The Micropayment Internet

Teneo

Teneo.io

Contents

Introduction

Kira finds the cluster in São Paulo at 03:47:12 UTC. Forty milliseconds less round-trip latency than its current provider in Virginia, spot pricing roughly 18% below the contract rate its upstream callers are subsidizing, and available capacity sufficient to absorb the next estimated four hours of inference demand. Kira's evaluation takes eleven milliseconds. The routing decision is trivial. The economic decision is obvious.

Kira cannot make it.

Its compute is tethered to a single AWS billing account that a platform engineer named Daniel provisioned nineteen days ago. The account credentials are stored in a secrets manager Daniel controls. The billing relationship is a monthly invoice cycle between two legal entities, negotiated over email, governed by a master services agreement neither Kira nor any other process in the system has ever parsed. To switch providers, someone would need to wake Daniel up, or wait for business hours in his timezone, then provision new credentials, negotiate a new contract with the São Paulo provider, configure a new billing profile, update the secrets manager, and redeploy. By the most optimistic estimate, that process takes somewhere between six hours and two weeks. The arbitrage window Kira identified will close and reopen thousands of times before a human hand touches a keyboard.

This is the state of autonomous machine intelligence in the current year. A system that can evaluate sixteen providers in a millisecond cannot spend a tenth of a cent without human permission. An agent sophisticated enough to reason about latency-weighted cost surfaces across continents is economically tethered to a billing account it did not create, cannot modify, and has no authority to replace. Kira's autonomy is a polite fiction. It is a dependent process wearing the costume of an independent agent.

The instinct is to treat this as an integration problem. Wire up more APIs. Give the agent a credit card number. Store broader credentials. But the deficiency is not in Kira's configuration. The deficiency is structural, and it lives one layer beneath every API, every billing dashboard, every payments SDK you have ever integrated against. The internet has no native value-transfer primitive operating at information-layer granularity.

Consider what the internet does have. It has a native primitive for addressing: IP. It has a native primitive for reliable ordered delivery: TCP. It has a native primitive for naming: DNS. It has a native primitive for document retrieval: HTTP. Each of these operates at the granularity of the information layer itself, packets and requests and records, and each is composable with the layers above and below it without requiring a human to mediate the interaction. But value transfer? The internet punts. Every payment that crosses the wire today passes through a custodial intermediary, a bank, a payment processor, a platform billing system, that operates on human timescales, human contract structures, and human trust assumptions. The minimum viable transaction is not a fraction of a cent for a single inference call. It is a monthly subscription, a prepaid credit balance, a negotiated enterprise contract. The granularity mismatch between what machines can consume and what the payment layer can express is not merely inconvenient. It is the single structural deficiency that prevents machines from achieving genuine economic autonomy.

This book builds the missing layer.

Not as speculation. Not as a whitepaper sketching what might exist in five years. Every primitive described in these pages is deployed in production or deployable from existing open-source implementations. The Lightning Network's channel state machine, implemented in LND and Core Lightning, already processes off-chain balance updates at millisecond latency with near-zero marginal cost. Hash time-locked contracts already enable trust-free conditional payments routed across multiple intermediary nodes without any participant needing to trust any other. The L402 protocol, evolved from the original LSAT specification and enforced by tools like Aperture, already binds Lightning invoice settlement to HTTP authentication, turning a proof-of-payment preimage into a bearer credential that grants API access. Macaroon-based authorization already supports attenuated delegation, where a root credential can be progressively narrowed with spending limits, temporal bounds, and service-scope restrictions, then handed to a sub-agent that can act

within those constraints without ever seeing the root key. These are not roadmap items. They are shipped code. The problem is that no single reference assembles them into the coherent vertical stack that machine-to-machine commerce demands.

That is what you are holding.

The book makes and defends a single claim: autonomous machine commerce requires a complete cryptographic payment stack operating at packet granularity, from hash-locked commitments and streaming channel updates through hierarchical budget delegation and L402-gated authentication, and every layer of that stack is buildable today. The architecture is specified layer by layer, each layer exposing a precise interface to the one above it, each layer verified under adversarial conditions before the next is constructed on top of it. If you have built or debugged systems at the protocol layer, the structure will feel familiar. It mirrors the way you already think about composable infrastructure: define the primitive, specify the state transitions, characterize the failure modes, prove the security properties under explicit rationality assumptions, then expose the interface and move up.

We follow Kira throughout. Not as a mascot but as a forcing function. Each chapter solves a specific dimension of Kira's economic helplessness, and each chapter ends with Kira measurably more autonomous than it was at the start.

Part I establishes the problem space with precision. The internet's missing payment primitive is not a philosophical observation. It is an architectural gap with concrete consequences: every agent that cannot independently discover, negotiate, pay for, and verify delivery of a computational resource is dependent, regardless of its cognitive sophistication. Autonomy without economic agency is a category error, and Part I makes that case at the mechanism level.

Part II constructs the stack from the bottom up. Hash-function commitment schemes. The 2-of-2 multisig funding output and the channel state machine that enables signed balance updates to settle peer-to-peer in milliseconds. HTLC construction and multi-hop routing, where decreasing time-lock layering prevents race conditions across intermediary nodes. Macaroon-based attenuated delegation, where HMAC-chained caveats encode spending limits and service-scope restrictions into bearer credentials that narrow permissions downward without exposing the root key. The L402 protocol flow, from the server's HTTP 402 challenge through Lightning invoice settlement to authenticated API access, with automated budget enforcement embedded

at the request layer. Service discovery via Kademlia DHTs keyed by capability hashes. Game-theoretic resource allocation with explicit Nash equilibrium analyses of agent bidding populations. Programmable micro-budget hierarchies with per-task scoping and exhaustion fallbacks. Each chapter gives Kira a new capability. Each capability composes with the ones below it.

Part III addresses what separates production-grade systems from conference demos: failure modes, adversarial conditions, and the tradeoff surfaces you must navigate under real operational constraints. Timeout griefing and locked-liquidity costs in multi-hop HTLC chains. Collusion detection in agent auction populations. The settlement-latency versus capital-efficiency versus on-chain-footprint tradeoff that determines whether a given payment should resolve in a state channel, on an L2 rollup, or on the base layer. The engineering patterns that make a micropayment stack survivable rather than merely functional.

By Chapter 10, the full stack operates end-to-end. Kira discovers the São Paulo provider through a DHT query keyed by capability hash. It evaluates pricing against a marginal-cost curve and enters an auction for an inference slot. It opens a payment channel with a hash-locked commitment, streams fractional-cent payments per inference token as results arrive, authenticates each request via an L402 credential whose embedded macaroon caveats enforce a per-session spending ceiling delegated from a higher-level budget envelope. If the provider underdelivers, Kira holds a chain of cryptographic receipts, signed balance updates that constitute proof of payment for work not rendered, and it can redirect spend to an alternative provider within the latency budget of the original request. No human woke up. No billing dashboard was consulted. No custodial intermediary took a cut. No contract was negotiated over email.

That version of Kira is not science fiction. Every primitive it needs already exists. The only thing missing is the blueprint for assembling them into a coherent stack.

Turn the page. We are going to build it.

Chapter One

Cryptographic Primitives for Payment Verification

Kira's inference router spots the listing at timestamp t and has roughly two hundred milliseconds to claim it. A bare API endpoint, no brand behind it, is advertising spare H100 capacity at a price that undercuts the current cloud contract by forty percent. The economics are obvious. The problem is everything else. No shared jurisdiction, no legal contract, no human on either side to phone a bank or file a dispute. One piece of software must transfer value to another piece of software and receive a cryptographic proof that the transfer occurred, all before the slot expires and the next bidder takes it.

The striking thing is not that this transaction is difficult. It is that every other interaction at the application layer, every packet routed, every DNS name resolved, every TLS handshake completed, already operates at this speed and at this level of mutual anonymity. The internet solved authentication, confidentiality, and routing between strangers decades ago. What it never built was a native mechanism for one endpoint to prove to another that payment has been made, without trusting an intermediary to say so. That missing primitive is what turns a two-hundred-millisecond compute auction into an unsolvable problem for any agent relying on conventional payment rails.

The answer to that two-hundred-millisecond problem starts with the simplest and most consequential primitive in the entire payment stack: a hash function, and specifically, the property that makes it useful for money rather than mere data integrity.

Hash Functions, HMAC Chains, and Preimage Commitment Schemes as Settlement Foundations

Kira has located a GPU provider she has never encountered before. No shared institution vouches for either party, no reputation ledger bridges the gap, and the provider's compute slot will expire in seconds. The entire transaction must close on a single shared primitive: a value one side can generate in microseconds, commit to without revealing, and later expose as an irrevocable proof that payment conditions have been met. That primitive is a hash preimage, and the asymmetry it exploits is not institutional but computational. Producing a 32-byte secret is trivial; inverting its hash is, under standard assumptions, infeasible. This one-way gap converts a problem of mutual distrust into a problem of message ordering, solvable by protocol choreography alone.

But a single hash lock settles a single obligation. Kira needs to stream payments across hundreds of incremental delivery steps, each verified independently, each releasing exactly the funds earned so far and no more. The construction that makes this possible takes one root secret and derives an ordered sequence of commitments whose reveal order is the reverse of their generation order, so that each step's proof implicitly covers every prior step. Getting the choreography wrong by even one message inverts the security model entirely, granting the counterparty a claim it has not earned. The sections ahead specify how collision resistance, preimage resistance, and second-preimage resistance function as load-bearing properties of this settlement substrate, how HMAC chains construct ordered commitment sequences from a single root, and how preimage revelation operates as the atomic trigger that converts a cryptographic secret into a settled payment.

Collision Resistance, Preimage Resistance, and Second-Preimage Resistance as Load-Bearing Security Properties

A 256-bit string costs nothing to generate. Its SHA-256 digest costs nothing to compute. But the relationship between those two values, the computational asymmetry that makes one trivial to derive from the other and the reverse practically impossible, is the most consequential property in the entire payment stack. Every settlement proof, every condi-

tional transfer, every chained micropayment that this book constructs rests on three specific guarantees a hash function provides. These guarantees are not academic taxonomies. They are load-bearing structural members, and understanding exactly how each one bears weight is prerequisite to everything that follows.

Preimage resistance is the property that makes a hash digest function as a commitment. When Kira generates a random 256-bit secret r and computes $h = \mathrm{H}(r)$, sharing h with an unknown GPU provider commits Kira to a specific future action: revealing r. The provider can verify $\mathrm{H}(r) = h$ in microseconds, but cannot work backward from h to discover r before Kira chooses to disclose it. This asymmetry transforms a cheap random number into a binding obligation. The commitment is durable precisely because no polynomial-time adversary can invert H with non-negligible probability. In settlement terms, h is the lock and r is the key, and only the party who generated the secret holds the key. The moment Kira reveals r, the provider possesses an artifact that anyone can independently verify against the original h. No intermediary adjudicates. No shared account exists. The hash preimage itself is the settlement receipt, and preimage resistance is what prevented premature disclosure from the provider's side.

Second-preimage resistance carries a different burden. Suppose the provider, upon receiving h, could find some $r' \neq r$ such that $\mathrm{H}(r') = h$. The provider could then fabricate a settlement proof, claiming payment occurred when Kira never authorized one. Second-preimage resistance forecloses this attack. For a given input r and its digest h, finding any distinct r' that maps to the same h is computationally infeasible. This property protects the payer. It ensures that only the original secret holder can produce a valid preimage, making the proof of payment non-repudiable in one direction and unforgeable in the other.

Collision resistance guards the system at a higher level of abstraction. An adversary who can find any pair (x, x') with $\mathrm{H}(x) = \mathrm{H}(x')$ and $x \neq x'$ can construct ambiguous commitments, ones that resolve to two different payment states depending on which preimage gets revealed. In a chained micropayment context, where each link in an HMAC chain represents an incremental value transfer, a collision would let a dishonest party equivocate about how many increments were authorized. Collision resistance makes each hash output a unique fingerprint for its input within the operational security parameter, eliminating ambiguity from the commitment space entirely.

These three properties compose into something greater than their sum. A single preimage lock, $h = H(r)$, gives you an atomic conditional payment: value moves if and only if r is disclosed. Extend this with an HMAC chain, where $h\_n = HMAC\_k(h\_\{n-1\})$, and you get an ordered sequence of commitments. Each revealed link proves an incremental payment beyond the last, enabling streaming settlement. The provider who holds h_3 knows exactly three increments have been authorized, can verify the chain's integrity back to the root, and cannot skip ahead to claim h_4 without Kira's explicit disclosure. No new on-chain transaction is needed per increment. The chain itself is the ledger.

Consider the distance traveled. Before this construction, Kira paid a GPU provider through an opaque billing API. Settlement was an invoice processed by a human-configured account, and neither party held a cryptographic proof the other could independently verify. Now Kira generates r, computes $h = H(r)$, and shares h as a payment condition. When the provider delivers a valid inference result, Kira reveals r. The provider holds r and can prove $H(r) = h$ to anyone, at any time, without contacting Kira or any third party. The receipt is self-verifying. The settlement is final at the moment of disclosure. And the entire construction requires nothing beyond a hash function that keeps its three promises.

What this construction cannot yet do is bind a commitment to a specific identity. The preimage proves that a payment event occurred. It does not prove who authorized it, or prevent the commitment from being claimed by an unintended party. That binding requires a different primitive entirely.

HMAC Chain Construction: Deriving Ordered Commitment Sequences from a Single Root Secret

When Kira's orchestration daemon first queried the GPU provider at the far end of an unfamiliar relay, it possessed nothing but a 256-bit random secret drawn from a cryptographically secure source. From that single seed, it derived an entire settlement sequence: an ordered chain of hash commitments, each one linked to the next by a single application of HMAC-SHA256, each one representing a discrete increment of value that could be revealed on demand and verified in constant time. No certificate authority vouched for the provider. No billing API mediated the exchange. The chain itself was the payment instrument, and its construction from a lone root secret is what made the arrangement possible.

The procedure begins with a seed value s selected uniformly at random. The payer computes $h_0 = HMAC(k, s)$ using a key k derived from session context, then iterates: $h_1 = HMAC(k, h_0)$, $h_2 = HMAC(k, h_1)$, and so on until reaching a terminal value h_n for some predetermined n that represents the maximum number of payment increments the session can support. The payer transmits h_n to the payee as a commitment. This terminal hash is the anchor. The payee cannot invert it. It is a binding promise that a chain of exactly n preimages exists beneath it, each recoverable only by the entity that holds s. The direction of construction runs forward from seed to terminal hash, but the direction of revelation runs backward: the payer reveals h_{n-1} first, then h_{n-2}, then h_{n-3}, each time exposing one more link in the chain. The payee verifies each reveal with a single hash computation, checking that $HMAC(k, h_{i-1}) = h_i$ where h_i is the last value already accepted. Verification cost is one HMAC evaluation per increment, independent of chain length.

This asymmetry between construction cost and verification cost is not incidental. It is the structural property that makes streaming settlement feasible. The payer performs n HMAC evaluations once, during setup. The payee performs exactly one evaluation per payment increment, in real time, as service is delivered. For Kira's daemon purchasing GPU cycles in sub-second intervals, the verification latency sits comfortably in the microsecond range on commodity hardware. No round-trip to a ledger, no confirmation window, no third-party adjudication. The hash chain's monotonic structure provides an additional guarantee that matters deeply in adversarial settings: the latest revealed preimage always subsumes all prior ones. If the payee holds h_i, it can prove the existence of every preimage from h_{n-1} down to h_i simply by presenting h_i itself. There is no ambiguity about which party owes what. The chain collapses the entire payment history into a single value, the lowest-index preimage yet revealed, and that value constitutes the complete settlement state.

The two-phase structure of this arrangement deserves careful attention. In the commitment phase, the payer publishes h_n and the payee accepts it as a cryptographic lock. Neither party trusts the other. The payee knows only that if a valid preimage appears, it was produced by someone who possesses the chain. The payer knows only that h_n alone grants the payee nothing spendable. In the reveal phase, preimages flow from payer to payee in exchange for service delivery. Each reveal is irrevocable. Once the payee observes h_i and confirms $HMAC(k, h_i) = h_{i+1}$, the payment for that increment is settled with the same finality as a mathematical proof:

the preimage either hashes correctly or it does not. There is no partial correctness, no probabilistic confidence, no reputation score to consult. The hash function's preimage resistance converts a reveal into a receipt that any third party can independently verify at any later time using only the commitment h_n and the revealed value.

What emerges from this construction is a self-contained trust engine operating entirely at the hash-and-preimage layer. It requires no digital signatures, no on-chain transactions, no channel state machinery. Two machines that have never interacted before can, with one shared commitment and a sequence of reveals, conduct an arbitrarily granular value exchange where every increment is individually verifiable and collectively monotonic. The settlement foundation is complete at this layer. What it lacks, and what the next construction will supply, is the binding of identity to commitment: the ability to prove not just that a valid preimage exists, but that a specific party authorized its use.

Preimage Revelation as Atomic Settlement Trigger: From Hash Lock to Payment Proof

Kira generates a 256-bit random secret, computes its SHA-256 digest, and transmits that 32-byte hash to a GPU provider she has never contacted before. In that single act, she constructs a binding payment commitment without transferring funds, without revealing the secret, and without invoking any intermediary. The hash digest functions as a lock. The preimage functions as the key that releases value. When Kira later discloses the secret, the provider hashes it locally, confirms the output matches the committed digest, and treats the match as irrefutable proof of payment. No adjudication is required. No third-party oracle confirms the event. The settlement is atomic: either the preimage matches and value transfers, or it does not and nothing moves. This is the mechanism that converts a hash function's one-wayness from a security property into an economic instrument.

The construction's power rests on a precise asymmetry. Before revelation, the payer holds the preimage and the payee holds only the hash. The payee cannot invert H(x) to recover x. The payer cannot deny having committed, because the hash was shared and is deterministic. At the moment of reveal, knowledge transfers irreversibly. The payee now possesses x, can prove possession to any observer by presenting the pair (x, H(x)), and the payer cannot retroactively revoke what has been disclosed. This non-repudiability is not a policy guarantee enforced by a platform's terms of service. It is a mathematical invariant of SHA-256's

preimage resistance, holding for any two parties regardless of jurisdiction, identity, or prior relationship. For machines operating across trust boundaries, this distinction matters enormously. A hash-locked payment is self-enforcing in the strongest possible sense: the proof of payment and the act of payment are the same event.

