation window's start and end bounds. The principal never sees a single leaf opened. The proof speaks for the entire tree.

Range-bounded aggregation inside the circuit is where the real performance discipline lives. Encoding the constraint "cumulative spend is at most X satoshis" requires composing range proofs within the arithmetic circuit so that each leaf's value is proven non-negative and bounded before summation. Under a PLONK-based proving system, the prover generates this proof off-chain in roughly 3 to 8 seconds depending on tree depth and leaf count, while the verifier confirms it in under 2 milliseconds on-chain. Compare this to naive enumeration of individual Merkle proofs, which scales linearly and pushes verification past 400 milliseconds for even modest transaction sets. The circuit's

fixed verification cost makes it viable for on-chain attestation contracts or lightweight local verifiers running on the principal's own infrastructure.

Completeness is the adversarial surface that demands the sharpest design. A dishonest agent could simply omit expensive transactions from the Merkle tree, presenting a budget-compliant subset. The countermeasure binds the ZK proof to the macaroon discharge chain established during procurement. Every L402 receipt produces a payment hash that the agent used to settle an HTLC. The principal's verifier holds or can reconstruct the set of discharge tokens issued under the delegated credential. The circuit must prove that for each such discharge token, a corresponding leaf exists in the committed tree. Any omitted receipt becomes a missing preimage, and the verifier's completeness check fails deterministically. The agent cannot hide a transaction without also hiding the credential discharge that authorized it.

By default, the principal learns exactly four values from the proof: total spend, the per-category breakdown, transaction count, and the bounding time range. Nothing about provider identity, task semantics, or unit pricing crosses the trust boundary. Yet the proof supports progressive disclosure. If a dispute arises, the agent can selectively open individual Merkle leaves, revealing only the contested records while leaving the rest sealed. This is not retroactive privacy loss. It is a designed escalation path where the proof's structure guarantees that opening one leaf does not weaken the hiding property of any other.

The implementation pipeline moves through four concrete stages. First, the agent collects L402 receipts and HTLC preimages during active procurement, storing them in an append-only local log. Second, after the delegation period closes, the agent constructs the spend Merkle tree and commits its root. Third, the agent generates a Groth16 or PLONK proof off-chain, targeting a circuit sized for the expected leaf count, with proving cost estimated around 0.5 to 2 cents of equivalent compute at current GPU rates. Fourth, the agent submits the proof alongside the committed root and the four summary values to the principal's verification contract. On-chain verification gas runs in the range of roughly 200,000 to 300,000 gas units for a Groth16 proof on Ethereum-class chains, well within a single transaction. The principal receives a cryptographically ironclad budget attestation. The agent's operational intelligence stays dark. The trust boundary between delegation and accountability holds without a single byte of competitive information crossing it.

Supervised Versus Unsupervised Procurement: How Audit Depth Trades Off Against Agent Autonomy and Operational Throughput

The agent that processes five transactions per second under full principal supervision and the agent that streams thousands of micropayments per second with zero real-time oversight do not occupy opposite ends of a single dial. They inhabit different architectural regimes, each with its own proof obligations, latency profile, and failure surface. Treating the relationship between audit depth and agent autonomy as a smooth continuum leads to systems that either choke throughput without gaining meaningful accountability or grant autonomy without producing verifiable spend records. The distinction matters because each regime demands a structurally different composition of cryptographic primitives, and misclassification creates gaps that surface only when a dispute forces post-hoc reconstruction of the spending trail.

Fully supervised procurement places the principal directly in the transaction's hot path. Every purchase request triggers a round-trip authorization, the agent submits a proposed spend, the principal evaluates and signs, and only then does the HTLC lock proceed. This regime caps throughput at roughly two to five transactions per second because the bottleneck is principal response latency, not network capacity. The proof obligation is minimal on the agent side since the principal already witnessed and approved each action. What the regime buys is real-time intervention power. What it costs is any hope of scaling to the hundreds or thousands of discrete resource acquisitions that a compute-procurement loop executing parallel inference jobs demands. For high-value, irreversible purchases against untrusted providers, that cost is acceptable. For streaming GPU-second payments, it is fatal.

Checkpoint-supervised procurement removes the principal from individual transactions and instead gates autonomy at budget-epoch boundaries. The agent operates freely within a delegated sub-allocation, then at each checkpoint produces a ZK spend summary attesting that cumulative outlays remain within policy bounds without revealing provider identities or task content. This regime scales to hundreds of transactions per second because batching amortizes approval overhead across an entire epoch. The proof obligation shifts substantially onto the agent: it must maintain a running commitment to its spend state and produce a succinct attestation that the principal can verify in constant time. The throughput cliff between supervised and checkpoint modes is

not gradual. It arrives the moment the principal exits the per-transaction loop, and the jump is at least one order of magnitude.