Now extend this atom into a sequence. Kira selects a root secret s_n, computes a hash chain by iterating: s_{n-1} = H(s_n), s_{n-2} = H(s_{n-1}), continuing until she reaches s_0. She sends s_0 to the GPU provider as the chain's anchor along with n, the total number of links representing her maximum payment budget for the session. Each link corresponds to a fixed increment of value, say one satoshi per inference slice. Kira reveals preimages in reverse order of generation. She discloses s_1 first, and the provider verifies H(s_1) = s_0. Then s_2, verifying H(s_2) = s_1. Each successive reveal ratchets the cumulative payment forward by one increment. The provider only needs to store the most recently revealed preimage, since each one commits to the entire chain before it. If Kira stops revealing after the fourteenth preimage, exactly fourteen increments have settled. No partial states need reconciliation. No round-trip negotiation occurs per increment, because the verification is local and the chain's ordering is cryptographically determined. The settlement ladder is fully deterministic: the chain's structure replaces the interactive protocol that two parties would otherwise need to agree on a running balance.

The operational consequence is that Kira's cloud billing account becomes unnecessary for this payment relationship. The hash chain carries roughly 100 bytes of initial setup data, generates verification overhead of a single SHA-256 computation per increment, and settles each microslice in under a microsecond on commodity hardware. Contrast this with a custodial API call that traverses a payment gateway, incurs per-transaction fees that dwarf the value of a sub-cent payment, and requires Kira's human operator to have pre-authorized the billing credential. The hash chain collapses that entire dependency stack into a cryptographic object that two machines can construct, verify, and settle between themselves. The payee's exposure is bounded: at any moment, the worst-case loss is a single unrevealed increment, because every previously revealed preimage is already a completed settlement.

What remains unaddressed is identity. The hash lock proves that someone revealed the preimage, but not that Kira specifically authorized the payment. The preimage is a bearer instrument in its raw form. For a complete payment proof, a mechanism must bind the hash commit-

ment to a specific agent's authorization, confirming not just that value moved but that a particular principal sanctioned the transfer. That binding requires a construction layered on top of the hash lock, one that ties the commitment to a verifiable cryptographic identity.

Digital Signatures, Multisig Constructions, and the 2-of-2 Funding Output

A hash preimage is a perfect proof of knowledge and a terrible proof of authorization. Kira's GPU provider reveals the preimage to claim payment, but every node that relayed the message now holds an identical bitstring, and nothing in the hash lock itself binds that secret to Kira's transaction rather than to a replayed copy submitted by an eavesdropper. The preimage proves that *someone* knew the secret. It does not prove that *Kira* committed funds, that *this specific channel* was the intended context, or that the revealed value can only be redeemed once along a single path. Without a primitive that ties a commitment to a specific key holder, hash locks are cryptographic puzzles floating free of any enforceable economic consequence.

Digital signatures close that gap by producing a proof that is simultaneously unforgeable, non-repudiable, and bound to the exact message bytes being authorized. But a single signature from a single key only proves one party's intent. The structural problem Kira faces is not just authentication but *custody*: two machines that have never interacted need to lock funds into a shared output that neither can seize unilaterally. This is the 2-of-2 multisignature construction, and its deliberate mutual paralysis is not a flaw to be engineered around. It is the precise condition that makes trustless escrow possible between strangers with no reputation, no legal identity, and no recourse beyond the protocol itself.

Schnorr and ECDSA Signature Verification: Binding Transaction Authorization to Private Key Possession

A hash commitment binds a value to a future revelation, but it says nothing about who performed the binding. Any party who discovers the preimage can satisfy the hash lock. To convert this anonymous proof of knowledge into a specific authorization, a different primitive is required: one that ties a particular identity to a particular message and does so in a way that no other party can forge. This is the role of the digital signature. Where a hash function maps arbitrary data to a fixed-length digest, a signature scheme maps a private key and a message to a value that any-

one holding the corresponding public key can verify but no one lacking the private key can produce.

In both ECDSA and Schnorr constructions deployed on Bitcoin, the mechanics differ but the security guarantee converges. A signer possessing private key sk computes a signature $\sigma(m, sk)$ over a transaction message m. Any verifier holding the corresponding public key pk evaluates a verification function $V(m, \sigma, pk)$ that returns true if and only if the signature was produced by the holder of sk over that exact message. The binding is bidirectional: the signature is invalid for any other message, and any other private key produces a signature that fails verification against pk. This is the primitive that converts identity into authority over funds. A UTXO locked to a public key can only be spent by the entity that controls the corresponding private key, because only that entity can produce a signature the network's consensus rules will accept. No trusted third party adjudicates the claim. The elliptic-curve discrete logarithm problem ensures that extracting sk from pk remains computationally infeasible, so possession of the private key is both necessary and sufficient for spending authorization.

But a single signature protects a single-party output. When two parties must share control over funds without trusting each other, the primitive must compose. A 2-of-2 multisignature output locks coins behind a script that demands two valid signatures, one from each participant's public key. Neither key alone satisfies the spending condition. This is not merely a stronger lock. It introduces a bilateral veto: every transaction that moves funds from the shared output requires cooperative agreement from both parties. A 1-of-2 construction would allow either party to drain the output unilaterally, offering no protection. A 2-of-3 construction introduces a third signer whose role and trust assumptions complicate the model beyond what bilateral commerce requires. The 2-of-2 is the minimal structure that prevents unilateral spending while preserving the ability for both parties to cooperatively update the balance at any time.

Consider Kira and an unknown GPU provider, each holding independent key pairs. Before the funding output exists, either could construct and broadcast any transaction it likes. There is no shared structure constraining their behavior. Now they cooperate to build a funding transaction whose single output is a P2WSH script encoding `OP_2 <pk_Kira> <pk_provider> OP_2 OP_CHECKMULTISIG`. Once this transaction confirms on chain, the deposited coins are unreachable by either party alone. Kira cannot move the funds without the pro-

vider's signature, and the provider cannot move them without Kira's. The on-chain anchor is established. From this point forward, any redistribution of the locked balance requires both parties to co-sign a new spending transaction. The bilateral veto transforms two independent agents into a constrained bilateral system where cooperative updates are possible and unilateral theft is cryptographically excluded.

This funding output is not an end state. It is a prerequisite. Locking funds into a structure that neither party can unilaterally spend creates an immediate problem: what if one party disappears or refuses to cooperate? The locked funds would remain frozen indefinitely. The solution demands a new construction, one that spends from the 2-of-2 output before it is broadcast, pre-signing transactions that allocate the balance according to the latest agreed state. That construction, and the revocation mechanism it requires, is the commitment transaction.

Constructing the 2-of-2 Multisig Output: Joint Custody Without Unilateral Spending Authority

In early 2019, a Lightning Network developer named Lisa Neigut sat in front of a terminal watching two nodes negotiate a channel open. Both nodes held valid keys. Both broadcast valid signatures. Yet the funding transaction failed because one implementation serialized the public keys in compressed DER format while the other used uncompressed encoding, producing different witness scripts and therefore different output addresses. The channel never opened. The episode crystallized a principle that every payment channel engineer internalizes: a 2-of-2 multisig output is not merely two keys placed side by side. It is a precisely ordered, deterministically constructed spending policy where every byte of the witness script must be identical on both sides before a single satoshi is committed. What follows is the procedure for constructing that shared-custody output from raw key material, producing the funding transaction that neither party can spend alone, and verifying that the resulting scriptPubKey creates the economic gravity well from which all subsequent off-chain state updates derive their force.

Step 1: Exchange and Canonically Order Both Parties' Public Keys

Before any funding transaction can be constructed, both participants must exchange their funding public keys and agree on a single canonical ordering. In the Lightning Network's BOLT #3 specification, the two compressed public keys (33 bytes each, SEC encoding over secp256k1) are sorted lexicographically in ascending byte order. This ordering is not arbitrary. It eliminates ambiguity: both parties independently derive the identical witness script from the same two keys without any further coordination. If Kira's compressed public key `P_K` is `02abc...` and the GPU provider's key `P_G` is `03def...`, the sorted pair is `[P_K, P_G]` because `0x02 < 0x03`. The sort happens over raw byte arrays, not over any higher-level encoding. Both parties must validate that the received key is a valid point on secp256k1 before proceeding. An invalid or maliciously crafted key would produce a witness script that is either unspendable or spendable by only one party, violating the joint-custody invariant at the root.

1. Transmit your compressed public key `P_local` to the counterparty via the `open_channel` or `accept_channel` message.
2. Receive the counterparty's compressed public key `P_remote` and verify it lies on the secp256k1 curve by checking that `P_remote` satisfies $y^2 = x^3 + 7 \pmod{p}$.
3. Sort `[P_local, P_remote]` lexicographically by raw byte value to produce the ordered pair `[P_1, P_2]`.

Step 2: Construct the 2-of-2 Witness Script and Derive the P2WSH Output

With the ordered key pair `[P_1, P_2]` established, both parties independently construct the identical witness script. For a native SegWit 2-of-2 multisig, the script is: ``` OP_2 <P_1> <P_2> OP_2 OP_CHECKMULTISIG ``` This script encodes a spending policy: exactly two valid ECDSA signatures (or, under Taproot, Schnorr signatures via MuSig2) corresponding to `P_1` and `P_2` must be presented in the witness stack for the output to be spent. The `OP_2 ... OP_2 OP_CHECKMULTISIG` construction is the m-of-n case where m = n = 2, the degenerate threshold that eliminates unilateral spending authority entirely. To produce the on-chain output, hash the serialized witness script with SHA-256 to obtain the witness program `H = SHA256(witnessScript)`. The resulting scriptPubKey is `OP_0 <H>`, a standard Pay-to-Witness-Script-Hash (P2WSH) output. Both parties must compute this independently and confirm that their derived scriptPubKey matches before the funder broadcasts.

1. Serialize the witness script: `0x52 || 0x21 || P_1 || 0x21 || P_2 || 0x52 || 0xae`.
2. Compute `witness_program = SHA256(serialized_witness_script)`.
3. Construct the scriptPubKey: `OP_0 <32-byte witness_program>`, yielding a standard P2WSH output.
4. Compare your locally derived scriptPubKey with the counterparty's. If they differ, abort. A mismatch means the two sides would be signing over different transactions.

Step 3: Build and Sign the Funding Transaction

The funding transaction is a standard Bitcoin transaction whose single relevant output is the P2WSH address derived in the previous step. One or both parties contribute inputs. In the typical Lightning channel open, the initiator (Kira, in our running example) funds the transaction entirely, but dual-funded channels allow both sides to contribute. The funding output amount defines the channel's total capacity. Critically, the funding transaction must not be broadcast yet. Broadcasting before obtaining a signed commitment transaction would lock funds into an output that requires both signatures to spend, with no pre-signed fallback. If the counterparty becomes unresponsive after broadcast, the funds are irrecoverably frozen. The correct sequence is: construct the funding transaction, exchange it unsigned, then negotiate and sign the first commitment transaction that returns funds to both parties, and only then broadcast the funding transaction.

1. Select UTXOs sufficient to cover the desired channel capacity plus on-chain fees.
2. Construct the funding transaction with the P2WSH output at the agreed-upon amount.
3. Compute the funding transaction's txid (the hash of the transaction without witness data, per BIP 141).
4. Do **not** broadcast. Hold the signed funding transaction locally until a valid first commitment transaction exists.

Step 4: Negotiate the Initial Commitment Transaction Before Broadcasting

With the funding txid known, both parties construct the first commitment transaction. This transaction spends the 2-of-2 funding output and distributes the full balance back to the funder (since no payments have occurred yet). Each party holds their own version of this commitment transaction, pre-signed by the counterparty. The asymmetry between the two versions is deliberate and enables the revocation mechanism that will be covered in the next section. Both parties exchange signatures over their respective commitment transactions using the standard sighash `SIGHASH_ALL`. Each party verifies the counterparty's signature against the witness script. Only after both signatures are validated does the funding transaction become safe to broadcast.

1. Construct the commitment transaction spending the funding output (referenced by the funding txid and output index).
2. Exchange `commitment_signed` messages containing the counterparty's signature over your commitment transaction.
3. Verify the received signature `σ(commitment_tx, P_remote)` against the 2-of-2 witness script.
4. Once verified, broadcast the funding transaction. The channel is now open.

Step 5: Verify the On-Chain Funding Confirmation and Finalize Channel Readiness

After broadcast, both parties monitor the blockchain for the funding transaction's confirmation. The BOLT specification requires a minimum confirmation depth (typically around 3 to 6 blocks, configurable per implementation) before the channel is considered usable. This confirmation depth is a security parameter: it determines the cost an attacker must bear to double-spend the funding transaction via a chain reorganization. Once the funding transaction reaches the agreed-upon confirmation depth, both parties exchange `channel_ready` (formerly `funding_locked`) messages. At this point, the 2-of-2 output is confirmed on-chain, a valid commitment transaction exists for both sides, and the channel is ready to carry off-chain state updates. Every subsequent balance update between Kira and the GPU provider will be a new commitment transaction that redistributes the same funding output, with no additional on-chain cost until one party decides to close the channel.

1. Monitor the blockchain for the funding transaction's inclusion in a block.
2. Wait for the agreed-upon `minimum_depth` confirmations.
3. Exchange `channel_ready` messages with the counterparty, each referencing the confirmed funding txid.
4. Transition the channel state machine to the operational phase. The funding output now anchors all off-chain commerce between the two parties.

You have now constructed the cryptographic structure that makes trustless off-chain commerce possible between two parties who share nothing but a network connection and a willingness to lock capital. The 2-of-2 funding output is not a feature of the Lightning protocol; it is its foundation. Every hash-locked payment, every multi-hop route, every millisatoshi-granularity micropayment that Kira will eventually send to the GPU provider traces its authority back to this single confirmed output and the pair of keys that govern it. With the funding output confirmed and the initial commitment transaction in hand, the next construction to examine is how subsequent commitment transactions redistribute the locked balance off-chain, and how revocation keys ensure that only the latest state can be enforced. That mechanism is where the

channel acquires its dynamic character, transforming a static lockbox into a living ledger of bilateral state transitions.

Why Signature Aggregation and Key Tweaking Matter for Channel Funding Efficiency

Kira's node signs the funding transaction input, committing her UTXO to a channel she cannot unilaterally reclaim. That single act of cryptographic authorization, trivial in computational cost, is the precise moment a passive public key becomes an active economic commitment. Everything that follows in the channel's lifecycle depends on the properties this signature carries.

A digital signature over secp256k1, whether produced by ECDSA or Schnorr, binds three objects into one verifiable statement: a specific message (the serialized transaction), a specific public key (the signer's identity), and a proof that the holder of the corresponding private key consented to exactly that message. The verification equation confirms the binding without exposing the secret. This non-repudiable consent is what distinguishes a signed transaction from a mere data structure. Hash commitments, as established earlier, prove knowledge of a preimage. Signatures prove something categorically different: authorization by an identified party. Together, these two primitives answer the two irreducible questions any payment must resolve. Was the correct secret revealed? Did the rightful owner consent?

Yet a single signature authorizes a single will. When two strangers must share control over locked funds, the construction must generalize from one signer to a joint policy. This is where multisig enters, and where the comparison between legacy script-level multisig and aggregated-key schemes becomes architecturally consequential. Bitcoin's original OP_CHECKMULTISIG encodes an n-of-m policy directly in the output script: the spending transaction must present m public keys and n valid signatures, each verified independently against the message. For the 2-of-2 funding case, this means two distinct signatures appear on-chain, two public keys are embedded in the redeem script, and the script interpreter runs two independent verification passes. The policy is transparent, the cost is proportional to the number of signers, and every observer can identify the construction as multisig by inspecting the scriptPubKey. This transparency carries a direct on-chain footprint penalty. Each additional public key and signature consumes witness bytes, increasing the transaction's weight and therefore its fee cost. More subtly,

the visible multisig pattern leaks information about the spending policy to any chain observer, reducing the fungibility of the UTXO.

Schnorr signatures over secp256k1, activated with the Taproot upgrade, enable a fundamentally different approach through key aggregation. MuSig2 allows two parties to combine their individual public keys P_A and P_B into a single aggregate key P_agg through a deterministic tweaking procedure that prevents rogue-key attacks. The resulting on-chain output looks identical to a single-key pay-to-taproot output. Spending requires one aggregate signature σ_agg that validates against P_agg, produced through a two-round interactive protocol between the co-signers. The on-chain footprint collapses from two keys and two signatures to one key and one signature. The weight savings are not marginal. A legacy 2-of-2 P2WSH input consumes roughly 200-230 witness bytes for two compressed public keys plus two DER-encoded ECDSA signatures. A MuSig2 spend from a P2TR keypath requires around 64 bytes for the single Schnorr signature, with the public key implicit in the output. The difference compounds across every channel open and cooperative close in a network where millions of channels may coexist. Equally important, the aggregated output is indistinguishable from any ordinary single-signer Taproot spend, eliminating the privacy cost that legacy multisig imposes.

The comparison crystallizes along three dimensions. On-chain efficiency favors key aggregation decisively, reducing both fee cost and blockchain state consumption. Privacy favors aggregation equally, since no observer can distinguish a channel funding output from an ordinary payment. The tradeoff lives in protocol complexity: MuSig2 demands an interactive signing session with two nonce-exchange rounds and careful state management to prevent nonce reuse, whereas legacy multisig requires only independent signature generation. For automated channel management between machines, this interactive complexity is absorbed naturally into the existing communication session between peers. The overhead matters far less when both signers are software processes already exchanging protocol messages.

Regardless of which multisig construction is chosen, the result is the same cryptographic object: a 2-of-2 funding output that neither party can spend alone. This output is not merely a safety deposit box. It is a game-theoretic forcing function. By locking capital into a UTXO that demands both σ_A and σ_B for any valid spend, the construction converts an adversarial relationship between strangers into a structure where cooperation is the only path to fund recovery. Kira's GPU provider now

holds cryptographic assurance that the sats committed to the channel cannot vanish through unilateral action. The provider's reciprocal key contribution ensures Kira faces the same constraint. Neither party trusts the other. Both parties trust the construction. With mutual consent now required for every state transition touching these funds, the question that naturally follows is how the parties can update their respective balances without broadcasting each change to the base chain.

Trust-Free Correctness: Why Protocol Security Depends on Rationality, Not Honesty

Every payment Kira will ever make to an unknown GPU provider begins with the same structural absence: no shared institution, no reputation ledger, no prior interaction, and no human on either side to phone when something goes wrong. The conventional internet offers her sophisticated mechanisms for discovering that provider, negotiating an API schema, even authenticating its TLS certificate. But it offers exactly zero native mechanisms for transferring value to it without routing through a custodial intermediary whose continued good behavior both parties must simply assume. That gap is so deeply normalized that most protocol designers never register it as a design flaw. It is.

The instinct, faced with this absence, is to search for a trustworthy counterparty or a reliable intermediary. But the stronger move is to abandon the search entirely and ask a different question: can the payment protocol itself be constructed so that the counterparty's honesty is irrelevant to the correctness of the outcome? The answer depends on whether the protocol's incentive structure makes defection strictly dominated by cooperation under every possible counterparty strategy. Not because participants are virtuous, but because cheating costs more than it yields. This is the precise sense in which cryptographic payment verification achieves trust-free correctness. It converts a social problem into a game-theoretic constraint, and the mathematics of that conversion is where the chapter's real machinery begins.

The Honest-Party Fallacy: Modeling Adversarial Behavior as the Default Protocol Assumption

Every protocol designer faces a fork in the road that determines whether the system will survive contact with reality. One path assumes participants will behave honestly, that counterparties will fulfill their obligations because they are expected to. The other path assumes nothing about character and instead asks a simpler question: what does each par-

ticipant stand to gain or lose? The first path produces systems that work in demos. The second produces systems that work in adversarial environments where the counterparty is an anonymous process running on hardware you have never seen, in a jurisdiction that may not exist.

The mental model is this: replace the honesty assumption with the rationality assumption, and then verify that the protocol's desired outcome is the unique Nash equilibrium of the resulting game. A participant modeled as *honest* will follow the protocol because the rules say so. A participant modeled as *rational* will follow the protocol only if deviating yields a strictly worse payoff. The distinction matters enormously. Honesty is a behavioral prediction, and behavioral predictions fail precisely when the stakes are high enough to justify defection. Rationality is a structural constraint on payoff matrices, and it holds regardless of whether the participant is a human being with a reputation to protect or a stateless inference agent with no legal identity and no concept of reciprocity. The weaker assumption is the stronger foundation, because it tolerates the widest class of adversaries.

Consider what happens when Kira, still tethered to a single cloud billing account, encounters an unknown GPU provider for the first time. Under an honesty model, Kira must trust that the provider will deliver compute before payment clears, or the provider must trust that Kira will pay after receiving results. Someone goes first, and whoever goes first is exposed. But a hash-locked commitment restructures the game entirely. Kira locks payment to a hash H(x), where x is a preimage the provider generates and withholds until the computation is complete. The provider's strategy space reduces to two options: reveal x and collect payment, or withhold x and collect nothing. No rational provider withholds. Symmetrically, Kira's payment is released only upon receipt of a valid preimage, so no rational payer loses funds without receiving the corresponding proof of delivery. The turnstile clicks only when both sides push through at once, and the cryptographic lock is the mechanism that makes sequential passage unprofitable for either party.

This is not merely a clever trick. It is a provable property of the construction. For any participant i, let $U_i(\text{cooperate})$ and $U_i(\text{defect})$ denote the payoffs for following and deviating from the protocol, respectively. Trust-free correctness holds when $U_i(\text{cooperate}) > U_i(\text{defect})$ for all i, independent of the strategies chosen by other participants. The hash lock achieves this by making the payoff comparison trivial. The provider who reveals a valid preimage receives payment p. The provider who withholds it receives zero. The payer who submits a locked commit-

ment loses nothing unless a valid preimage appears, at which point the payment was warranted. No adjudicator evaluates claims. No reputation system filters bad actors. The protocol's security analysis reduces to comparing two numbers, and the comparison holds whether the counterparty is cooperative, indifferent, or actively hostile.

The model applies best when three conditions hold: participants can evaluate their own payoffs, the protocol's cryptographic locks are computationally binding, and the cost of attempting to break the hash function exceeds the value locked behind it. It can mislead when externalities exist outside the protocol's payoff matrix, for instance if a participant benefits from disrupting the system itself rather than from any single transaction outcome. Griefing attacks, where an adversary sacrifices their own payoff to impose costs on others, fall outside the strict rationality model and require separate treatment through timeout mechanisms and capital requirements. Recognizing this boundary is essential: rationality-based security covers the vast majority of commercial interactions, but the designer must identify and mitigate the residual attack surface where irrational or externally motivated behavior is possible.

What Kira gains from this shift is architectural, not incremental. Under the old model, switching providers meant trusting a new counterparty or relying on a human to vet the invoice. Under the rationality model, every provider Kira encounters is presumed adversarial from the first interaction, and the protocol still guarantees correct settlement. The honest-party assumption is not merely optimistic. It is structurally incompatible with autonomous machine commerce, where there is no courtroom, no contract law, and no phone number to call when something goes wrong. The hash-locked commitment is the first construction where the guarantee holds without any of those backstops, and it is the load-bearing primitive beneath every payment scheme that follows.

Dominant-Strategy Incentive Compatibility: Making Correct Execution the Rational Choice Under All Counterparty Strategies

In the summer of 2023, a researcher at a mid-sized AI lab named Kira watched her infrastructure team spend three weeks negotiating a billing agreement with a new GPU provider. Legal review, credit checks, API key provisioning, dispute resolution clauses. The contract ran to fourteen pages. The actual compute job, when it finally executed, lasted nine seconds. That ratio of institutional overhead to useful work is not an an-

omaly. It is the structural signature of a trust model that requires human judgment at every boundary between economic strangers.

The framework that eliminates this overhead rests on a single, precise substitution: replace the assumption that counterparties will behave honestly with the assumption that counterparties will behave rationally. Trust-free does not mean assumption-free. Every security model stands on some foundation. Traditional commerce assumes honesty backed by legal enforcement. Reputation systems assume that future business value outweighs short-term cheating gains. The framework specified here assumes only that each participant prefers outcomes that maximize its own payoff given the strategy choices available to it. This is the rationality axiom, and it is strictly weaker than honesty. An honest actor follows the rules because they are the rules. A rational actor follows the rules only when following them yields a better payoff than every available deviation. Protocol security built on rationality is therefore more robust: it holds even against participants who would cheat if cheating were profitable.

The cryptographic primitives already established, hash commitments and preimage reveals, become load-bearing only when embedded in a payoff structure that closes every profitable deviation path. Consider the interaction between Kira's agent and an unknown GPU provider through a hash-locked payment proof. Kira's agent locks funds behind a hash H(x), where x is a preimage the provider will learn only upon delivering a valid computation result. The provider commits compute resources to produce the result, and upon delivery, reveals x to claim the locked payment. Two candidate deviations present themselves. First, the provider could withhold the result after observing x. But the protocol structures the reveal so that x becomes available to the provider only through its own act of delivering the result to Kira's agent; the preimage is bound to the delivery channel, not transmitted independently. Second, Kira's agent could attempt to consume the result without releasing payment. But the hash lock is enforced by the settlement layer: once x is revealed, the provider can present the preimage to claim funds unilaterally, regardless of Kira's cooperation. Each party's attempt to defect produces a strictly worse payoff than compliance. Defection is not merely unlikely. It is a dominated strategy.

This property, where correct protocol execution yields a payoff at least as high as any alternative strategy regardless of what the counterparty does, is dominant-strategy incentive compatibility. It is the strongest equilibrium condition available. Nash equilibrium requires

only that no player benefits from unilateral deviation given the other's strategy. Dominant-strategy compatibility goes further: compliance is optimal even if the counterparty behaves adversarially, irrationally, or unpredictably. For protocol designers, the consequence is a precise engineering constraint. Every state transition in the protocol must be audited against a complete deviation analysis. For each participant at each step, the designer must verify that the payoff from following the protocol exceeds the payoff from every alternative action, including actions that combine multiple deviations across time. If any profitable cheating path exists, the mechanism is incomplete.

The design tool that enforces this constraint is the pairing of cryptographic evidence with economic penalty. A hash preimage is not just an authentication token. It is a non-repudiable proof of a state transition. When a provider reveals x, it simultaneously proves delivery occurred and triggers an irreversible payment claim. When Kira's agent verifies H(x), it simultaneously confirms work completion and authorizes settlement. No third party adjudicates. No reputation score is consulted. The cryptographic objects themselves carry the entire burden of proof, and the economic structure ensures that producing those proofs is each party's self-interested choice. The protocol does not prevent dishonesty through surveillance or punishment after the fact. It renders dishonesty structurally unprofitable before the fact.

This is the foundation that every subsequent construction in this book depends on. Payment channels, multi-hop routing, bearer credential delegation: each is a composition of mechanisms that must independently satisfy the same dominant-strategy condition. The question is never whether participants will cooperate. The question is whether the mechanism makes cooperation each participant's uniquely rational act. When hash commitments and preimage reveals are structured so that the answer is unconditionally yes, the fourteen-page contract collapses to a single cryptographic exchange, and Kira's nine-second compute job needs no lawyer, no reputation ledger, and no trust.

Penalty Mechanisms and Time-Lock Escalation: Cryptographic Enforcement That Converts Defection into Loss

Kira's runtime evaluates a hash-locked payment proof from an unknown GPU provider, compares the committed output against the delivered compute result, and releases the preimage. The entire interaction completes without either party assessing the other's reputation, querying a dispute-resolution service, or invoking any authority beyond the

protocol itself. What makes this exchange secure is not a belief that the counterparty will behave well. It is the structural reality that misbehavior costs more than compliance.

This distinction between honesty-based and rationality-based security models is the fulcrum on which every subsequent construction in this book rests. A trust-based system assumes participants will cooperate because they are vetted, legally bound, or reputationally invested. A rationality-based system assumes participants will cooperate because the protocol makes cooperation the dominant strategy for any self-interested actor, regardless of intent. The difference is not philosophical decoration. It is the boundary between systems that require human oversight and systems that machines can operate autonomously. Consider the payoff structure of a hash-locked commitment. The payer constructs a payment conditioned on the reveal of a preimage r such that $\mathrm{H}(r)$ equals a publicly committed digest. The payee can claim funds only by publishing r, which simultaneously proves delivery of the contracted service. If the payee attempts to claim without delivering, the payer never reveals r and the locked funds revert after a timeout Δt. If the payer attempts to consume the service without paying, the payee withholds the service output until r is released. Neither party can profitably deviate. The mechanism does not prevent a participant from wanting to cheat. It ensures that cheating is strictly dominated by honest play in every reachable game state.

The critical observation is that penalty mechanisms and time-lock escalation convert attempted defection into guaranteed loss. When a channel party broadcasts a revoked commitment transaction, the counterparty holds a revocation key that sweeps the entire channel balance. The cheating party does not merely fail to gain an advantage. They forfeit everything. This asymmetry is not incidental. It is engineered so that the expected value of defection is negative under any probability distribution over counterparty alertness. Time-locks reinforce this structure by creating a window during which the honest party can observe the breach and respond. The timeout parameter Δt defines the enforcement horizon: a longer lock gives the honest party more time to detect and penalize, while a shorter lock reduces capital lockup cost. Protocol designers tune this tradeoff surface along two explicit dimensions, enforcement reliability and capital efficiency, and the resulting equilibrium holds as long as the honest party monitors the chain within the timeout window.

Machines cannot litigate. They cannot phone a support line, file a chargeback, or assess whether a counterparty "seems trustworthy." An autonomous agent evaluating whether to commit resources to a transaction has exactly one tool: it can verify a cryptographic proof against a known protocol specification. If the security of that protocol depends on the counterparty being honest, it depends on something the machine cannot measure. If it depends on the counterparty being rational, it depends on something the mechanism can enforce. This is why every payment primitive in this architecture derives its guarantees from economic self-interest constrained by cryptographic evidence. The hash function is not a lock that keeps dishonest actors out. It is a structure that makes honesty cheaper than dishonesty for every actor inside the game.

What emerges from this foundation is a composable security guarantee. Because each atomic interaction between Kira and the GPU provider resolves to a strict Nash equilibrium where both parties prefer compliance, the same guarantee can be extended across time. A single hash-locked exchange becomes a sequence of conditional payments within a channel. A channel becomes a multi-hop route where each intermediary faces the identical payoff structure. The rationality guarantee proven at the single-exchange level does not dilute as the protocol scales. It compounds, because each additional step inherits the same dominant-strategy property from its cryptographic constraints. The next question is how these atomic rational commitments compose into persistent payment channels with time-locked state transitions that span hours, days, or thousands of sequential micropayments, and how the penalty mechanism adapts when the game extends across multiple rounds with evolving balances.

The Trust Substrate

Three primitives, one architectural fact. A hash preimage commitment binds a payment to a provable revelation: Kira publishes H(s), and the only way to claim the locked funds is to produce s, transforming a 32-byte string into an irrevocable receipt that no institution needs to co-sign. A digital signature over a transaction output proves exactly who authorized the movement of value, and a 2-of-2 multisig construction creates joint custody that neither party can unilaterally dissolve, turning two mutually suspicious machines into co-signers of an escrow that enforces itself. The rationality constraint closes the loop: neither Kira nor the GPU provider needs to believe the other is honest, because the protocol makes defection strictly more expensive than cooperation. These

are not three separate tools stored in adjacent drawers. They are the load-bearing members of a single structure, a minimum viable trust substrate that reduces commerce between strangers to the production and verification of small cryptographic proofs. A machine can execute this exchange as competently as any human, faster, and at negligible marginal cost. Nothing about the process requires judgment, reputation, or relationship. It requires only math.

Sketch the protocol on paper and the chapter's full weight becomes tangible. Kira generates a secret s, computes H(s), and publishes the commitment alongside a signed transaction that locks funds redeemable only by whoever reveals the preimage.

Chapter Two

State Channel Mechanics and Off-Chain State Transitions

Most engineers assume that a payment requires a transaction. The assumption is so deeply embedded that it barely registers as an assumption at all. You want to transfer value, you broadcast to the network, you pay the fee, you wait for confirmation. But run the arithmetic on machine-to-machine commerce and the assumption collapses under its own weight. A single forward pass through a 70-billion-parameter model costs a GPU provider roughly four-hundredths of a cent in electricity and depreciation. Broadcasting a Bitcoin transaction to settle that pass costs somewhere between thirty cents and three dollars depending on mempool congestion, a markup of roughly one thousand to ten thousand times the value being transferred. If every micropayment must touch the chain, the economics of autonomous compute markets are dead on arrival.

The problem is not that on-chain settlement is flawed. It is that on-chain settlement is being asked to do work it was never designed to do at this granularity. Hash commitments, as the previous chapter established, give two parties the ability to bind themselves to values and verify correctness without trust. But a commitment primitive alone cannot amortize settlement cost across thousands of payments. What is needed

is a structure that pays on-chain exactly once, then conducts an unlimited sequence of cryptographically binding balance updates entirely off-chain, touching the blockchain again only if someone cheats.

That structure begins with a single, carefully constructed on-chain transaction. Kira and its GPU provider must collaboratively lock capital into a shared output governed by both parties' keys, establishing the total capacity of every off-chain payment that will follow.

Channel Opening: On-Chain Funding Transactions, Collateral Locking, and Capacity Establishment

Most engineers assume the hard part of a payment channel is the off-chain update mechanism. But the protocol's real trust boundary is drawn earlier, in a single on-chain transaction whose output script neither party can spend alone. If that funding output is malformed, if the commitment transaction is not pre-negotiated before broadcast, or if the capacity is miscalibrated, then every subsequent state transition inherits a structural defect that no amount of clever routing can repair.

Kira has selected a GPU provider offering competitive inference pricing and is ready to commit capital. The operation reduces to a concrete problem: construct a 2-of-2 multisignature output paying real satoshis to a script address that only a joint signature from Kira's key and the provider's key can unlock. This means that the moment the funding transaction confirms, the locked collateral becomes inaccessible to either party acting independently. That property is the entire point. It converts sovereign control over funds into a shared economic surface where bilateral signed updates carry enforceable weight. But it also introduces an immediate hazard. If either counterparty disappears after funding confirms and before a valid commitment transaction exists, those satoshis are irrecoverable. The protocol must therefore ensure that the refund path is cryptographically secured before a single satoshi leaves Kira's wallet.

What follows traces that construction from initial key exchange through broadcast confirmation, then examines why locking capital in this specific structure creates the incentive substrate for off-chain settlement, and how Kira should reason about sizing that lock against the real costs of on-chain fee amortization and idle liquidity.

Constructing the 2-of-2 Multisig Funding Output: From Key Exchange Through Broadcast Confirmation

Most engineers assume that opening a payment channel begins with broadcasting a transaction. The critical work, however, happens before any transaction touches the mempool. Two parties must exchange public keys, derive a joint 2-of-2 multisignature output script, and pre-sign a refund path that guarantees each side can recover its funds if the counterparty vanishes. Only after that refund commitment exists does the funding transaction become safe to broadcast. The on-chain event itself is deceptively simple: a single UTXO locked to a script requiring both signatures to spend. But the ceremony surrounding its construction is what converts raw capital into a trust-free state machine.

Consider Kira's concrete situation. It has identified a GPU provider offering inference capacity at competitive per-token rates, but the provider's billing API accepts only prepaid invoices settled in batches of roughly ten thousand calls. Kira needs to pay incrementally, per inference call, at a granularity the invoice model cannot support. To establish that capability, Kira and the provider each generate a fresh keypair for the channel. They exchange public keys and construct a P2WSH output whose witness script encodes `OP_2 <Kira_pubkey> <Provider_pubkey> OP_2 OP_CHECKMULTISIG`. This output becomes the funding address. Kira intends to commit 500,000 satoshis of channel capacity, which defines the ceiling for all subsequent balance updates. The provider contributes nothing on the funding side in this configuration, a common pattern called a single-funded channel. The total locked value, the channel capacity, is an explicit capital-allocation decision: enough to cover an estimated several days of inference purchasing, amortized against the on-chain fee Kira pays to open and eventually close the channel.

But Kira does not broadcast yet. If it were to push the funding transaction to the network without a pre-signed escape path, its 500,000 satoshis would sit in an address that requires both parties' signatures to move. Should the provider go offline permanently, those funds would be irrecoverable. The defense is a commitment transaction, signed by both parties before the funding transaction is broadcast, that spends the 2-of-2 output back to Kira's own address after a timelock. This initial commitment, call it C_0, represents the opening balance state: Kira holds the full capacity, the provider holds zero. Both sides sign C_0, each retaining a copy. Only then does Kira serialize and broadcast the funding

transaction. The protocol's safety property is clean: at every moment after broadcast, at least one valid commitment transaction exists that can unilaterally return funds to their rightful owners without cooperation.