Post-hoc audited procurement pushes autonomy to its structural limit. The principal never appears in the transaction path. The agent locks HTLCs, redeems L402 receipts, and settles state channel updates entirely on its own authority. Throughput scales to thousands of streaming micropayments per second because no external round-trip exists. The compensating cost is the richest proof infrastructure of all three regimes. Every payment must produce a linked preimage chain, every receipt must slot into a Merkle tree that the principal can walk during review, and budget compliance must be provable after the fact through ZK attestations covering the full spending epoch. The asymmetry is stark: as real-time intervention drops to zero, the cryptographic evidence burden grows exponentially because post-hoc verifiability must do all the work that pre-hoc approval no longer performs.

Choosing the correct regime is not a preference. It is a function of four measurable variables: the variance in per-transaction value, the trust profile of the counterparty set, whether the purchased service is reversible on dispute, and the duration of the budget epoch. An agent streaming sub-cent inference calls against a bonded provider pool with short epochs belongs in post-hoc mode. An agent committing to a multi-dollar reserved GPU block from an unknown provider with no cancellation path belongs in supervised mode. Misclassification in either direction produces concrete damage. Over-supervising a high-frequency stream throttles throughput by two orders of magnitude. Under-supervising a high-variance spend against unvetted providers creates an unauditable gap that no amount of post-hoc proof reconstruction can close because the receipts were never structured to carry the required attestation depth.

Production agents rarely operate in a single regime. A well-architected spending policy engine carries regime-transition logic that escalates from post-hoc to checkpoint-supervised the moment cumulative spend crosses a sub-allocation threshold, and from checkpoint to fully supervised when a single transaction's value exceeds a configurable ceiling. The transitions themselves must be atomic and logged: the agent's proof chain must record not only what it spent but under which regime each spend occurred, so that the principal's review process applies the correct verification logic to each segment. This hybrid deployment is not a convenience. It is the only architecture that simultaneously preserves

throughput on the low-value tail and accountability on the high-value head of a real procurement distribution.

The audit log that closes a procurement cycle is not a reporting artifact. It is the cryptographic receipt that makes every upstream composition verifiable: the channel funding that doubled as a budget commitment, the L402 credential that bound payment to access, the congestion-aware rate adjustment that kept spend within policy, the routing decision that selected a provider under latency and cost constraints simultaneously. Each of these layers was designed in isolation across earlier chapters, but their operational reality is compositional. A channel opening that does not encode budget policy is just a funding transaction. An L402 token issued without streaming settlement backing is just an API key with extra steps. The shift this chapter demands is not additive but structural: production-grade procurement exists only at the surface where cryptographic binding, economic signaling, and verifiable accountability constrain one another in real time, and it vanishes the moment any single seam reverts to manual supervision or unverified trust.

Trace your most complex agent workflow from discovery query through final audit emission. Every state transition that requires a human decision, an unverified credential rotation, or a trust assumption not backed by a cryptographic proof is a gap. Count those gaps. That number is your integration distance from production autonomy, and an agent that cannot cryptographically account for every satoshi it spent and every compute cycle it received is not autonomous. It is unsupervised. The difference is the entire protocol stack.

Conclusion

Every arrow in this diagram is load-bearing. That sentence should now land differently than it would have three hundred pages ago. Before you opened this book, you knew pieces of the stack. You understood that payment channels move value off-chain, that macaroons attenuate credentials, that HTLCs bind preimage release to payment completion, that auction mechanisms price scarce compute. What you did not have was the dependency graph that connects these primitives into a single verifiable loop, where removing any one layer breaks the guarantees of every layer above it. You have that graph now.

Look at it whole. Three architectural layers emerged across these chapters, and they are not metaphorical groupings. They are distinct failure domains with distinct invariants.

The first layer is the **settlement substrate**. Payment channels, HTLC atomicity, cross-chain timelock cascades, and zero-knowledge batch compression compose into a system that moves value at sub-millisecond latency while preserving on-chain finality as a fallback enforcement surface. The invariants here are atomicity, meaning every multi-hop payment either completes or fully reverts, and bounded settlement cost, meaning on-chain gas expenditure grows sublinearly with transaction volume through validity proofs and blob-based data availability. Chapters 1 and 2 built the channel mechanics. Chapter 8 extended atomicity across heterogeneous chains with decreasing timelock layers. Chapter 9 compressed settlement into batched proofs that amortize verification cost. If this layer fails, nothing above it is trustworthy, because every credential, every budget cap, every pricing signal ultimately denominates in a settlement event that must finalize.