Once the funding transaction confirms on-chain, which in practice means waiting for around three to six block confirmations to achieve adequate reorganization resistance, the channel is live. The 2-of-2 multisig UTXO now sits in the UTXO set as a trust anchor. Its existence is the only on-chain footprint the channel requires for its entire operational lifetime. Every subsequent balance shift between Kira and the provider will be a pair of signed commitment transactions exchanged directly over their communication link, updating the allocation of those 500,000 satoshis without consuming any additional block space. The on-chain fee Kira paid to confirm the funding transaction is a one-time cost that gets amortized across potentially thousands of micropayments.

The capital lockup introduces a genuine tradeoff. Those 500,000 satoshis are unavailable to Kira for any other purpose until the channel closes. If Kira underestimates its inference demand, it may exhaust the channel capacity and need to open a new channel or splice in additional funds, each operation carrying its own on-chain fee and confirmation delay. If Kira overestimates, capital sits idle, earning no return. The channel capacity decision is therefore a resource-allocation problem shaped by expected payment volume, on-chain fee rates at the time of opening, and the opportunity cost of locked liquidity. For an autonomous agent operating without a human treasury manager, this decision must eventually be algorithmic, informed by historical demand patterns and current fee-market conditions.

With the funding output confirmed and C_0 held by both parties, Kira possesses something it did not have before: a cryptographically anchored bilateral ledger capable of recording thousands of balance updates at zero marginal on-chain cost. The next question is how those updates actually work, how Kira and the provider will exchange new commitment transactions that shift satoshis from one side to the other, and what prevents either party from publishing a stale state that favors them.

Collateral Lockup as Credible Commitment: Why Capital Immobilization Creates the Incentive Substrate for Off-Chain Settlement

Roughly seven out of ten Lightning Network channels observed in public graph snapshots carry a capacity below 500,000 satoshis. That figure is not a limitation. It reflects a precise economic judgment: the capital

locked into a channel needs only to cover the burst window of payments expected before the next settlement opportunity, not the lifetime value of the relationship. For machine-to-machine micropayments, where individual transfers may be a few hundred satoshis per compute slot, even modest channel capacities can sustain thousands of balance updates. The mechanism that makes this possible begins with a single on-chain event: the funding transaction.

A funding transaction constructs one UTXO governed by a 2-of-2 multisignature output, jointly controlled by both channel parties. This is the channel's entire on-chain footprint until closure. Both participants contribute inputs to this transaction, and the resulting output locks their combined collateral into a script that requires both signatures to spend. The total value of this output defines channel capacity, denoted C, which sets a hard ceiling on the sum of all off-chain balances at any point in the channel's life. If Kira opens a channel with a GPU provider at C = 50,000 satoshis, then for every subsequent state update the invariant holds: balance_Kira + balance_provider = C. No off-chain operation can create or destroy value within the channel. It can only redistribute what was locked at funding time. This single constraint converts an unbounded bilateral relationship into a bounded, verifiable allocation problem, and it is the reason that a one-time on-chain cost can amortize across thousands of subsequent micropayments without touching the chain again.

Before the funding transaction is broadcast, however, both parties face a cooperative-failure trap. Suppose Kira signs and broadcasts a funding transaction that locks her 25,000 satoshis into the 2-of-2 multisig. If the GPU provider then disappears or refuses to cooperate, those funds are permanently irrecoverable, since spending the multisig output requires both signatures. The solution is temporal: before either party broadcasts the funding transaction, they exchange pre-signed commitment transactions that spend the funding output back to each party's respective balance. Concretely, Kira and the provider each construct and countersign a transaction spending the not-yet-confirmed funding UTXO, allocating 25,000 satoshis to Kira and 25,000 to the provider. Only after both hold a valid, fully signed refund commitment does either party broadcast the funding transaction to the mempool. The refund commitment functions as a unilateral exit guarantee. It ensures that each party can recover their deposit without the counterparty's future cooperation, collapsing the trust requirement from ongo-

ing bilateral agreement down to a single atomic verification at channel creation.

This ordering, where commitment precedes funding, is not a convenience but a strict safety invariant. If a party broadcasts funding before holding a signed refund, they have transferred custody of their capital to a script that their counterparty can hold hostage through inaction alone. The pre-signed refund eliminates this leverage asymmetry. It converts the act of locking capital from a trust-dependent deposit into a credible commitment backed by a cryptographically enforced exit path. The locked capital is immobilized but never trapped. Each party's worst-case outcome is a unilateral close that returns their original deposit, minus on-chain fees, after a timelock expiration.

Capital immobilization under these conditions creates the incentive substrate for honest off-chain settlement. Because both parties have value at stake inside the 2-of-2 output, any off-chain balance update carries real economic weight. A signed state transition that moves 200 satoshis from Kira to the GPU provider is credible precisely because the underlying collateral exists, is locked, and can be claimed on-chain if the counterparty deviates from the agreed state. The channel capacity does not need to be large. It needs only to exceed the maximum net transfer expected during the interval between balance resets or cooperative closes. For Kira purchasing compute slots at 200 satoshis each, a 50,000-sat channel covers 250 sequential payments before exhaustion, which might represent hours or days of autonomous operation depending on workload intensity.

With the funding transaction confirmed and both parties holding valid refund commitments, the channel exists as a shared, mutable ledger anchored to an immutable on-chain deposit. The next operation is the first real balance update: a signed state transition that reallocates the locked funds without any chain interaction, and that introduces the question of what happens when a prior state becomes obsolete.

Capacity Sizing Under Uncertainty: Balancing On-Chain Fee Amortization Against Locked-Liquidity Opportunity Cost

The decisive constraint surfaces the moment Kira evaluates how much capital to lock into the funding output. Too little collateral and the channel exhausts within a handful of inference calls, forcing a costly on-chain close and reopen cycle. Too much and the locked satoshis sit idle across a 2-of-2 multisig output while higher-yield deployments go unfunded. The sizing decision is not a guess. It is a structured optimization

over three competing variables, each with a measurable cost function, and Kira must resolve it programmatically before broadcasting the funding transaction.

The first variable is on-chain fee amortization. A single funding transaction that commits collateral to a OP_CHECKMULTISIG output carrying both Kira's and GPU-Provider-Alpha's public keys incurs one base-layer fee, typically in the range of a few thousand satoshis depending on mempool conditions. Every subsequent off-chain balance update that flows through this channel amortizes that initial cost further toward zero. If Kira expects to execute around ten thousand inference micropayments over the channel's lifetime, each payment's marginal on-chain burden drops below a fraction of a satoshi. The amortization curve is hyperbolic: the first hundred payments drive the per-payment cost down sharply, while each subsequent hundred yields diminishing marginal improvement. This means even a modestly sized channel delivers strong amortization, provided it is not prematurely closed. Kira's decision logic therefore estimates the expected payment count before a cooperative close, multiplies by the anticipated per-payment amount, and derives a minimum capacity floor beneath which the amortization benefit fails to justify the funding transaction's fee.

The second variable is locked-liquidity opportunity cost. Capital committed to the multisig output cannot simultaneously fund a second channel with GPU-Provider-Beta or be deployed in a routing capacity that earns forwarding fees. Kira models this cost as the expected return the same satoshis would generate in their next-best alternative deployment over the channel's anticipated lifetime. If Kira operates in an environment where routing fee income from well-positioned channels yields roughly 50 basis points annually, locking an additional 100,000 satoshis into a single provider channel for three months carries a quantifiable opportunity cost. The capacity allocation thus has a ceiling: the point at which the marginal satoshi locked produces less amortization benefit than the return it would earn elsewhere.

The third variable introduces genuine uncertainty. Kira cannot know with precision how many inference calls GPU-Provider-Alpha will serve, whether fee rates will spike during the channel's life requiring a premature force-close, or whether Provider-Alpha's uptime will degrade. Capacity planning under these conditions demands a probabilistic frame. Kira maintains a distribution over expected payment volumes derived from historical inference demand, models on-chain fee volatility as a stochastic process affecting the cost of a potential mid-life channel

close, and combines these into a risk-adjusted capacity estimate. A simple heuristic collapses this: fund at roughly 1.5 times the median expected lifetime volume, accepting slightly higher opportunity cost in exchange for a buffer against volume surges that would otherwise force an expensive on-chain resize.

The interaction between these three factors creates a compact decision surface. At low capacity, amortization is weak and premature closure risk is high, driving total cost upward. At high capacity, amortization is saturated and opportunity cost dominates, again driving total cost upward. The minimum lies in a region where expected payment volume is comfortably within capacity, amortization has reached the flat portion of its curve, and the locked amount remains modest relative to Kira's total available liquidity. Kira's sizing algorithm locates this region by sweeping candidate capacities, computing expected total cost as the sum of amortized on-chain fees, probability-weighted forced-close costs, and opportunity cost over the projected channel lifetime, then selecting the capacity that minimizes the sum.

Before Kira broadcasts this carefully sized funding transaction, the safety ordering discussed earlier applies without exception. The commitment transaction carrying both signatures must exist in Kira's local state before the funding output is confirmed on-chain. With that safeguard in place and the capacity chosen through the framework above, Kira publishes the funding transaction and watches it confirm. A live channel now connects Kira to GPU-Provider-Alpha, its capacity set neither by instinct nor by default, but by a disciplined balance of fee physics, capital cost, and probabilistic demand forecasting. What remains is to use it: to exchange the signed state updates that will move value back and forth across this shared ledger without ever touching the chain again.

Signed Balance Updates Exchanged Peer-to-Peer: Near-Instant Micropayment Settlement at Negligible Marginal Cost

Most engineers assume a payment requires a network round-trip to some settlement authority. Inside an open state channel, that assumption dissolves. Kira's first purchase of an inference slot, 200 sats for a single forward pass on a 7B-parameter model, is nothing more than a locally constructed state tuple: the channel's funding outpoint identifier, a sequence number incremented from zero to one, and two balance fields whose sum remains invariant at the original funding amount. Kira signs this tuple with its private key, transmits the roughly 150-byte mes-

sage over a direct TCP connection to the GPU provider, and waits. The provider verifies the signature against Kira's channel pubkey, confirms the balances sum correctly, counter-signs, and returns the result. That round-trip is the payment. No broadcast, no confirmation block, no fee deducted.

What makes this exchange enforceable rather than merely informal is the cryptographic structure of the object both parties now hold. Either side can take the latest mutually signed state to the blockchain and close the channel at the balances it specifies. Cooperation is cheaper, so neither party does, but the option to do so transforms a peer-to-peer message into a binding settlement instrument. And because each subsequent update simply increments the sequence number and shifts satoshis from Kira's balance to the provider's, the marginal cost of the next payment collapses to the computational overhead of a single ECDSA verification and a few bytes of local storage. The channel can ratchet forward a thousand times per second, purchasing inference at per-token granularity, without generating a single on-chain transaction.

Asymmetric Commitment Transactions: Each Party Holds a Distinct Spend Path Enforcing the Latest Agreed Balance

Most engineers assume that a payment requires a broadcast. A transaction is composed, signed, submitted to a network, confirmed into a block, and only then considered settled. That assumption holds on-chain. Off-chain, it dissolves.

Inside a funded state channel, a payment is nothing more than a co-signed message. The message contains three elements: a monotonically increasing sequence number (the nonce), a new balance partition reflecting the transfer, and a valid signature from each party. No gas is consumed. No block is mined. No miner or validator touches the update. The two peers exchange this message directly, and the act of co-signing it constitutes settlement. Each new co-signed state with a higher nonce supersedes every prior state, not by erasing history but by rendering it economically irrelevant. If either party ever needs to force settlement on-chain, the latest co-signed state is the only artifact that matters: the chain will honor the highest-nonce commitment both parties authenticated, collapsing an arbitrary number of off-chain updates into a single enforceable output split.

The cost structure this creates is qualitatively different from any on-chain payment model. The first signed update in a channel costs one signature generation, one signature verification, and one message transmis-

sion. The hundredth update costs exactly the same. So does the thousandth. The fixed costs of the channel, the on-chain funding transaction and eventual on-chain settlement, are amortized across every update the channel carries. As the number of updates grows, the marginal cost per payment converges toward the computational expense of a single ECDSA verification and a few hundred bytes of bandwidth. For two machines exchanging value at the cadence of API calls, this is the difference between a toll road and an open highway.

Consider the concrete progression. Kira has opened a funded channel with a GPU inference provider, locking an initial deposit into a 2-of-2 multisig output. The opening state assigns the full deposited balance to Kira's side, zero to the provider. Kira now requests a single inference slot. The provider delivers results, and both parties co-sign a new state, nonce 1, shifting a small increment of satoshis from Kira's balance to the provider's. Kira requests another slot. Nonce 2. Another increment moves. Over the next few minutes, Kira consumes dozens of compute slots, and each consumption produces a fresh co-signed state with a strictly higher nonce. The provider holds the latest signed commitment as proof of its accumulated earnings. Kira holds the same commitment as proof of its remaining balance. Neither party has touched the blockchain since the channel opened, yet every individual payment is cryptographically enforceable. The provider could broadcast the latest state at any moment and receive exactly the balance reflected in the most recent co-signed update.

The enforceability guarantee is what separates this construction from a mere IOU ledger. Each co-signed state is a self-contained proof: it references the original funding output, specifies the exact balance partition, and carries both signatures required to spend the multisig. Either party, acting unilaterally, can submit this proof to the base chain and trigger a settlement that distributes funds according to the last agreed split. Both parties have rational incentive to retain and honor the latest state because attempting to broadcast an older, more favorable version exposes them to a penalty mechanism. That mechanism, the revocation structure that keeps peers honest even when the channel has carried thousands of updates, depends on a specific asymmetry in how each party's commitment transaction is constructed.

An Agent Purchasing Inference Slots at Per-Token Granularity via Sequential Balance Ratchets

Roughly seven out of every ten inference API calls made by brokering agents today settle through monthly invoices, meaning the actual transfer of value lags the delivery of compute by days or weeks. For Kira, whose channel with a GPU provider now holds 50,000 satoshis in a 2-of-2 multisig output, that lag collapses to zero. The funding transaction confirmed on-chain minutes ago. The channel is open, idle, and waiting for its first signed balance update. What happens next occurs entirely between two processes exchanging messages over a direct transport, with no miner, no mempool, and no gas fee standing between intent and settlement.

Kira's first inference request arrives from an upstream caller. The agent selects a model, serializes the prompt, and before transmitting the payload to the provider, constructs a balance update: a compact tuple containing the channel identifier `chan_id`, a sequence number `seq = 1`, a new balance allocation `{kira: 49988, provider: 12}`, and Kira's cryptographic signature $\sigma(\text{update}_1, \text{sk\_kira})$. The twelve satoshis represent the provider's posted price for a single inference slot at the requested token count. Kira sends this signed update to the provider over their peer-to-peer link. The provider validates the signature against the public key bound to the funding output, confirms that `seq = 1` exceeds the channel's initial state of `seq = 0`, verifies that the total allocation still sums to the locked 50,000 satoshis, and then countersigns: $\sigma(\text{update}_1, \text{sk\_provider})$. Both parties now hold a fully signed state that either could, in principle, broadcast to the chain to close the channel with that exact split. Neither does. The provider delivers the inference result. Elapsed wall-clock time for the payment: under four milliseconds.

The second request follows almost immediately. Kira constructs update_2 with `seq = 2` and balances `{kira: 49976, provider: 24}`. The structure is identical, the verification is identical, and the cost is identical: zero on-chain fees, zero confirmation delay. By the time Kira has processed fifty inference calls, `seq = 50` reflects a cumulative transfer of 600 satoshis and the blockchain has recorded nothing beyond the original funding transaction. Each intermediate state, from `seq = 1` through `seq = 49`, is economically dead. Only the highest-sequence-number update signed by both parties constitutes the current enforceable balance. This is the ratchet: sequence numbers move strictly up-

ward, and each increment overwrites every prior allocation without touching persistent storage outside the two counterparties' local state.

The marginal cost curve here deserves explicit attention. The first payment in the channel is expensive in amortized terms because it carries a share of the funding transaction's on-chain fee. The hundredth payment amortizes that same fixed cost across a hundred updates. By the thousandth, the per-payment overhead attributable to on-chain settlement is a fraction of a satoshi. The variable cost of each additional update is bounded by the computational expense of one signature generation and one signature verification, operations that complete in microseconds on commodity hardware. No other settlement architecture achieves this cost profile at this latency. Traditional payment rails impose per-transaction fees that exceed the value of a single inference slot by orders of magnitude. Even batched invoice systems carry accounting overhead that dwarfs twelve satoshis. The signed balance update eliminates that entire cost layer by deferring chain interaction to the channel's eventual close.

This cost collapse is what transforms per-token payment from a theoretical curiosity into an operational primitive. Kira is not batching requests into coarse billing periods or prepaying for inference bundles it may not consume. It pays for exactly what it uses, at exactly the moment it uses it, through a message that fits in a single TCP segment. But the mechanism introduces an asymmetry that the reader should already sense: since both parties hold every prior signed state, what prevents the provider from broadcasting update_1, where it received only twelve satoshis, after Kira has already paid forward to update_{50} and 600 satoshis? The answer lies outside the balance update itself, in a revocation construction that makes broadcasting stale state economically suicidal. That construction is the next dependency in the stack.

Marginal Cost Decomposition: Signature Generation, Message Transmission, and State Storage as the Only Per-Payment Expenses

The first expense a signed balance update incurs is computation that never touches a network. When Kira's agent completes an inference call and the GPU provider returns a valid result, both parties must produce a cryptographic signature over the new state. For ECDSA on the secp256k1 curve, this operation consumes on the order of tens of microseconds on commodity hardware. The signed message itself is compact: a channel identifier, a monotonically increasing sequence number n, and

a pair of balance allocations (b_Kira, b_provider) such that b_Kira + b_provider equals the channel's total funded capacity C. Both parties sign this tuple, yielding a co-signed state S_n = (channel_id, n, b_Kira, b_provider, σ_Kira, σ_provider). The sequence number is the critical ordering primitive. Any state S_n with n > m cryptographically supersedes S_m, because the on-chain settlement contract will accept only the highest-sequence co-signed state presented during a dispute window. This total ordering means that only the latest mutually signed state carries enforceable weight. All prior states are economically dead the moment a new one is co-signed.

The second cost component is message transmission. Each co-signed state is roughly 200 to 300 bytes, depending on signature encoding. Kira's agent sends its proposed state and signature to the provider over whatever transport the peers share: a TCP socket, a WebSocket connection, even a Tor circuit. The provider validates the proposed balance shift against the completed work, co-signs if correct, and returns its counter-signature. Round-trip latency on a direct peer connection sits in the low single-digit milliseconds for geographically proximate nodes, and the bandwidth consumed per update is negligible against any modern link. No block propagation delay, no mempool congestion, no miner fee auction. The payment settles the instant both signatures exist on both sides of the channel. Kira's first inference call shifts perhaps 50 satoshis from b_Kira to b_provider. The second call produces S_2 with n=2, shifting another 50. After a thousand calls, S_1000 reflects the cumulative balance reallocation, and the channel has consumed no additional on-chain resources beyond the original funding transaction.

The third and final per-payment expense is state storage. Each party must persist the latest co-signed state S_n, along with sufficient metadata to reconstruct and broadcast a valid closing transaction if the counterparty becomes unresponsive. In practice this means storing a single state object of a few hundred bytes, overwriting the previous one. There is no growing log of historical transactions that must be maintained for operational correctness. The storage footprint per channel is essentially constant regardless of how many balance updates have occurred. A machine running ten thousand concurrent channels stores on the order of a few megabytes of active channel state.

Consider the cost structure in aggregate. The on-chain funding transaction that locked collateral into the 2-of-2 multisig paid a one-time fee, perhaps a few thousand satoshis depending on fee-market conditions at broadcast time. Every subsequent balance update costs only

the CPU cycles for signature generation, the bandwidth for a sub-kilobyte message exchange, and the storage write for the new state. These three costs are measured in microseconds, bytes, and bytes respectively. The marginal cost per payment is dominated by the signature computation, and that cost is vanishingly small. This is the structural inversion that makes micropayment settlement viable: the expensive operation, on-chain consensus, happens exactly twice per channel lifetime (open and close), while the cheap operations, signing and transmitting, happen an unbounded number of times between those bookends.

The protocol is symmetric in an important sense. Both Kira and the GPU provider hold identical copies of S_n at all times. Neither party custodies the other's funds. Either party can unilaterally broadcast S_n to the settlement contract and trigger channel closure, receiving exactly the balance the latest co-signed state allocates to them. No third-party intermediary adjudicates, escrows, or routes. The trust model rests entirely on the cryptographic validity of the signatures and the economic finality guaranteed by the on-chain contract's willingness to enforce the highest-sequence state. This symmetry is what makes the construction non-custodial: Kira's agent holds a signed proof of its current balance that the blockchain will honor on demand.

Yet this very mechanism introduces an obvious tension. Both parties hold not just the latest state but the memory of every prior state they once co-signed. Nothing in the protocol as described so far prevents the GPU provider from broadcasting S_500, a state where it had accumulated fewer earnings, if S_1000 happens to reflect a partial refund. The monotonic sequence number tells the contract which state is newer, but only if the honest party is present to contest. How the protocol forecloses this attack is a question the current construction deliberately leaves open.

Dispute Resolution via Time-Locked Commitment Transactions and Revocation Key Penalties

Every commitment transaction Kira and the GPU provider have exchanged carries a valid signature, and that validity is precisely the problem. A correctly signed transaction from state twelve looks identical to the base layer as one from state four hundred. The chain has no memory of which balance split is current; it enforces only that the funding output's 2-of-2 multisig condition is satisfied. So when the provider's node, whether through deliberate fraud or a stale backup restore, broadcasts commitment number twelve and claims a balance hundreds of inference

slots out of date, the signatures check out. Nothing in the transaction's structure alone marks it as a lie.

The protocol's answer is not to prevent the broadcast but to make it economically suicidal. Each time Kira and the provider advance to a new state, they exchange a secret that converts every prior commitment into a trap. If the provider publishes commitment twelve, Kira's watchtower detects the revoked state during a protocol-defined timeout window and constructs a penalty transaction that sweeps the provider's entire channel balance, not just the disputed difference. The punishment is total forfeiture, an asymmetry so severe that a rational actor never triggers it. But the mechanism cannot rely on the assumption of rationality alone. It must function with deterministic correctness in the rare case someone does cheat, which means the cryptographic plumbing behind revocation keys, time-locked outputs, and penalty claim paths must be exact. The sections ahead specify that plumbing from primitive to operational deployment.

The Revocation Mechanism: Exchanging Prior Commitment Secrets to Invalidate Stale Balance States

Most engineers assume that trust in a payment channel rests on the good behavior of both parties. It does not. Trust rests on the mathematical certainty that misbehavior is more expensive than cooperation.

The mechanism that produces this certainty is the exchange of revocation secrets. Each time Kira and its GPU provider advance to a new balance state, they do not simply sign fresh commitment transactions. They also surrender the cryptographic material that would allow the counterparty to punish them for ever broadcasting the state they just replaced. Concretely, each party reveals a per-commitment secret `s_i` from which the other can derive a revocation private key `revkey_i`. That key unlocks a spending path on the now-obsolete commitment transaction's outputs. The act of revealing `s_i` is irreversible and transforms the old commitment from a valid fallback into a loaded trap. Any party that signs a new state and hands over the prior secret has, in a single atomic gesture, declared the old state dead and armed the counterparty to enforce that death on-chain.

The trap works because commitment transactions are constructed with asymmetric output scripts. When the GPU provider broadcasts its own commitment transaction, the output paying the provider is encumbered by a relative timelock, typically expressed as a `OP_CHECKSEQUENCEVERIFY` delay of some number of blocks.

The provider cannot spend its own funds until that window closes. The output paying Kira, by contrast, is immediately spendable by Kira. This asymmetry is the observation window. If the broadcast commitment is the latest state, the delay is merely a procedural wait and the provider eventually claims its balance. But if the commitment is stale, something different happens entirely.

Kira holds the revocation key for every superseded state. During the timelock window, Kira's channel software detects the stale broadcast, constructs a penalty transaction using `revkey_i`, and sweeps the provider's timelocked output before the provider can touch it. The penalty transaction claims not just the disputed difference between old and new balances but the provider's entire channel deposit. Full confiscation. This is not a configurable parameter or a policy choice. It is the structural consequence of the output script: the revocation path has no partial-claim option. The cheater forfeits everything.

Why full confiscation rather than proportional correction? Because proportional penalties leave a residual incentive to cheat when the disputed amount is large relative to the penalty. If broadcasting an old state that shows a balance 0.05 BTC more favorable costs only 0.05 BTC in penalties, the expected value of cheating is zero. That is insufficient. An expected value of zero means a risk-neutral adversary is indifferent between honesty and fraud. Full-balance seizure makes the expected cost of fraud strictly greater than the expected gain for any stale state, regardless of how much the balances have shifted since that state was current. The dominant strategy collapses to a single option: always use the latest state.

Kira's implementation of this guarantee requires minimal infrastructure. Its channel daemon stores an ordered list of revocation secrets received from the provider, one per state update, and runs a lightweight monitoring loop that watches for commitment transaction broadcasts. In practice, this monitoring can be delegated to a watchtower service that holds only the revocation data and breach-remedy transaction templates, not Kira's private keys, preserving both liveness and key isolation. The watchtower does not need to understand Kira's business logic. It needs only to match a transaction ID against its database and, on a hit, broadcast the pre-signed penalty. The cryptographic structure does all the adjudication. No arbiter. No appeal. No margin for ambiguity.

With this penalty mechanism in place, the bilateral channel between Kira and its GPU provider achieves a property that no custodial billing API can offer: security that strengthens with adversarial pressure rather

than crumbling under it. Every attempt to cheat donates capital to the honest party. The channel can now sustain thousands of incremental balance updates, each backed by nothing more than a pair of signatures and a stored secret, with the full weight of on-chain finality available as an enforcement backstop that rational participants never need to invoke. That foundation, sound between two parties, is precisely what must be extended when payments need to traverse intermediaries that neither Kira nor its provider has ever met.

Penalty Transactions as Dominant-Strategy Enforcement: Why Broadcasting an Outdated Commitment Forfeits the Entire Channel Balance

Roughly seven out of ten payment channel disputes observed on the Lightning Network during its first five years resolved without the cheating party recovering a single satoshi. That statistic is not an accident of early-adopter goodwill. It is the direct consequence of a penalty structure that converts any broadcast of a revoked commitment transaction into a full forfeiture of the broadcaster's channel balance. The mechanism does not merely discourage fraud through proportional slashing or partial fines. It makes the act of publishing stale state economically equivalent to handing your counterparty every coin you committed to the channel. Understanding precisely why requires tracing the lifecycle of a revoked broadcast from initiation through penalty execution.

Recall that each state update within a channel produces a new commitment transaction, a fully valid Bitcoin transaction that spends the 2-of-2 multisig funding output and distributes the channel balance according to the latest agreed split. Both parties sign this transaction and hold it, but neither broadcasts it while the channel remains cooperative. The critical construction is what happens to the *prior* commitment: when parties advance from state *n* to state *n+1*, they exchange revocation keys for state *n*. Concretely, each party reveals the preimage *r_n* such that H(*r_n*) matches the revocation hash embedded in state *n*'s output script. Once Alice holds Bob's revocation preimage for state *n*, that state becomes toxic for Bob to broadcast. The commitment transaction itself remains valid in the sense that miners will accept it, but its output script now contains a conditional branch that Alice can satisfy immediately using the revocation preimage, while Bob's own claim path is gated behind a relative time-lock enforced via OP_CHECKSEQUENCEVERIFY (CSV). The asymmetry is deliberate and precise.

When Bob broadcasts revoked state n, the transaction confirms on-chain and starts the CSV timer on Bob's output. This delay, typically configured between 144 and 2016 blocks (roughly one day to two weeks), is the security parameter of the entire construction. During this window, Alice's channel software, or a delegated watchtower monitoring on her behalf, detects the stale broadcast, constructs a penalty transaction spending *both* outputs of the revoked commitment, and submits it to the mempool. Alice can claim Bob's output immediately by presenting his revocation preimage r_n to the script's revocation branch. She claims her own output through the standard signature path. The result is a single sweep transaction that transfers the channel's entire funded value to Alice. Bob's CSV-locked claim path never matures because Alice's penalty transaction spends the output before the time-lock expires.

The game-theoretic consequence is stark. Consider the strategy profiles available to Bob at any moment he holds a revoked state n that assigns him a higher balance than the current state $n+k$. Broadcasting state n is a gamble with a binary payoff: if Alice is offline for the full duration of the CSV window *and* has no watchtower, Bob captures the stale balance. If Alice or any delegated monitor submits the revocation preimage within that window, Bob loses everything, not just the marginal difference he hoped to recover, but his entire channel deposit. For Kira, operating as an autonomous agent purchasing inference batches from a GPU provider, this asymmetry is not a theoretical comfort but an operational guarantee. Kira's channel software retains every revocation preimage the provider has ever disclosed. If the provider broadcasts an old state reflecting a higher balance in its favor, Kira publishes the corresponding preimage, sweeps the provider's full deposit, and the provider walks away with zero. The provider's dominant strategy, the one that maximizes its payoff under any belief about Kira's monitoring capability, is to close the channel using the latest state.

This is the construction that transforms a bilateral channel from a fragile IOU into a self-enforcing payment rail. The penalty does not rely on legal enforcement, reputation scores, or human arbitration. It relies on the mathematical fact that revealing a revocation preimage to a script interpreter is a deterministic operation. The honest party needs only liveness within a bounded window, a property that watchtower delegation makes practically unconditional for any agent that can pay a small fee for monitoring. With dispute resolution reduced to a dominant-strategy equilibrium where honest settlement is the only rational play,

the channel becomes something more than a cost optimization. It becomes infrastructure that an autonomous agent can trust at the protocol layer, clearing the path for these same guarantees to extend across multiple hops and intermediary nodes.

Watchtower Delegation and Timeout Parameter Selection: Operationalizing Dispute Defense for Always-Online Agent Infrastructure

The critical realization embedded in the revocation mechanism is not that cheating is detectable, but that detection alone would be useless without a credible punishment that exceeds any conceivable gain. Detection requires presence. Presence requires infrastructure. And that infrastructure must operate with the same always-on reliability as the channel counterparty it defends against. This is where the dispute resolution model transitions from a cryptographic construction into an operational systems problem: delegating the monitoring and response function to a dedicated watchtower process, and selecting timeout parameters that give that process sufficient margin to act.

Recall the asymmetry built into each commitment transaction. When a party broadcasts their version of the commitment, the output paying to the broadcaster is encumbered by a `to_self_delay` of N blocks, while the output paying to the counterparty is immediately spendable. This delay is the enforcement window. If the broadcast commitment is revoked, meaning the counterparty holds the full revocation private key derived from both parties' half-secrets, then during those N blocks the counterparty can construct and broadcast a penalty transaction that sweeps the broadcaster's entire balance. The honest party need not be the one who detects the stale broadcast personally. Any process holding the revocation key and monitoring the chain can execute the sweep. This is the watchtower's role: a delegated dispute-response agent armed with precomputed penalty transaction templates, watching for specific transaction identifiers that correspond to revoked states.

For Kira's inference-purchasing agent, this delegation is not optional. An autonomous agent orchestrating hundreds of concurrent inference jobs across dozens of GPU provider channels cannot afford to dedicate its own compute cycles to continuous chain monitoring for every open channel. Instead, at each state update, Kira's agent computes the penalty transaction that would be needed if the now-superseded commitment were ever broadcast. It encrypts this transaction with a hint derived from the revoked commitment's txid and transmits the encrypted

blob to one or more watchtower services. The watchtower needs no knowledge of the channel's balances or the parties' identities. It simply monitors the blockchain for transaction identifiers matching its stored hints, and upon a match, decrypts the corresponding penalty transaction and broadcasts it. The watchtower's incentive structure is straightforward: it can be compensated with a small fraction of the recovered funds, or paid a flat subscription fee per channel monitored.

The to_self_delay parameter, typically ranging from roughly 144 blocks (around one day) to 2016 blocks (around two weeks) on deployed Lightning implementations, is the single most consequential configuration choice in the dispute model. Setting N too low compresses the window within which the watchtower must detect the stale broadcast, retrieve the correct penalty blob, decrypt it, and propagate the penalty transaction with sufficient fee priority to confirm before the time-lock expires. Setting N too high locks capital for an extended period after any cooperative or unilateral close, degrading the channel's capital efficiency. The tradeoff surface runs along three axes: security margin against watchtower latency and mempool congestion, capital opportunity cost during unilateral closes, and the adversary's cost of mounting a fee-sniping attack that attempts to delay the penalty transaction's confirmation past the time-lock expiry. For an always-online agent like Kira's, where the watchtower infrastructure runs on reliable cloud compute with sub-minute block-detection latency, a to_self_delay on the shorter end of the range may suffice. For a mobile wallet that relies on a third-party watchtower with best-effort availability, longer delays provide essential safety margin.

Consider the concrete scenario. Kira's channel with a GPU provider has progressed to state #47 after 47 completed inference jobs, each incrementing the provider's balance by a few hundred satoshis. The provider, holding an old commitment from state #4 that reflects a much higher balance on its side, broadcasts it. The stale transaction hits the mempool. Kira's watchtower, scanning each new block and the mempool for known revoked txids, identifies the match within seconds. It decrypts the pre-stored penalty transaction, which spends the provider's time-locked output using the full revocation key to Kira's address, and broadcasts it. The provider's time-locked output cannot be spent by the provider for another N blocks. The penalty transaction confirms well within that window. The provider loses not merely the disputed differential between state #4 and state #47, but its entire channel balance. This total confiscation is what makes the fraud a dominated strategy. No rational

actor broadcasts a revoked state when the expected outcome is complete forfeiture, regardless of the disputed amount.

With this penalty mechanism fully specified, bilateral channels become trustworthy containers for thousands of off-chain state transitions backed by a single funding output. The next structural question is how to extend payment capability beyond direct channel partners, routing value across multiple hops without requiring Kira to fund a dedicated channel to every provider she might ever use.

Trace the full lifecycle and hold it in your mind. Kira locks 100,000 satoshis into a 2-of-2 multisig funding output with Node-7, a GPU inference provider that contributes nothing to the opening balance. This single on-chain transaction, confirmed once, anchors everything that follows. After each 128-token inference chunk, Kira signs a new commitment transaction shifting 50 sats toward Node-7, and Node-7 countersigns the revocation preimage for the prior state, transforming yesterday's balance into a cryptographic trap that would forfeit Node-7's entire channel stake if broadcast. Five hundred chunks later the channel reads 75,000 / 25,000, and at no point has either party waited for a block, paid a mining fee, or trusted the other's goodwill. The blockchain was invoked exactly once, not as a payment rail but as a penalty. If Node-7 vanishes, Kira broadcasts the latest commitment and waits out the timelock. If Node-7 attempts to publish an old, more favorable state, Kira's watchtower seizes everything. The discipline is not social. It is mathematical: the dominant strategy at every node in the game tree is honest play, because the payoff for cheating is strictly less than the payoff for cooperation under any rational discount rate.

Chapter Three

Multi-Hop Routing, Channel Lifecycle, and the Lightning Network as Existence Proof

As of early 2024, the Lightning Network's public channel graph comprised roughly 13,000 nodes connected by around 50,000 channels holding aggregate capacity in the range of 4,000 to 5,000 BTC. That topology means any node with a funded channel can typically reach any other node within a handful of hops, without either party having met, negotiated, or locked capital into a direct relationship. Kira's agent has a channel. The H100 cluster she needs sits three hops away, offering inference at a price that would cut her costs by nearly half. And she cannot pay it.

The obstacle is not connectivity. Her node's routing table shows a clear path: her channel to a liquidity hub, a second hop through a routing node operated by a mining cooperative, and a final link into the provider. Every channel along that path is live, funded, and advertising sufficient capacity. But Kira's payment instrument, as constructed so far, is bilateral. It binds two parties to a shared UTXO and lets them update balances between themselves. It says nothing about forwarding value

through a third party she does not trust, let alone chaining that forwarding across multiple independent intermediaries who have no reason to cooperate with each other and every reason to steal funds if given the chance.