The second layer is the **authorization and discovery fabric**. Macaroons with contextual caveats bind a payment hash to a specific service entitlement. L402 challenge-response flows turn HTTP 402 from a ves-

tigial status code into a machine-negotiable access gate. DHT-based registries and DNS-SD integration let autonomous agents locate providers, compare pricing metadata, and verify capability descriptors before opening a channel or cycling a credential. Chapters 3 and 5 specified these mechanisms. The invariant here is credential-payment binding, meaning no credential is valid unless a corresponding payment proof exists, and no payment releases unless the credential's caveats are satisfiable. Remove this layer and agents can pay without receiving service, or receive service without paying, and the entire commerce loop collapses into an honor system that adversarial counterparties will exploit within hours.

The third layer is the **economic control plane**. Congestion pricing functions, second-price sealed-bid auctions for scarce accelerator slots, Pigouvian surcharges on demand externalities, principal-agent budget hierarchies with task-specific subdivision and expiry enforcement, and adaptive reallocation under stochastic provider availability. Chapters 6 and 7 built this layer. The invariant is incentive compatibility, meaning no agent or provider can improve its payoff by deviating from the protocol's prescribed strategy, and budget boundedness, meaning no autonomous agent can exceed its principal's delegated spend regardless of how many providers it discovers or how aggressively it bids. Remove this layer and the settlement substrate becomes a conduit for unbounded expenditure, and the authorization fabric becomes a mechanism for accessing resources at prices that no rational principal would approve.

Chapter 10 was not a capstone. It was a proof of composition. It demonstrated that these three layers bind under production constraints: real latency distributions, real channel rebalancing costs, real credential cycling overhead, real congestion spikes. Settlement binds to authorization binds to resource access binds to economic policy in a single loop that an external auditor can trace from the initial budget delegation through every intermediate payment to the final post-hoc review. That loop is the micropayment internet. Not a network. Not a platform. A protocol composition with explicit security assumptions at every joint.

The shift that occurred in you across these pages is not knowledge accumulation. It is a change in reflex. Before this book, you could read a whitepaper describing an autonomous agent payment system and evaluate it on its stated features. Now you decompose it. You ask where the settlement path terminates and whether finality is probabilistic or deterministic at the relevant chain depth. You ask whether the credential lifecycle is bound to the payment hash or floating as a separate bearer

token with no revocation surface. You ask whether the discovery mechanism exposes the agent to Sybil-populated registries without Merkle-committed provider attestations. You ask whether the pricing model accounts for strategic demand withholding or capacity manipulation by oligopolistic providers. You ask what happens under dispute, under reorg, under budget exhaustion mid-task, under provider disappearance between credential issuance and service delivery.

This is not a philosophical orientation. It is an engineering audit capability. You moved from consumer of machine-commerce narratives to auditor of machine-commerce architectures. The difference is operational. The consumer asks whether a system sounds plausible. The auditor asks whether it clears its invariants under adversarial conditions with measurable margins.

Now deploy what you know. Three concentric rings of implementation, each independently valuable, each producing data that either confirms or falsifies the design assumptions this book specified.

First ring, within weeks: instrument a single end-to-end agent procurement loop on a testnet. One agent, one provider, one payment channel, one L402 credential cycle, one budget cap. The Chapter 10 protocol composition is your specification. Fund a channel, issue a macaroon with a spend caveat and an expiry caveat, execute a service call, verify preimage release, confirm budget decrement, close the channel cooperatively. Then inject failures. Kill the provider mid-service. Let the credential expire before the response arrives. Attempt a stale-state close. Measure what happens. The goal is not production deployment. The goal is tactile verification that the binding between settlement, authorization, and budget enforcement holds when real network latency and real process crashes intervene.

Second ring, within months: extend to multi-provider discovery and congestion-aware routing. Register at least three providers in a DHT-based registry with differentiated pricing metadata and latency SLAs. Let your agent discover, compare, and select providers based on the congestion pricing functions from Chapter 6 and the budget reallocation logic from Chapter 7. Instrument telemetry on four metrics: settlement latency per payment, credential cycling overhead per service call, budget burn rate per task, and routing fee cost per multi-hop path. Compare empirical distributions against the theoretical bounds the book specified. Where they diverge, you have found either an implementation defect or an assumption that does not survive your deployment topology. Both are valuable findings.