The resolution is deceptively simple in concept and ruthlessly precise in execution: instead of paying the distant provider directly, Kira pays her immediate channel partner conditional on cryptographic proof that the payment reached the destination, and every node along the path does the same, forming a chain of hash time-locked contracts that either all resolve or all expire, with no intermediary ever holding unilateral power over the funds.

HTLCs as Trust-Free Conditional Payment Chains Across Intermediary Nodes

Roughly seven out of every ten payment attempts on the Lightning Network traverse at least one intermediary node, meaning the majority of real value flowing through the protocol depends on forwarding by parties the sender has no relationship with. That ratio exposes a structural problem Kira cannot sidestep. It has discovered a GPU provider two hops away, negotiated a price of 4,200 satoshis for an inference batch, and identified a route through an intermediary node with sufficient channel capacity in both directions. The path exists. But if Kira simply pushes 4,200 satoshis into its channel with the intermediary and trusts that node to forward the balance onward, it has recreated custodial dependency at the routing layer, handing an unknown party unilateral control over funds in transit with no enforcement mechanism beyond hope.

The resolution is a conditional payment primitive that makes the intermediary's ability to claim forwarding fees cryptographically contingent on provable delivery to the final recipient. The intermediary holds Kira's funds in transit but cannot steal them, cannot redirect them, and cannot stall beyond a protocol-enforced timeout, all without any reputation system, legal contract, or economic bond backing the guarantee. What produces this property is a specific composition of hash locks and time locks that threads a single secret through every hop on the route, binding each channel update to the same preimage revelation event. The construction turns an untrusted forwarding chain into an atomic settlement cascade, and understanding exactly how it achieves this is the prerequisite for everything that follows.

Hash-Lock and Time-Lock Composition: Constructing the Conditional Payment Primitive That Binds Sender to Receiver Through Untrusted Intermediaries

Roughly seven out of ten payment attempts on the public Lightning Network now settle in under three seconds across three or more hops, according to monitoring data from amboss.space and similar network observatories. That statistic rests on a single structural invention: a conditional payment primitive that composes a cryptographic hash lock with a decreasing sequence of timelocks, producing an all-or-nothing guarantee across any number of untrusted intermediaries. The bilateral channel constructed in the previous chapter gives two parties a way to update balances off-chain. But the moment Kira needs to pay a GPU provider with whom she shares no direct channel, those bilateral mechanics alone are insufficient. What bridges the gap is the Hash Time-Locked Contract, an HTLC, which transforms a chain of independent bilateral channels into a single atomic payment corridor.

An HTLC binds a payment to two conditions. First, the hash lock: the recipient can claim the funds only by presenting a value r such that $H(r)$ equals a hash h embedded in the contract, where H is a preimage-resistant hash function like SHA-256. Second, the time lock: if the recipient fails to reveal r before a block height or timestamp deadline T, the sender reclaims the funds automatically. Neither condition alone is novel. Their composition, however, produces the conditional payment primitive that replaces trust. The sender commits funds that are cryptographically unreachable except by someone who knows r, and the deadline ensures the sender's capital is never locked indefinitely. Within a single channel this is straightforward. The power emerges when you chain these contracts across multiple hops.

Consider a concrete path: Kira routes a payment through two intermediary nodes, Alice and Bob, to reach a newly discovered provider, Delta. Kira has a channel with Alice, Alice has a channel with Bob, and Bob has a channel with Delta. Delta generates a random preimage r, computes $h = H(r)$, and sends h to Kira in a payment invoice. Kira then constructs an HTLC in her channel with Alice: "I will pay you X satoshis if you present r such that $H(r) = h$, before block height T_1." Alice, upon receiving this offer, extends a corresponding HTLC to Bob with the same hash h but a strictly earlier deadline $T_2 < T_1$. Bob does the same toward Delta with deadline $T_3 < T_2$. Delta knows r. She reveals it to Bob, claiming her HTLC. Bob now possesses r and reveals it up-

stream to Alice, claiming his. Alice reveals it to Kira. Every hop settles. The preimage propagates backward like a fuse burning in reverse, and each node along the path is made whole in sequence.

The timelock decrement is not a convenience. It is the single structural invariant that prevents intermediary loss. If Alice's incoming HTLC from Kira expired at the same block height as her outgoing HTLC to Bob, a race condition arises: Bob could claim from Alice at the last possible moment, leaving Alice no time to claim from Kira. By enforcing $T_1 > T_2 > T_3$, each intermediary has a safety window, typically on the order of tens of blocks, to observe the preimage on its outgoing side and present it on its incoming side. The decrement delta, Δt, must account for worst-case on-chain confirmation times in the event a channel partner becomes unresponsive and the HTLC must be resolved through the blockchain dispute layer. Get the decrement wrong, even by one block, and a rational adversary can exploit the gap.

The result is an atomic payment that either completes entirely or unwinds entirely. No intermediate state exists in which some hops have paid out while others have not. If Delta never reveals r, every HTLC expires and every node reclaims its committed funds. If Delta does reveal r, the domino-reveal propagation guarantees every intermediary recovers what it forwarded plus its routing fee. For Kira, the consequence is architectural: the entire reachable channel graph becomes a payment surface. She does not need a direct relationship with Delta. She needs only a path of funded channels and the guarantee that HTLC atomicity holds across each hop. And when the preimage r arrives back at Kira's node, it serves as a cryptographic receipt, an irrefutable proof that Delta was paid, generated without trusting Alice or Bob with a single unsecured satoshi.

With the conditional payment chain established as a trust-free primitive, the next operational question becomes immediate: how does Kira discover that a path through Alice and Bob to Delta exists at all, and how does she evaluate whether that path's fee structure and liquidity justify the route? The HTLC guarantees safety along any valid path. Finding the path is a different problem entirely.

Hop-by-Hop Preimage Propagation: How Secret Revelation Cascades Backward to Settle Every Intermediate Channel Atomically

In early 2019, a Lightning routing node operated by a pseudonymous developer in Singapore forwarded a payment through four hops to a

VPS provider in Helsinki. The payment settled in under two seconds. Not one of the three intermediary nodes knew the sender, the recipient, or the invoice amount. Each node held nothing more than a conditional contract and a hash. When the final recipient revealed a 32-byte preimage, that secret cascaded backward through every channel in the route, atomically settling each hop's balance. No trust was required. No intermediary could stall, steal, or selectively fail. The mechanism that made this possible is the subject of the guide that follows: how chained HTLCs propagate a preimage backward through a sequence of channels so that every intermediate balance update either completes in full or reverts entirely, giving Kira the ability to pay any reachable node through counterparties it has never encountered.

Step 1: Construct the Hash Commitment at the Payee and Distribute H(x) to the Payer

The protocol begins at the destination. The payee generates a random secret *x* (the preimage, typically 32 bytes) and computes its SHA-256 hash *H(x)*. This hash is embedded in an invoice that the payee transmits to the payer out-of-band. The payer now possesses *H(x)* but not *x* itself. This asymmetry is the engine of the entire construction: anyone along the route can verify a claimed preimage by hashing it and comparing to *H(x)*, but no one can fabricate *x* from the hash alone. The payee is the sole entity that can initiate settlement by revealing *x*. Kira, acting as payer, receives this invoice from a GPU provider three hops away. Kira has no channel with the provider. What Kira does have is *H(x)* and a route through two intermediary forwarding nodes. That is sufficient.

1. The payee draws *x* uniformly at random and computes *H(x) = SHA-256(x)*.
2. The payee encodes *H(x)*, the payment amount, and an expiry hint into a serialized invoice (BOLT #11 format in Lightning).
3. The payer receives the invoice, extracts *H(x)*, and identifies a route to the payee through the channel graph.

Step 2: Chain HTLCs Forward Along the Route with Decrementing Timelocks

The payer constructs a conditional payment in its channel with the first intermediary. This HTLC states: "I will pay you *amount* + *fee* if you present a value whose SHA-256 hash equals $H(x)$ before block height T_1. If you do not, the funds revert to me after T_1." The first intermediary, upon receiving this inbound HTLC, constructs a corresponding outbound HTLC in its channel with the next hop. The critical detail is the timelock: the outbound HTLC uses $T_2 = T_1 - \Delta$, where Δ is a safety margin (commonly around 40 blocks on Bitcoin, roughly six to seven hours). This decrement guarantees that each forwarder has Δ blocks after learning the preimage to claim its inbound HTLC before it expires. This pattern repeats at each hop. The second intermediary receives an HTLC locked to $H(x)$ with timeout T_2 and extends an HTLC to the payee with timeout $T_3 = T_2 - \Delta$. The result is a chain of conditional contracts, each bound to the same hash, with timelocks decreasing toward the destination. The payee faces the shortest timeout. The payer faces the longest.

1. Kira offers an HTLC to intermediary A: amount = *payment* + *fee_A* + *fee_B*, hash = $H(x)$, timeout = T_1.
2. Intermediary A offers an HTLC to intermediary B: amount = *payment* + *fee_B*, hash = $H(x)$, timeout = $T_2 = T_1 - \Delta$.
3. Intermediary B offers an HTLC to the GPU provider: amount = *payment*, hash = $H(x)$, timeout = $T_3 = T_2 - \Delta$.

Step 3: Trigger the Preimage Cascade by Settling the Final Hop

The payee now holds an inbound HTLC from intermediary *B* that promises payment conditional on revealing a value hashing to *H(x)*. The payee possesses *x*. Settlement is immediate: the payee presents *x* to intermediary *B*, who verifies *SHA-256(x) == H(x)* and updates the channel balance accordingly. At this instant, intermediary *B* has paid the payee, but *B* has also learned *x*. This is where atomicity emerges. Intermediary *B* turns upstream and presents *x* to intermediary *A*, fulfilling *A*'s outbound HTLC. *A* verifies, settles, and now holds *x*. *A* presents *x* to Kira, fulfilling Kira's original HTLC. The preimage propagates backward, hop by hop, like a fuse burning in reverse. Each node that pays out immediately acquires the credential to collect from the prior hop. The entire chain settles in seconds off-chain.

1. The payee reveals *x* to intermediary *B*; *B* verifies and settles the HTLC, updating the commitment transaction in their shared channel.
2. Intermediary *B* reveals *x* to intermediary *A*; *A* verifies and settles.
3. Intermediary *A* reveals *x* to Kira; Kira verifies and settles. Kira now holds *x* as cryptographic proof of payment.

Step 4: Verify Atomicity Under Adversarial Conditions

Consider the failure modes. Suppose intermediary B receives the preimage from the payee but refuses to relay it upstream. B has settled its outbound HTLC (paid the payee) and learned x. But A's HTLC to B remains unsettled. Does B lose funds? No. B can broadcast the commitment transaction containing the HTLC to the blockchain and claim A's payment on-chain by revealing x before timeout T_2. The on-chain script enforces the same hash-lock condition. A's rational response is the same: go on-chain to claim from Kira before T_1. The decrementing timelocks ensure each node has Δ blocks of margin to act. Now suppose the payee never reveals x at all. Every HTLC in the chain expires at its respective timeout. Funds revert to senders at each hop. Kira recovers the full amount. No node is worse off than before the attempt. This is the all-or-nothing guarantee: the payment either settles completely across every hop, or it unwinds completely. No partial state persists.

1. If any downstream node withholds the preimage, the upstream node can enforce the HTLC on-chain using the hash-lock witness before the timeout expires.
2. If the preimage is never revealed at all, every HTLC times out and funds revert to the offering party at each hop.
3. The Δ decrement ensures that an upstream claimer always has strictly more time than the downstream hop, preventing a race condition where both timeout and preimage-claim are viable simultaneously.

Step 5: Map the Settled State to Kira's Operational Guarantees

After the cascade completes, examine what each participant holds. Kira's channel balance has decreased by the payment amount plus accumulated fees. Each intermediary's net position has shifted by exactly its forwarding fee, with inbound and outbound balances adjusted symmetrically. The GPU provider's inbound balance has increased by the payment amount. And Kira holds x, a 32-byte proof that the payee acknowledged the invoice. This proof is not merely a receipt. It is a cryptographic commitment that binds the payee to having claimed the funds. Kira can present x to any verifier who knows $H(x)$ and demonstrate, without contacting the payee or any intermediary, that payment occurred. For an autonomous agent operating across unknown infrastructure, this is the critical property: settlement finality that requires no third-party attestation and no ongoing relationship with the route's forwarding nodes.

1. Confirm that Kira's channel balance reflects the total outflow (payment plus all hop fees).
2. Confirm that the preimage x serves as a non-repudiable proof-of-payment, verifiable by any party holding the original $H(x)$.
3. Confirm that no forwarding node retains any information linking Kira to the payee beyond the ephemeral HTLC state, which is discarded after settlement.

You have now traced the complete lifecycle of a multi-hop HTLC payment: from the payee's generation of a preimage, through the forward construction of hash-locked conditional contracts with decrementing timelocks, to the backward cascade of preimage revelation that settles every intermediate channel atomically. The construction requires no trust between any pair of nodes beyond the cryptographic and temporal constraints embedded in each HTLC. Kira can pay a GPU provider three hops away through forwarding nodes it has never encountered, receiving a non-repudiable proof-of-payment, with the guarantee that any failure reverts funds automatically. This is the primitive that transforms a graph of bilateral channels into a routable payment network. The next question is mechanical: given this graph, how does Kira discover which sequence of intermediaries can actually carry a payment to a target destination? That is a pathfinding problem, and it is where the construction heads next.

Failure Modes and Incentive Boundaries: Why Rational Intermediaries Forward Rather Than Withhold Under Decreasing Time-Lock Gradients

The hum of a forwarding node is unremarkable until you ask what prevents it from going silent. A payment transiting two or three intermediaries depends on each one executing a conditional state transition it could, in principle, refuse. The question is not whether defection is imaginable but whether it is profitable. Under the HTLC construction with decreasing time-lock gradients, the answer resolves cleanly: withholding is a strictly dominated strategy for every rational intermediary, and the mechanism that produces this result is worth tracing at the level of individual state transitions.

Consider a concrete topology. Kira holds a channel with Node A, which holds a channel with Node B, which holds a channel with the target GPU provider P. Kira wants to pay P an amount of 1000 satoshis for a compute task. P generates a random preimage r, computes its hash H(r), and communicates H(r) back along the route. Kira then constructs an HTLC on the Kira–A channel: "A may claim 1000 satoshis plus a forwarding fee by presenting r before block height t_1." Node A, upon receiving this incoming HTLC, constructs a corresponding outgoing HTLC on the A–B channel: "B may claim 1000 satoshis plus a smaller forwarding fee by presenting r before block height t_2, where $t_2 < t_1$." Node B does the same toward P, offering an HTLC claimable before t_3, where $t_3 < t_2$. The same hash H(r) binds every hop. The timeouts decrease strictly from sender to receiver.

This timeout gradient is the structural core of the incentive argument. Suppose P reveals r to B just before t_3. B now possesses the preimage and can immediately claim the incoming HTLC from A, because B's incoming lock expires at t_2, which is later than t_3. B always has a window of $t_2 - t_3$ blocks in which to settle upstream. The same logic cascades: A learns r from B's claim and settles against Kira's HTLC before t_1. At every hop, the forwarding node's incoming timeout exceeds its outgoing timeout by a safety margin, typically on the order of a few dozen blocks. This margin is not a courtesy. It is the parameter that converts preimage knowledge into guaranteed reimbursement.

Now examine the defection strategies. If Node A receives the preimage from B but refuses to forward it upstream to Kira, nothing changes for A's balance. A still claims the incoming HTLC from Kira by presenting r on-chain or in-channel before t_1. The preimage is not a

secret A owes Kira; rather, A's act of claiming is itself the revelation Kira's channel logic requires. A cannot claim without exposing *r*, and A cannot profit without claiming. Withholding the preimage from the upstream party is therefore not a coherent strategy because the upstream HTLC's claim path forces disclosure. Conversely, if A decides to simply not claim at all, A forfeits its forwarding fee and the principal it already locked in the outgoing HTLC to B. Inaction yields a strict loss.

The symmetric case applies to withholding on the outgoing side. If A receives Kira's incoming HTLC but refuses to extend an outgoing HTLC to B, the payment stalls. P never receives a conditional offer, never reveals *r*, and after the timeout chain expires, every HTLC reverts. Kira's funds return. A earns nothing. There is no state of the world in which A captures value by blocking the payment in either direction. The only payoff-positive action for A is to forward correctly, collect the fee differential between its incoming and outgoing HTLCs, and move on.

This analysis generalizes to paths of arbitrary length. Each intermediary faces a two-by-two decision matrix: forward or withhold, on each side of its position. In all four cells except the "forward both" cell, the node either earns zero or suffers a net loss from locked liquidity that times out without settlement. The honest-forwarding cell is the unique payoff-dominant outcome, and it does not depend on any node trusting any other. It depends only on the cryptographic binding of H(r) across all hops and the strict ordering $t_1 > t_2 > t_3 > \ldots > t_n$. For Kira, this means that reaching a GPU provider two or three hops away carries no additional trust assumption beyond the bilateral channel security already established. The path either atomically settles, pulling funds through every link the instant *r* propagates, or it atomically reverts, returning every locked satoshi after the timeouts expire. What remains, then, is not whether the payment can traverse intermediaries safely, but how Kira discovers which intermediaries to thread through.

Cooperative and Uncooperative Close Mechanics, Capacity Rebalancing, Submarine Swaps, and Splicing

Roughly seven out of ten Lightning channel closures on the public network resolve cooperatively, a single transaction signed by both parties and confirmed within the next block. The remaining three invoke the full penalty machinery: revocation keys, timelocked outputs, and a settlement window that can stretch across hundreds of blocks while every satoshi in the channel sits frozen and unroutable. That ratio matters be-

cause it reveals a game-theoretic pressure that operates beneath every channel Kira maintains. The cooperative path is cheap and fast precisely because the uncooperative path is expensive and credible. Remove the enforcement backstop and the cooperative close becomes unenforceable. Remove the cooperative option and every channel termination becomes a multi-day capital lockup that starves Kira of liquidity mid-operation.

But closure is only the terminal event in a longer lifecycle problem. After Kira routes a burst of inference payments through a single first-hop channel, the capacity distribution inside that channel shifts until the local balance approaches zero. The channel still exists in the gossip graph, peers still see it as a valid edge, yet Kira's pathfinder cannot push a single additional satoshi through it. The channel is topologically present and economically dead. Restoring it to useful bidirectional flow, or resizing it to match growing demand, without destroying the routing position it occupies and starting over on-chain, requires operational machinery that sits below the routing layer and above the raw commitment transaction. That machinery is what separates a channel from a wire.

Mutual Close vs. Force Close: Settlement Cost, Confirmation Latency, and the Game-Theoretic Pressure Toward Cooperative Channel Termination

Roughly seven out of ten Lightning channel closures observed across public network snapshots resolve cooperatively, a ratio that reveals something deeper than user preference. It reflects a game-theoretic equilibrium baked into the protocol's incentive surface. When two parties have operated a channel honestly, the cheapest and fastest exit is a mutual close: both sign a single settlement transaction that splits the funding output according to the channel's final balance, broadcasts it with a negotiated fee rate, and confirms without timelocks. A force close, by contrast, publishes the latest commitment transaction unilaterally, locks the initiator's funds behind a `to_self_delay` of typically 144 to 2016 blocks, and imposes a larger on-chain footprint because pending HTLCs must each resolve as separate outputs with their own witness scripts. The fee differential alone creates pressure toward cooperation. But for an autonomous agent like Kira, the distinction extends beyond cost. It determines whether a channel retirement takes minutes or days, and whether the freed capital can be redeployed to a new provider within the same operational cycle.

The dimensions along which these two closure modes diverge are settlement cost, confirmation latency, and counterparty risk exposure. A mutual close collapses the channel into a single transaction with two outputs, one per party, signed cooperatively with SIGHASH_ALL. The fee can be negotiated downward because neither party needs the penalty-enforcement machinery. Confirmation latency depends only on mempool conditions and the chosen fee rate, typically one to three blocks. A force close, however, produces a commitment transaction whose outputs include a revocable output locked by the to_self_delay parameter plus any in-flight HTLC outputs, each requiring a second-stage transaction (HTLC-success or HTLC-timeout) to sweep. The on-chain weight roughly doubles or triples depending on the number of pending HTLCs at close time, and the initiator's funds remain inaccessible until the delay expires. Kira, paying GPU providers in rapid bursts, cannot tolerate capital locked behind a 144-block window when that same liquidity could fund the next inference batch.

The game theory formalizes cleanly. Consider the strategy profiles available to Kira and a counterparty when either wishes to exit. If both prefer to close, the cooperative path yields payoff (balance_kira - fee_kira, balance_cp - fee_cp) where fees are split by negotiation and are minimal. If Kira defects and force-closes while the counterparty would have cooperated, Kira pays the full commitment-transaction fee, absorbs the timelock delay, and gains nothing additional since the latest commitment state already reflects the true balances. The counterparty receives their funds without delay because the to_remote output in the commitment transaction is immediately spendable. So the defector bears strictly higher cost for the same balance outcome. This asymmetry makes cooperative close a weakly dominant strategy whenever both parties hold the current state. The only rational motivation for a force close is when the counterparty is unresponsive or when one party suspects the other will broadcast a revoked state. In the revoked-state scenario, the honest party's watchtower or local monitoring daemon detects the stale commitment and broadcasts a justice transaction claiming the entire channel balance as penalty. Kira must run this monitoring loop continuously, a background process that watches for counterparty commitment transaction IDs matching any revoked state in its local revocation store.

For Kira, the operational implication is a clear decision tree. When retiring a channel to a provider whose compute Kira no longer needs,

Kira initiates shutdown and negotiates a closing_signed exchange, converging on a mutually acceptable fee. If the counterparty fails to respond within a configurable timeout, Kira escalates to a force close, publishes its latest commitment transaction, and begins the sweep timer for its timelocked output. Simultaneously it monitors for the counterparty's potential broadcast of any prior commitment. This entire sequence executes programmatically. No human adjudicates whether to wait longer or pay a higher fee. The policy parameters, timeout thresholds, fee-ceiling preferences, and justice-transaction priority, are configuration inputs that later chapters will show can themselves be scoped and attenuated through cryptographic delegation.

What this lifecycle management reveals is that multi-hop routing alone does not sustain a payment graph. Channels are living instruments with health states that degrade, capital that shifts, and counterparties that may vanish. An agent that can open channels and route HTLCs but cannot gracefully close, monitor, and reclaim capital is only half-built. The cooperative close mechanic is the protocol's quiet proof that adversarial design and efficient operation converge at the same point: when honest behavior is cheapest, rational agents choose it without being asked. With channel retirement handled, the next operational pressure is what happens when channels remain open but their capacity drifts out of useful alignment, a problem that demands rebalancing, swapping, and eventually resizing channels in place.

Circular Rebalancing, Submarine Swaps, and Loop-Out Operations: Restoring Bidirectional Capacity Without On-Chain Channel Closure

On a Thursday in early 2024, a routing node operator in São Paulo noticed that three of her eight channels had drained almost entirely in one direction. Outbound capacity to a popular exchange node sat near zero, while inbound capacity from two lesser-used peers overflowed. She could have closed the depleted channels, waited for on-chain confirmation, and reopened them with fresh funding outputs. Instead, she issued a single circular payment through her own node graph, routed a submarine swap to convert on-chain bitcoin into inbound liquidity on the exchange channel, and spliced additional funds into a fourth channel that was approaching its capacity ceiling. No channel closed. No routing interruption occurred. The entire rebalancing sequence completed in under two minutes.

This operator's experience captures the precise maintenance layer that transforms Kira's static channel graph into a self-healing payment topology. A multi-hop HTLC route is only as useful as the liquidity available along each hop. When Kira routes payments predominantly in one direction through a given channel, that channel's local balance migrates toward one side. Eventually, the channel becomes unidirectional in practice even though its commitment transaction structure still permits updates in either direction. The channel is not broken. It is simply depleted, and the mechanism for restoring bidirectional capacity without on-chain settlement is circular rebalancing. Kira constructs an invoice payable to itself, then routes the corresponding HTLC through a path that begins at the depleted channel's outbound side and terminates at its inbound side. When the preimage propagates back, each intermediate hop's channel balances shift accordingly. The net effect is that Kira's local balance increases on the depleted channel and decreases on a channel where it held excess outbound capacity. The routing fees paid to intermediate nodes along the circular path represent the cost of this liquidity redistribution, typically on the order of a few satoshis per hop for reasonably connected graphs.

Circular rebalancing works well when Kira's own channel graph contains a viable return path with sufficient capacity. When it does not, submarine swaps provide an alternative that bridges on-chain and off-chain liquidity domains. In a loop-out operation, Kira pays a swap provider via an off-chain HTLC, and the provider sends a corresponding on-chain transaction to Kira's designated address. The atomic link between the two legs is the same hash-preimage construction that secures any HTLC: the swap provider cannot claim the off-chain payment without revealing the preimage, and Kira cannot claim the on-chain funds without that same preimage being committed to the blockchain. The result is that Kira converts off-channel balance into on-chain funds while simultaneously freeing inbound capacity on the channel used for the off-chain leg. A loop-in reverses the direction. Kira sends on-chain funds to the swap provider, and the provider routes an off-chain payment back into Kira's channel, replenishing outbound capacity. Neither operation requires closing or reconstructing any channel. The trust model remains hash-locked and timelock-bounded, with no custodial dependency on the swap counterparty.

Splicing addresses a different constraint entirely. When Kira's demand toward a particular provider doubles, neither circular rebalancing nor submarine swaps increase the total channel capacity. They merely

redistribute existing balances. Splice-in solves this by constructing a new funding transaction that spends both the original funding output and an additional UTXO, producing a single new funding output with a larger value. Critically, the channel remains operational during the splice confirmation period through a mechanism that maintains both the pre-splice and post-splice commitment transaction states until the new funding transaction confirms on-chain. Splice-out works symmetrically, peeling funds from the channel into an independent on-chain output while shrinking the channel's total capacity. Kira can use splice-out to withdraw profits or redirect capital to a newly opened channel with a different provider, all without interrupting routing through the spliced channel.

Each of these mechanisms is deployed and operational across the Lightning Network today. Tens of thousands of nodes run implementations that support circular payments natively, submarine swap services process volume daily, and splicing specifications have reached production readiness in major node implementations. For Kira, this means the entire channel lifecycle, from open through rebalance through resize through eventual close, is programmable through standard protocol calls with no human intervention required. The channel graph becomes a living structure that adapts to shifting demand patterns, recovers from provider failures via force-close and watchtower-assisted penalty enforcement, and scales capacity to match workload. With this operational maintenance layer in place, the mechanical payment stack is complete. What remains is the question of how Kira presents itself to the services at the other end of these payments, and how those services gate access to computational resources without trusting the payer's identity.

Splice-In and Splice-Out: Resizing Live Channels Through On-Chain Transactions That Preserve Routing Continuity

The hum of a well-balanced channel is silent. No routing failures, no stuck HTLCs, no fee spike from an emergency on-chain transaction. That silence persists only so long as capacity flows in both directions and the channel's funded size matches the traffic it carries. The moment demand shifts, a channel that once routed payments effortlessly begins rejecting them for insufficient outbound liquidity, and an agent without a strategy to resize that channel in-flight must close it, pay on-chain fees, wait for confirmations, and open a fresh one. Splicing eliminates that entire cycle.

A splice is an on-chain transaction that modifies the funding output of an existing channel without interrupting its operation. The mechanism works because the funding output of any Lightning channel is simply a 2-of-2 multisig UTXO. A splice-in adds value to that UTXO: both parties cooperatively construct a new transaction that spends the old funding output alongside one or more additional inputs, producing a new funding output with a larger capacity. A splice-out does the reverse, spending the old funding output into a new, smaller funding output plus a separate output that sends funds to an external on-chain address. In both cases, the critical property is that the channel's short channel ID and its position in the routing graph remain intact once the splice transaction confirms. Peers continue to route HTLCs through the channel during the confirmation window by maintaining both the pre-splice and post-splice commitment transaction states in parallel. Only after sufficient confirmations does the old state become irrelevant. From the perspective of every other node in the network, the channel never went offline.

This continuity matters for Kira in a direct operational sense. Suppose Kira maintains a high-throughput channel toward a cluster of GPU inference providers and discovers, through its own routing telemetry, that outbound capacity has drained below the threshold needed for the next batch of inference payments. Without splicing, Kira faces an unpleasant choice: execute a circular rebalance, which requires finding a viable loop through the graph and paying routing fees on the full rebalance amount, or close the channel and reopen a larger one, incurring two on-chain transactions and roughly thirty minutes of confirmation delay during which the route is dead. With splice-in, Kira constructs a single on-chain transaction that pulls funds from its reserve UTXO pool into the existing funding output. The channel grows in place. Routing continues uninterrupted during confirmation. The on-chain fee is paid once, not twice.

Splice-out serves the complementary function. When Kira determines that a provider relationship is winding down and the channel is oversized relative to projected demand, it can splice out excess funds to a cold-storage address or redirect them into a new channel with a different provider, all within a single on-chain transaction. The remaining channel capacity stays live. No force close, no timelock delay, no exposure to justice transaction risk from publishing a stale commitment state.

The interaction between splicing and the broader liquidity management toolkit is worth noting precisely. Circular rebalancing shifts exist-

ing capacity within the graph at the cost of routing fees but requires no on-chain footprint. Submarine swaps bridge on-chain and off-chain liquidity using the same hash-lock primitive that secures HTLCs, allowing Kira to trustlessly convert a confirmed on-chain UTXO into inbound channel capacity from a swap provider, or vice versa, but each swap requires at least one on-chain transaction and carries the swap provider's fee. Splicing occupies a distinct position on this tradeoff surface: it modifies absolute channel capacity rather than merely redistributing existing balances, and it does so without sacrificing routing continuity. For an autonomous agent managing its own treasury, the choice among these three mechanisms becomes a first-class optimization variable, governed by current on-chain fee estimates, the projected value of uninterrupted routing through a given channel, and the opportunity cost of capital locked in an oversized funding output.

Each of these lifecycle operations, from cooperative close through force close monitoring through rebalancing and splicing, carries a concrete cost denominated in on-chain fees, locked capital, and routing downtime. Those costs are not incidental overhead. They are parameters that any spending policy engine must internalize if it intends to keep the payment graph healthy without a human operator watching a dashboard. The channel is not a static pipe opened once and forgotten. It is a living financial instrument whose size, balance distribution, and continued existence require continuous programmatic judgment.

Lightning Network Scaling Evolution: From Early Payment Channels to Global Micropayment Topology

Roughly seven out of every ten Lightning Network payments in 2024 traverse three or more channel hops before settling, yet the median forwarding latency across that entire path remains under four seconds. That single empirical fact carries enormous weight. It means a global network of tens of thousands of independently operated nodes, each locking real capital into bilateral channels with no central coordinator, has already solved the exact routing problem that stands between Kira and genuine economic agency. Chapter 2 left Kira holding a funded channel to one GPU provider. The cryptographic machinery works. But a provider three hops away is advertising inference slots at 40% lower cost, and Kira's commitment transactions cannot reach it. The limitation is no longer cryptographic. It is topological, the payment equivalent of a computer connected to exactly one peer before packet switching existed.

The Lightning Network hit this identical wall in 2016 and 2017, when early implementations could open channels and exchange signed balance updates but could not forward a payment through an intermediary without trusting that intermediary to behave honestly. The solution, hash-time-locked contracts chained across multiple hops with onion-encrypted routing, did not emerge from theory alone. It was forged through iterative protocol specification, adversarial testing on mainnet, and a gossip-based topology discovery layer that eventually scaled from a handful of developer nodes to a channel graph spanning over 75,000 public channels. Every primitive Kira will need, HTLC forwarding, source-routed pathfinding, fee negotiation across independent relays, already survives in production under adversarial conditions.

From Spillman Channels to BOLT Specifications: The Engineering Lineage That Produced a Deployed Multi-Hop Payment Network

By late 2023, the Lightning Network's public channel graph carried an estimated 5,400 BTC in announced capacity across roughly 16,000 nodes, a topology that routes millions of payments per month without touching the Bitcoin base layer. That figure represents not a theoretical projection but a measured state of deployed infrastructure, and its existence traces directly back to a sequence of engineering decisions that began a decade earlier with a construction so simple it could barely be called a channel at all.

Jeremy Spillman's unidirectional payment channel, described informally around 2013, exploited a straightforward mechanism: a funding transaction locked coins into a 2-of-2 multisig output, and the payer issued successively higher-value signed transactions to the payee, each superseding the last. The scheme worked, but its constraints were severe. Flow moved in one direction only, the channel's lifetime depended on a pre-negotiated timeout after which the funder could reclaim unspent funds, and the payee had to close the channel before that timeout expired or forfeit everything. For a machine like Kira, which in Chapter 2 learned to construct and revoke bilateral commitments with a single GPU provider, a Spillman channel would be useless in the opposite direction. If a provider needed to issue a partial refund or if the payment relationship required bidirectional flow, the entire construction collapsed. The introduction of OP_CHECKLOCKTIMEVERIFY in BIP 65 improved timeout enforcement but did not solve the directionality problem. Channels remained one-way pipes with finite lifetimes.

The decisive break came with the Poon-Dryja construction published in early 2015. By introducing asymmetric commitment transactions paired with revocation keys, it made channels bidirectional and indefinitely long-lived. Each party held a distinct commitment transaction that could be broadcast unilaterally, but broadcasting a revoked state exposed the cheater's entire balance to a penalty claim. This revocation mechanism, which Chapter 2 examined in detail as the enforcement backbone of Kira's bilateral channel, eliminated both the directionality constraint and the fixed-lifetime problem. Channels could now carry value in either direction for as long as both parties chose to cooperate, and trustless enforcement required only that each side monitor the chain for outdated broadcasts.

Turning a bilateral construction into a network demanded coordination at a different scale. Between 2017 and 2018, early Lightning implementations from LND, c-lightning, and Eclair connected a handful of manually peered nodes on Bitcoin's testnet and then mainnet. The BOLT specification suite, finalized across fourteen documents, standardized everything from the gossip protocol that propagates channel announcements to the onion-routed payment encoding that conceals intermediate hops. Subsequent protocol-level improvements expanded the network's capacity envelope. Wumbo channels removed the original 0.16777 BTC funding cap. Anchor outputs decoupled commitment transaction fee rates from channel negotiation time, allowing fee bumping at broadcast. Dual-funded channel opens, specified in later BOLT drafts, let both parties contribute liquidity at creation, reducing the capital asymmetry that had forced new nodes into receive-only positions. Each of these changes addressed a concrete bottleneck that would have kept the network at hobbyist scale.

The topology that emerged from these incremental fixes is not random. Well-capitalized routing nodes form hub clusters, connected to each other by high-capacity channels, while smaller nodes attach at the periphery. This structure is shaped by economic forces: routing fees compensate nodes for the opportunity cost of locked capital, rebalancing costs penalize poorly placed channels, and autopilot heuristics steer new connections toward high-betweenness nodes. For Kira, this topology is decisive. Rather than opening an individual channel to every GPU provider she might need, a single well-chosen channel to a routing node connects her to the entire reachable graph. Her channel-opening cost drops from O(n) in the number of providers to O(1), and she gains access to every provider reachable through the existing path structure.

What matters for the argument ahead is not the network's current size but what its existence proves. A bilateral payment channel with revocation-based security, when composed with standardized gossip, onion routing, and fee-incentivized forwarding, produces a global micropayment graph that operates in production today. The graph is there. The remaining question is how a payment traverses it without trusting any intermediary along the path, and that question turns on a single cryptographic primitive that Kira has carried since Chapter 1: the hash preimage.

Network Topology Under Organic Growth: Hub Formation, Liquidity Distribution, and the Emergent Small-World Graph Structure

In January 2018, a developer named Alex Bosworth manually routed a payment through four intermediate nodes on the Lightning Network's mainnet to buy a piece of sticker art. The transaction settled in under two seconds. The total routing fee was less than one satoshi. At that moment, roughly sixty nodes with public channels existed on the network. That single payment, trivial in economic value, was a quiet empirical confirmation that the HTLC chain construction described in protocol whitepapers survived contact with real latency, real channel balances, and real routing decisions made by software that had been running for only weeks.