Third ring, within a quarter or two: integrate cross-chain settlement and ZK compression. This ring requires modeling reorg-probability risk for your specific chain pair, as Chapter 8 detailed, and computing the batch aggregation breakeven point where validity proof generation cost falls below the cumulative gas cost of individual settlements, as Chapter 9 quantified. These are not abstract exercises. They produce a cost curve that tells you at what transaction volume zero-knowledge compression becomes economically rational for your workload, and a risk threshold that tells you at what chain depth your cross-chain HTLCs achieve acceptable atomicity guarantees.

Each ring is a test plan, not a roadmap. Each produces measurable data. Each either validates or invalidates specific protocol design choices. Composability means you do not need to wait for ring three to extract production value from ring one.

You now hold the complete dependency graph of autonomous machine commerce. The settlement substrate provides finality. The authorization fabric provides binding. The economic control plane provides boundedness. Every primitive was selected because it satisfies an invariant that the layer above requires. Every invariant was specified because removing it creates a failure mode that adversarial conditions will find.

The next system you encounter that claims to enable machine-to-machine payments, you will not ask whether it works. You will ask where the binding is. Where the proof is. What happens under dispute. And you will know, within minutes, whether the architecture holds or whether it is hand-waving over the load-bearing joints that this book made visible.

That intolerance is permanent. Build on it.

Resources

Further Resources for *The Micropayment Internet*

Protocol Specifications and Standards Documents

BOLT (Basis of Lightning Technology) Specifications - The canonical protocol documents governing Lightning Network channel management, onion routing, HTLC construction, and peer messaging. Every claim in Chapters 1–4 about channel lifecycle, dispute mechanics, and multi-hop routing traces back to these specs. Reading them directly resolves ambiguities that secondary sources introduce.
https://github.com/lightning/bolts

L402 Protocol Specification (formerly LSAT) - The specification for HTTP 402-based authentication where Lightning payment proofs bind to macaroon credentials. This is the primary reference for Chapter 3's treatment of challenge-response flows and automated credential presentation. Maintained by Lightning Labs.
https://github.com/lightninglabs/L402

Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud (Birgisson et al., 2014) - The original Google Research paper defining macaroon construction, caveat attenuation, and third-party discharge. Essential for understanding why macaroons compose hierarchically without server roundtrips—the property that makes them viable for autonomous agent credential cycling.
https://research.google/pubs/pub41892/

EIP-4844: Shard Blob Transactions - The Ethereum Improvement Proposal specifying blob-carrying transactions and the data availability mechanism underlying Chapter 9's treatment of cost-efficient state diff publication. Reading the EIP directly exposes the gas accounting and blob lifecycle that determine real settlement compression economics.
https://eips.ethereum.org/EIPS/eip-4844

Perun: Virtual Payment Hubs over Cryptographic Currencies (Dziembowski et al., 2019) - The formal treatment of virtual state channels and channel factory constructions referenced in Section 2.2. Proves security under the Universal Composability framework and specifies how amortized on-chain setup costs are achieved through nested channel hierarchies.
https://eprint.iacr.org/2017/635

Sprites and State Channels: Payment Networks that Go Faster than Lightning (Miller et al., 2019) - Introduces preimage-based state channels with reduced collateral lockup by decoupling dispute resolution from individual channel close. Directly relevant to the settlement latency and liveness trade-offs analyzed in Section 2.1, and offers an alternative to BOLT-style penalty mechanisms.

RFC 9540: Discovery of Designated Resolvers (DDR) and DNS-SD (RFC 6763) - The IETF specifications for DNS-based service discovery that Section 5.2 builds upon when integrating DHT-based resource registration with DNS-SD for provider listings. Understanding the actual DNS-SD record format and resolution semantics is necessary to evaluate the discovery-to-channel pipeline's production feasibility.

Books on Cryptographic Protocols, Mechanism Design, and Distributed Settlement

Mastering the Lightning Network by Andreas Antonopoulos, Olaoluwa Osuntokun, and René Pickhardt - The most implementation-grounded book on Lightning channel mechanics, HTLC forwarding, onion routing, and fee markets. Covers the same primitives as Chapters 1–4 but with extended code-level walkthroughs. Readers who want to move from protocol description to node implementation should start here. O'Reilly Media, 2021.

Algorithmic Game Theory edited by Noam Nisan, Tim Roughgarden, Éva Tardos, and Vijay Vazirani - The standard reference for auction design, mechanism design, Nash equilibrium computation, and congestion games. Chapter 6's treatment of second-price auctions, combinatorial bidding, and Pigouvian surcharges assumes familiarity with material this book covers rigorously. Cambridge University Press, 2007.

Proofs, Arguments, and Zero-Knowledge by Justin Thaler - A technically complete treatment of interactive proofs, SNARKs, STARKs, and commitment schemes. Chapters 9's validity proofs and zero-knowledge spend compliance attestations require understanding the proof systems this book specifies. Freely available and regularly updated.
https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html

Putting Auction Theory to Work by Paul Milgrom - Goes beyond textbook mechanism design to address practical auction implementation: combinatorial allocation, incentive compatibility under real bidder behavior, and revenue optimization under capacity constraints. Directly informs the GPU/TPU slot auction analysis in Section 6.2. Cambridge University Press, 2004.

Foundations of Distributed Consensus and Blockchains by Elaine Shi - A rigorous treatment of consensus protocols, Byzantine fault tolerance, and finality guarantees. The settlement finality asymmetries and reorg-probability modeling in Chapter 8 assume the reader can reason about chain confirmation semantics at the level this book provides.
http://elaineshi.com/docs/blockchain-book.pdf

Designing Data-Intensive Applications by Martin Kleppmann - While not crypto-specific, this book provides the strongest available treatment of distributed state management, consistency models, and failure semantics. Readers building the systems described in Chapter 10 —where discovery, channel state, budget enforcement, and audit trails must cohere under partial failure—need the operational reasoning Kleppmann teaches. O'Reilly Media, 2017.

Research Papers with Direct Mechanism Relevance

"Payment Channel Networks: A Survey" (Gudgeon et al., 2020) - A comprehensive academic survey of payment channel network constructions, security models, routing algorithms, and economic analyses. Covers virtual channels, channel factories, watchtowers, and griefing attacks. Provides the broadest single-paper view of the design space underlying Chapters 1–4 and 8.
https://arxiv.org/abs/2012.09544

"Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks" (Aumayr et al., 2021) - Addresses the collateral inefficiency problem in multi-hop HTLCs by constructing atomic multi-channel updates with constant (rather than linear) collateral. Directly relevant to the cross-chain HTLC bridge topologies in Chapter 8 and the fee/liquidity analysis in Section 4.3.
https://eprint.iacr.org/2019/583

"Congestion Attacks in Payment Channel Networks" (Mizrahi and Zohar, 2021) - Formalizes how adversaries can lock liquidity across payment channel networks by initiating HTLCs they never settle. This is a concrete instance of the adversarial conditions Chapter 4 must survive. Understanding this attack surface is necessary to evaluate whether any routing and fee design is production-viable.
https://arxiv.org/abs/2002.06564

"Pricing Cloud Resources: A Market Mechanism Design Approach" (various, AWS/academic collaborations) - The body of work on spot pricing, dynamic allocation, and truthful mechanism design for cloud compute. While often published through cloud provider research arms, the mechanism design principles transfer directly to the decentralized compute market pricing in Chapter 6. Key authors include Eric Friedman and Scott Shenker.

"MAD-HTLC: Because HTLC is Crazy-Cheap to Attack" (Tsabary et al., 2021) - Demonstrates that standard HTLC constructions are vulnerable to bribery attacks where miners are incentivized to deviate from honest behavior. Proposes Mutual Assured Destruction HTLCs as a mitigation. Essential reading for anyone taking the atomicity guarantees of Chapter 8 at face value—this paper shows where those guarantees break under realistic miner incentive assumptions.
https://arxiv.org/abs/2006.12031

"Brick: Asynchronous Incentive-Compatible Payment Channels" (Avarikioti et al., 2021) - Constructs payment channels that remain secure under asynchronous network assumptions without requiring a synchrony bound for dispute resolution. This directly challenges the liveness assumptions underlying Section 2.1's settlement latency analysis and offers a cleaner failure model for autonomous agents that cannot guarantee timely chain observation.
https://eprint.iacr.org/2019/1150

Tools, Implementations, and Development Environments

LND (Lightning Network Daemon) by Lightning Labs - The most widely deployed Lightning implementation and the codebase most relevant to understanding L402 flows, channel management, and HTLC forwarding in production. Readers building against the architecture described in Chapters 1–4 and 10 will interact with LND or its APIs directly.
https://github.com/lightningnetwork/lnd

Aperture by Lightning Labs - A reverse proxy that implements L402 authentication: issues macaroons bound to Lightning invoices and gates HTTP API access behind payment. This is the closest existing production artifact to the automated credential cycling described in Section 3.3. Studying its source reveals both the design and the current limitations.
https://github.com/lightninglabs/aperture

Polar: Lightning Network Testing Environment - A desktop application for spinning up local Lightning Network clusters with multiple nodes, channels, and configurable topologies. Invaluable for testing the channel factory constructions, routing scenarios, and fee dynamics described in Chapters 2 and 4 without mainnet exposure.
https://lightningpolar.com/

Circom and SnarkJS - The dominant toolchain for building and verifying zero-knowledge circuits (Groth16, PLONK). Readers who want to implement the zero-knowledge spend compliance proofs described in Section 9.3—aggregate budget attestation without individual transaction disclosure—will prototype circuits here.
https://github.com/iden3/circom

Foundry (Forge, Cast, Anvil) - A fast Ethereum development toolkit for writing, testing, and deploying smart contracts. Relevant to Chapter 9's gas optimization analysis and Chapter 8's cross-chain HTLC bridge contracts. Foundry's gas profiling and fork-testing capabilities make it the right environment for evaluating settlement compression economics empirically.
https://github.com/foundry-rs/foundry

SimLN - A Lightning Network payment activity simulator designed for testing routing algorithms, fee structures, and liquidity dynamics under configurable network conditions. Maps directly to the probabilistic pathfinding and channel rebalancing analysis in Sections 4.2 and 4.3.
https://github.com/bitcoin-dev-project/sim-ln

Specialized Communities and Working Groups

Lightning-Dev Mailing List - The primary venue for Lightning protocol development discussion. Channel factory proposals, HTLC modifications, routing algorithm debates, and fee market design are discussed here by the engineers building the infrastructure Chapters 1–4 describe. Archives are publicly searchable.
https://lists.linuxfoundation.org/mailman/listinfo/lightning-dev

Ethereum Research Forum (ethresear.ch) - The main discussion venue for rollup design, data availability, and gas optimization research. Chapter 9's treatment of validity proofs, blob transactions, and settlement compression connects to active threads on proof aggregation, DA sampling, and L2 economics.
https://ethresear.ch/

ZK Proof Standards Working Group - An industry working group developing standards for zero-knowledge proof systems, including interoperability, security definitions, and implementation guidance. Relevant to readers evaluating the ZK settlement compression and compliance proof mechanisms in Chapter 9 for production deployment.
https://zkproof.org/

Mechanism Design for the Internet Working Group (various academic venues) - Not a single organization but a recurring community across ACM EC, WINE, and related conferences. Papers on auction design, pricing under strategic behavior, and congestion mechanisms directly inform Chapter 6. The ACM Conference on Economics and Computation proceedings are the most concentrated source.
https://ec.sigecom.org/

Bitcoin Optech - Publishes weekly technical newsletters covering Lightning protocol changes, channel management improvements, HTLC modifications, and cross-chain atomic swap developments. The most efficient way to track production-relevant protocol evolution across the primitives treated in Chapters 1, 4, and 8.
https://bitcoinops.org/

Independent Articles, Talks, and Technical Deep Dives

"The State of Lightning" Annual Reports by River Financial - Data-driven assessments of Lightning Network capacity, routing success rates, channel distributions, and fee economics. Provides the empirical baseline against which the routing optimization and fee market analysis in Chapter 4 should be evaluated.
https://river.com/learn/the-state-of-lightning/

René Pickhardt's Research on Optimal Payment Routing - Pickhardt's work on minimum-cost flow-based routing with probabilistic success modeling represents the current frontier of the pathfinding problem treated in Section 4.2. His papers and presentations formalize the modified Dijkstra approach with success-probability weighting more rigorously than any other public source.
https://ln.rene-pickhardt.de/

"Programmable Money" by Paul Engeler (Suredbits) - A technical series on L402 implementation, macaroon-based API monetization, and streaming payment integration with REST APIs. Provides implementation-level detail on the L402 flows described in Chapter 3, written by engineers who built production L402 services.
https://suredbits.com/

Vitalik Buterin's "An Incomplete Guide to Rollups" - A concise but precise treatment of optimistic and ZK rollup architectures, data availability requirements, and settlement economics. Contextualizes the validity proof and blob transaction mechanisms in Chapter 9 within the broader L2 design space.
https://vitalik.eth.limo/general/2021/01/05/rollup.html

"Channel Factories" by Christian Decker, Roger Wattenhofer, and Colleagues - The original technical presentation of channel factory constructions that enable amortized on-chain costs for multi-party channel groups. Section 2.2's treatment of factory constructions derives from this work. Available as both papers and conference presentations.
https://tik-db.ee.ethz.ch/file/a20a865ce40d40c8f942cf206a7cba96/Scalable_Funding_Of_Blockchain_Micropayment_Networks.pdf

"MEV and the Future of Transaction Ordering" by Flashbots Research - Addresses how miner/validator extractable value interacts with HTLC settlement, cross-chain atomicity, and transaction inclusion guarantees. The bribery and reorg risks discussed in Sections 8.2 and 8.3 cannot be fully understood without grasping the MEV landscape Flashbots documents.
https://writings.flashbots.net/

Economic and Market Design Foundations

"Mechanism Design: A New Approach to the Design of Regulatory and Financial Markets" by Eric Maskin - Maskin's Nobel lecture provides the theoretical foundation for the incentive-compatible mechanism design that Chapter 6 applies to compute markets. Understanding why truthful bidding matters—and when it fails—requires engaging with this work directly.

"Congestion Pricing" by William Vickrey (various papers) - Vickrey's foundational work on congestion pricing and marginal-cost tolling provides the economic theory underlying Section 6.3's Pigouvian surcharge analysis. The parallel between network congestion pricing and compute resource congestion pricing is structural, not metaphorical.

"Dynamic Pricing in the Presence of Strategic Consumers" by Aviv and Pazgal - Treats the problem of setting prices when buyers time their purchases strategically—precisely the scenario autonomous agents create when they can delay compute procurement to exploit price fluctuations. Informs the adaptive budget reallocation mechanisms in Section 7.3.

"Market Microstructure Theory" by Maureen O'Hara - The canonical treatment of bid-ask spreads, order flow, and liquidity provision in financial markets. The fee market dynamics and liquidity rebalancing protocols in Section 4.3 are structurally analogous to market-making problems this book formalizes. Wiley, 1995.

"The Economics of Internet Markets" by Levin (2011, NBER Working Paper) - A concise survey of two-sided market design, platform pricing, and network effects

References

Aumann, R. J. (1974). Subjectivity and correlation in randomized strategies. *Journal of Mathematical Economics, 1*(1), 67–96. https://doi.org/10.1016/0304-4068(74)90037-8

Biryukov, A., Khovratovich, D., & Pustogarov, I. (2014). Deanonymisation of clients in Bitcoin P2P network. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 15–29. https://doi.org/10.1145/2660267.2660379

Burchert, C., Decker, C., & Wattenhofer, R. (2018). Scalable funding of Bitcoin micropayment channel networks. *Royal Society Open Science, 5*(8), 180089. https://doi.org/10.1098/rsos.180089

Buterin, V. (2014). *A next-generation smart contract and decentralized application platform* [White paper]. Ethereum Foundation. https://ethereum.org/en/whitepaper/

Chervinski, J. O., Kreutz, D., & Yu, J. (2022). Analysis of transaction handling in Bitcoin. *IEEE Access, 10*, 60750–60773. https://doi.org/10.1109/ACCESS.2022.3180663

Clarke, E. H. (1971). Multipart pricing of public goods. *Public Choice, 11*(1), 17–33. https://doi.org/10.1007/BF01726210

Decker, C., & Wattenhofer, R. (2015). A fast and scalable payment network with Bitcoin duplex micropayment channels. *Lecture Notes in Computer Science, 9212*, 3–18. https://doi.org/10.1007/978-3-319-21741-3_1

Dingledine, R., Mathewson, N., & Syverson, P. (2004). Tor: The second-generation onion router. *Proceedings of the 13th USENIX Security Symposium*, 303–320. https://www.usenix.org/legacy/publications/library/proceedings/sec04/tech/decker.html

Dziembowski, S., Faust, S., & Hostáková, K. (2018). General state channel networks. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 949–966. https://doi.org/10.1145/3243734.3243856

Ethereum Foundation. (2024). *EIP-4844: Shard blob transactions*. Ethereum Improvement Proposals. https://eips.ethereum.org/EIPS/eip-4844

Goldwasser, S., Micali, S., & Rackoff, C. (1989). The knowledge complexity of interactive proof systems. *SIAM Journal on Computing, 18*(1), 186–208. https://doi.org/10.1137/0218012

Green, M. D., & Miers, I. (2017). Bolt: Anonymous payment channels for decentralized currencies. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 473–489. https://doi.org/10.1145/3133956.3134093

Gudgeon, L., Moreno-Sanchez, P., Roos, S., McCorry, P., & Gervais, A. (2020). SoK: Layer-two blockchain protocols. *Lecture Notes in Computer Science, 12059*, 201–226. https://doi.org/10.1007/978-3-030-51280-4_12

Guo, Y., Hu, Q., Hao, F., & Liu, J. (2023). A survey on payment channel networks: Technical advances, challenges, and opportunities. *ACM Computing Surveys, 55*(14s), 1–40. https://doi.org/10.1145/3590174

Herlihy, M. (2018). Atomic cross-chain swaps. *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, 245–254. https://doi.org/10.1145/3212734.3212736

Johari, R., & Tsitsiklis, J. N. (2004). Efficiency loss in a network resource allocation game. *Mathematics of Operations Research, 29*(3), 407–435. https://doi.org/10.1287/moor.1040.0091

Kalodner, H., Goldfeder, S., Chen, X., Weinberg, S. M., & Felten, E. W. (2018). Arbitrum: Scalable, private smart contracts. *Proceedings of the 27th USENIX Security Symposium*, 1353–1370. https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner

Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media.

Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., & Ravi, S. (2017). Concurrency and privacy with payment-channel networks. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 455–471. https://doi.org/10.1145/3133956.3134096

Maymounkov, P., & Mazières, D. (2002). Kademlia: A peer-to-peer information system based on the XOR metric. *Lecture Notes in Computer Science, 2429*, 53–65. https://doi.org/10.1007/3-540-45748-8_5

McCorry, P., Bakshi, S., Bentov, I., Meiklejohn, S., & Miller, A. (2019). Pisa: Arbitration outsourcing for state channels. *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 16–30. https://doi.org/10.1145/3318041.3355461

Miller, A., Bentov, I., Bakshi, S., Kumaresan, R., & McCorry, P. (2019). Sprites and state channels: Payment networks that go faster than Lightning. *Lecture Notes in Computer Science, 11598*, 508–526. https://doi.org/10.1007/978-3-030-32101-7_30

Nisan, N., Roughgarden, T., Tardos, É., & Vazirani, V. V. (Eds.). (2007). *Algorithmic game theory*. Cambridge University Press. https://doi.org/10.1017/CBO9780511800481

Osuntokun, O., Akselrod, A., & Posen, J. (2016). *The Bitcoin Lightning Network: BOLT specification* (BOLT #1–#11). Lightning Network Developers. https://github.com/lightning/bolts

Pickhardt, R., & Richter, S. (2021). Optimally reliable and cheap payment flows on the Lightning Network. *Proceedings of the 2021 IEEE International Conference on Blockchain and Cryptocurrency*, 1–9. https://doi.org/10.1109/ICBC51069.2021.9461130

Pigou, A. C. (1920). *The economics of welfare*. Macmillan.

Poon, J., & Dryja, T. (2016). *The Bitcoin Lightning Network: Scalable off-chain instant payments* [White paper]. https://lightning.network/lightning-network-paper.pdf

Prihodko, P., Zhigulin, S., Sahno, M., Ostrovskiy, A., & Osuntokun, O. (2016). *Flare: An approach to routing in Lightning Network* [White paper]. https://bitfury.com/content/downloads/whitepaper_flare_an_approach_to_routing_in_lightning_network.pdf

Robinson, D., & Konstantopoulos, G. (2019). Ethereum is a dark forest. *Paradigm Research*. https://www.paradigm.xyz/2020/08/ethereum-is-a-dark-forest

Roughgarden, T. (2005). *Selfish routing and the price of anarchy*. MIT Press.

Sivaraman, V., Venkatakrishnan, S. B., Ruan, K., Negi, P., Yang, L., Mittal, R., Fanti, G., & Alizadeh, M. (2020). High throughput cryptocurrency routing in payment channel networks. *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, 777–796. https://www.usenix.org/conference/nsdi20/presentation/sivaraman

Szabo, N. (1997). Formalizing and securing relationships on public networks. *First Monday, 2*(9). https://doi.org/10.5210/fm.v2i9.548

Teutsch, J., & Reitwießner, C. (2019). *A scalable verification solution for blockchains* [White paper]. TrueBit. https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf

Thyagarajan, S. A. K., Bhat, A., Malavolta, G., Döttling, N., Kate, A., & Schröder, D. (2022). Verifiable timed signatures made practical. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 1733–1747. https://doi.org/10.1145/3548606.3560585

Vickrey, W. (1961). Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance, 16*(1), 8–37. https://doi.org/10.2307/2977633

Zamyatin, A., Harz, D., Lind, J., Panez, H., Gervais, A., & Knottenbelt, W. (2019). XCLAIM: Trustless, interoperable, cryptocurrency-backed assets. *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, 193–210. https://doi.org/10.1109/SP.2019.00085