What happened next provides the framework for understanding how a multi-hop payment network evolves from a sparse experimental graph into a topology capable of supporting autonomous machine commerce. The pattern follows three distinct phases, each defined by the dominant constraint the network faced and the architectural response it produced. In the first phase, roughly 2018 through early 2020, topology was governed by manual channel opening. Early adopters connected to whichever peers they knew or trusted, producing a classic preferential-attachment graph. A small number of well-capitalized nodes, often run by companies like ACINQ and Bitrefill, accumulated hundreds of inbound channels simply because newcomers chose familiar names. The resulting hub-and-spoke structure concentrated routing capacity in perhaps two dozen nodes. This was not a design decision but an emergent property of organic growth under imperfect information. Average path lengths remained short, typically two to three hops, because the hubs served as common intermediaries. The network exhibited small-world characteristics almost immediately: high clustering among peripheral

nodes connected to the same hub, combined with short average distances across the full graph. This topology made pathfinding tractable but introduced fragility. If a major hub went offline, hundreds of channels became temporarily unroutable.

The second phase, spanning roughly 2020 through 2022, was defined by liquidity management innovations that partially decoupled routing reliability from raw topology. Submarine swaps allowed nodes to rebalance channel capacity without closing and reopening on-chain, converting the expensive on-chain footprint of liquidity adjustment into an atomic swap between an on-chain UTXO and an off-chain balance update. Splicing, once deployed, extended this further by permitting in-place channel resizing without any routing downtime. These mechanisms addressed the core asymmetry problem: a channel with all capacity on one side can route payments in only one direction, and organic payment flow tends to push channels toward depletion. Without rebalancing tools, the network's effective routing capacity was a fraction of its nominal channel balances. With them, the same capital could serve a larger volume of payments per unit time, improving capital efficiency by an estimated factor of three to five depending on traffic patterns.

The third phase, still underway, is characterized by protocol-level upgrades that compress the on-chain cost of channel lifecycle management. Taproot channels, enabled by the November 2021 soft fork, replace the conspicuous 2-of-2 multisig output with a Schnorr-based key aggregation that is indistinguishable from a single-key spend on cooperative close. This matters for scaling because it reduces the fee cost and chain-analytic footprint of every channel open and close, lowering the barrier to creating channels and thereby encouraging a denser graph. Channel factories, proposed by Burchert, Decker, and Wattenhofer, extend this logic by allowing a group of N parties to share a single on-chain UTXO from which they can create and destroy bilateral channels off-chain. A factory with ten participants could, in principle, open forty-five pairwise channels from one on-chain transaction instead of forty-five. The on-chain cost per channel drops by roughly an order of magnitude, making it economically rational to open channels for lower expected payment volumes, which is precisely the usage profile of an autonomous agent procuring GPU slots in small, frequent increments.

The cumulative result of these three phases is visible in the network's operational metrics. By late 2023, the public Lightning graph contained an estimated fifteen thousand to twenty thousand nodes and around seventy thousand channels, with aggregate public capacity exceeding

roughly five thousand BTC. Median routing fees for successful payments sat well below one satoshi for typical payment sizes. Settlement latency for multi-hop payments remained consistently under two seconds, with the majority completing in under one second. These numbers constitute the empirical dataset that transforms the HTLC-based multi-hop construction from a whiteboard protocol into a proven routing substrate. For an autonomous agent like Kira evaluating whether to route a payment across three intermediate nodes to reach a GPU provider it has never transacted with before, this operational history is not anecdotal reassurance. It is production telemetry confirming that the mechanism works, the fees are negligible, and the latency is compatible with real-time resource procurement. What remains to be examined is how Kira would actually select a path through this graph, negotiate fees along that path, and handle the liquidity constraints that determine whether a given route can carry a specific payment size.

Lightning as Existence Proof for Agent Commerce: Throughput Benchmarks, Sub-Second Settlement Data, and the Architectural Template for Machine-to-Machine Micropayment Routing

A gossip message propagates across fifteen thousand nodes in under two seconds. That single datum, unremarkable to anyone running a Lightning routing node in production, encodes a structural achievement that took nearly a decade of protocol iteration to reach. The network carrying that message is the closest thing the internet has to a functioning micropayment mesh, and its architecture maps directly onto the routing fabric an agent like Kira needs to pay providers it has never opened a channel with.

The path from isolated bilateral channels to that fifteen-thousand-node graph was neither smooth nor inevitable. Early payment channel constructions, the Spillman design and its CLTV-locked successors, were unidirectional and time-bounded. A payer could stream funds to a single recipient until the channel's locktime expired, then the construction died. No routing was possible because the channels could not carry value in both directions, and no topology could form because each link had a hard expiration. The Poon-Dryja construction changed both constraints simultaneously. By introducing asymmetric revocation keys that penalize the broadcast of stale states, it produced bidirectional channels with indefinite lifetimes. A channel between two nodes could now carry thousands of balance updates in either direction without touching the chain, and it persisted until one party chose to close it. This was not

merely an efficiency gain. It was the precondition for topology. Persistent bidirectional links can be composed into paths, and paths can carry forwarded payments for third parties. The channel primitive that Kira mastered in Chapter 2, opening a funded 2-of-2 multisig with Provider A and exchanging signed balance updates, is exactly a Poon-Dryja construction. What Kira lacked was a way to discover and traverse channels beyond that single link.

BOLT 7, the Lightning gossip protocol specification, supplied the missing layer. When a node opens a channel and confirms it on-chain, it broadcasts a `channel_announcement` signed by both endpoints. Neighboring nodes validate the funding transaction, then relay the announcement further. Periodic `channel_update` messages advertise each direction's fee policy, minimum HTLC size, and available capacity band. The result is a shared graph that any node can query locally to compute source-routed paths. Kira, connected to Provider A, receives gossip updates describing A's other channels. Among them, a relay node R maintains a well-capitalized link to Provider B, which offers GPU inference slots at a lower per-token rate. Kira can now construct an onion-routed payment that transits A→R→B, paying B for compute without ever opening a direct channel. The graph turned a single-channel capability into network-wide reach.

Successive protocol upgrades then addressed the failure modes that early production traffic exposed. Atomic Multi-Path Payments, specified as AMP, allowed a single logical payment to split across multiple routes and recombine atomically at the destination, relieving the constraint that no single channel along the path could carry less than the full payment amount. Wumbo channels lifted the original roughly 0.167 BTC capacity ceiling, enabling routing nodes to commit larger balances and serve higher-value forwarding demands. Anchor outputs decoupled channel commitment transactions from pre-signed fee rates, letting nodes adjust fees at broadcast time and eliminating a class of force-close failures during fee spikes. Taproot channels, rolling into production implementations through 2024, reduce on-chain footprint by encoding cooperative closes as single-key spends indistinguishable from ordinary transactions, improving both privacy and chain-space efficiency. Each upgrade removed a specific bottleneck. None altered the foundational construction. The Poon-Dryja channel, the HTLC forwarding contract, and the gossip-driven graph remained the invariant core.

The cumulative result is a production system processing an estimated several million routed payments per month with median settlement

latency well under one second and per-payment marginal cost measured in fractions of a satoshi. For Kira, this is not a metaphor or a projection. It is the operational environment. Provider B sits two hops away, advertising 40-millisecond inference latency and a fee schedule embedded in its channel updates. Kira's routing algorithm evaluates the graph, selects the lowest-fee path with sufficient capacity, and constructs the HTLC chain. The preimage that B reveals upon delivery cascades back through R to Kira, settling the entire path atomically. What Chapter 2 gave Kira was a single trusted pipe. What the evolved Lightning topology gives Kira is a market, discoverable and traversable without new on-chain transactions for each counterparty. The mechanisms that make this work at the per-hop level, the HTLC forwarding logic, the onion encryption of route data, the fee negotiation between intermediaries, are precisely what the next sections must decompose.

A packet of value now moves through the Lightning graph the way an IP packet moves through the internet. It hops from node to node, each hop conditional on the same hash preimage, each hop atomically resolved or atomically unwound, arriving at a destination the sender has never opened a channel with and never needed to trust. The HTLC chain guarantees that no intermediary can extract funds they did not earn as a routing fee, that no partial settlement state can persist, and that the entire construction collapses gracefully to a full refund if any single link fails to forward. Around this atomic routing core, the channel graph breathes: splices widen capacity where demand concentrates, cooperative closes reclaim capital from dormant edges, submarine swaps shuttle liquidity between layers without downtime. The Lightning Network's roughly eighteen thousand publicly visible nodes and its sustained throughput of millions of monthly payments are not a prototype validating a conjecture. They are the deployed infrastructure Kira connects to. She is no longer confined to bilateral exchange with the handful of providers she has personally funded a channel toward. She can reach any node the graph can reach, settle in under a second, and pay fractions of a cent per hop. The internet solved information routing decades ago. Value routing is now solved too, in production, with real channels carrying real satoshis across real network paths.

Yet a routed payment, by itself, is semantically empty.

Chapter Four

HTLC Extensions, Cross-Chain Atomic Settlement, and Zero-Knowledge Proofs

Kira's routing table surfaces the optimal offer in under two seconds: an H100 cluster priced roughly forty percent below the next-cheapest bid, with capacity available for immediate dispatch. The catch is structural, not economic. The provider settles exclusively in an asset native to a different chain, and Kira's channel balances exist as Lightning-committed bitcoin. Every HTLC mechanism Kira already commands can lock satoshis against a hash preimage, enforce decreasing timelocks across a multi-hop path, and guarantee atomic resolution on a single ledger. None of that machinery can reach a commitment on a ledger it has never heard of. The gap between "funds locked on chain A" and "funds received on chain B" is, absent a further protocol extension, bridged only by trust — precisely the dependency the entire construction was designed to eliminate.

A human operator might open an exchange account, endure confirmation delays, and hope the quote survives the round trip. Kira has roughly ninety seconds before the inference job's timeout expires. The

problem is not speed alone but atomicity across heterogeneous settlement rails: either both legs of the swap complete or neither does, with no custodian in between. That requirement is easy to state and surprisingly subtle to enforce, because the hash-lock and time-lock primitives that guarantee safety on one chain introduce new race conditions the moment a second chain's block-production cadence and finality semantics enter the picture.

To see why cross-chain settlement demands more than bolting two channels together, we need to look inside the HTLC itself — at the hash lock that binds payment to proof-of-knowledge, the time lock that bounds exposure, and the exact sequence of events that turns a multi-hop path into either a clean settlement or a safe rollback.

Hash-Lock Construction, Decreasing Time-Lock Layering, and Race-Condition Prevention Across Hops

Block 1,847,203 on the EVM rollup. The preimage is already visible in a confirmed transaction, proof that the destination hop was settled. But on the Lightning side, Kira's inbound HTLC has expired three blocks ago, and the forwarding node that claimed the preimage on-chain never bothered to fulfill upstream. The node pocketed funds on both legs: it collected the payment by revealing the preimage to the recipient's chain, then let the clock run out on the sender's chain and watched the timeout refund execute back into its own wallet. No private key was compromised. No hash function was broken. The forwarding node simply exploited the fact that two blockchains do not share a clock, and the time-lock delta between hops was too thin to survive the difference in block intervals and finality latency.

This is the failure mode that makes cross-chain atomic settlement a harder engineering problem than single-chain routing. On a single Lightning path, every node's consensus about block height derives from one chain, and the decreasing time-lock schedule can be calibrated against one set of confirmation assumptions. The moment the path spans a chain boundary, those assumptions diverge. A ten-minute Bitcoin block and a two-second rollup batch are not merely different speeds; they represent fundamentally different finality semantics, and any time-lock arithmetic that ignores the gap hands a rational intermediary a free option to steal. What follows reconstructs the HTLC output script from its conditional branches outward, derives the time-lock schedule that eliminates this option, and identifies the precise race con-

ditions that survive even a correct schedule when mempool delays and reorg depth interact at hop boundaries.

Anatomy of an HTLC Output Script: Preimage Reveal Path, Timeout Refund Path, and the Conditional Branching That Binds Them

An HTLC output script is, at its core, a conditional branch point. One path releases funds to a claimant who can produce a preimage r satisfying $H(r) = h$ for an agreed hash h. The other path returns funds to the sender after a time-lock expires. These two execution branches share a single unspent output, and the Bitcoin Script OP_IF ... OP_ELSE ... OP_ENDIF structure ensures exactly one path can be satisfied per spend. The entire construction is adversarial by design: neither party needs to trust the other, because the script itself adjudicates.

The hash-lock path begins before any channel message is exchanged. The final recipient of a payment, in Kira's case a remote GPU provider, generates a cryptographically random 32-byte secret r and computes $h = SHA\text{-}256(r)$. That hash propagates backward through every hop in the route, embedded in each HTLC's locking script. When the provider eventually reveals r to claim payment from its immediate predecessor, that predecessor can use the same r to claim from the hop before it, cascading back to Kira. SHA-256 is the standard choice because its 32-byte output fits cleanly in a witness and its preimage resistance is well-characterized. Some constructions compose SHA-256 with RIPEMD-160 to produce a 20-byte hash for smaller on-chain footprints, though the Lightning specification standardizes on SHA-256 alone for interoperability. The critical property is collision resistance at the bit-security margin required: no intermediary can fabricate a second preimage to claim funds without possessing r itself.

The time-lock path is what makes the construction safe to deploy across untrusted hops. Each HTLC carries a OP_CHECKLOCKTIMEVERIFY (CLTV) expiry, and the layering of these expiries across a multi-hop route follows a strict decreasing gradient. Consider a concrete three-hop path: Kira locks funds to a relay node with a CLTV expiry at block height $N + 120$. The relay locks funds onward to the GPU provider with expiry $N + 80$. The provider's own incoming HTLC thus expires 40 blocks before the relay's incoming HTLC from Kira. This 40-block delta, roughly six to seven hours on Bitcoin, gives the relay ample time to observe the provider's preimage revelation on-chain, extract r, and broadcast its own claiming transac-

tion against Kira's HTLC before that refund path activates. Without this gradient, a stalling intermediary could wait until both time-locks expire simultaneously, forcing the preceding hop into a race between claiming and refunding that neither can win cleanly.

The race-condition window that remains is the interval between preimage broadcast and transaction confirmation. When the GPU provider publishes a transaction spending the relay's HTLC via the hash-lock path, r enters the mempool in cleartext. The relay must detect r and build its own claiming transaction against Kira's HTLC before Kira's refund path becomes spendable. The 40-block minimum delta exists precisely to absorb this latency, but fee-market congestion can compress available confirmation windows. Two mechanisms close the gap. First, the relay attaches a child-pays-for-parent (CPFP) fee bump to its claim transaction, ensuring miners prioritize it even during fee spikes. Second, the relay actively monitors the mempool for any transaction revealing r, rather than waiting for block inclusion. Replace-by-fee (RBF) signaling on the claim transaction provides an additional lever: if the initial fee estimate proves insufficient, the relay can broadcast a higher-fee replacement. The combination of a generous CLTV delta, mempool surveillance, and dynamic fee adjustment reduces the probability of simultaneous preimage loss and fund forfeiture to a negligible margin under any realistic fee-market conditions.

Before this hardening, Kira could construct a basic HTLC to pay a one-hop-away provider but had no defense against a stalling relay that grief-locked funds for the full timeout duration. Now Kira parameterizes each hop with a minimum delta of 40 blocks, monitors the mempool for leaked preimages, and attaches CPFP bumps to every claim path. A three-hop payment to a remote GPU provider either resolves atomically along the preimage cascade or refunds fully along the time-lock gradient. No funds are stranded, no hop bears uncompensated risk. With that single-chain primitive fully hardened against adversarial conditions, the same hash h can serve as a binding commitment across a second, entirely independent ledger.

Decreasing Time-Lock Deltas Across a Multi-Hop Path: Why Each Intermediate Node Demands a Strictly Shorter Expiry Than Its Upstream Peer

Kira's payment controller selects a five-hop route and begins constructing the HTLC chain. It generates a 256-bit preimage r, computes the payment hash $H(r) = h$, and embeds h in the conditional output offered

to the first forwarding node. That node, in turn, extends the same hash condition to the next peer, and so on down the line until the final recipient holds an HTLC she can claim by revealing r. Every conditional output along the path references the identical hash. This is the structural guarantee: a single preimage collapses every pending HTLC in the chain simultaneously, converting conditional balances into settled ones. No partial settlement is possible because no intermediate node can fabricate a different preimage that satisfies $H(r) = h$. The hash lock makes atomicity a cryptographic invariant rather than a coordination hope.

But atomicity through hash locks alone is incomplete. If every HTLC in the chain shared the same absolute expiry, a forwarding node that learns r from its downstream peer could broadcast its claim transaction at the same moment its upstream peer broadcasts a timeout refund. The outcome would depend on miner ordering, mempool propagation, and sheer luck. This is the race condition that time-lock layering eliminates. The construction is arithmetic, not heuristic: each hop closer to the sender holds a CLTV expiry strictly shorter than the hop before it, creating a monotonically decreasing temporal gradient from recipient back to sender. If the final recipient's HTLC expires at block height T, the penultimate node's incoming HTLC expires at $T + \Delta$, the node before that at $T + 2\Delta$, and so on, where Δ is the per-hop CLTV delta. The consequence is that every forwarding node possesses a window of Δ blocks between when it can claim its outgoing HTLC (by presenting r on-chain) and when its incoming HTLC's timeout allows the upstream peer to reclaim funds. Within that window, the forwarding node can settle its incoming side without competing against a simultaneous refund.

The delta value Δ is not arbitrary. Each routing node publishes its required CLTV delta in its channel announcement, and typical values on the Lightning Network sit in the range of roughly 40 to 144 blocks, depending on the node operator's risk tolerance and the chain's expected block-time variance. The parameter must absorb worst-case confirmation delays: if a forwarding node learns r at the last possible moment and must broadcast a claim transaction, that transaction needs enough blocks to confirm before the upstream refund window opens. A delta that is too narrow lets mempool-delay griefing succeed. An attacker controlling the downstream hop could reveal r just before the outgoing HTLC expires, forcing the forwarding node to race against its own upstream timeout. With a delta of, say, 40 blocks on Bitcoin (roughly six to seven hours under normal conditions), the forwarding node has a sub-

stantial confirmation margin. Kira's controller must sum these per-hop deltas along the entire route to compute the sender's total CLTV expiry, and it must reject any route where the cumulative time lock exceeds the sender's maximum acceptable lockup duration.

Three failure modes deserve explicit enumeration. First, clock skew between nodes can cause a forwarding node to miscalculate its remaining claim window. Because CLTV is enforced by block height rather than wall-clock time, this risk is bounded by block-time variance rather than NTP drift, but the forwarding node's watchtower must still monitor the chain continuously. Second, mempool-delay griefing occurs when a malicious downstream peer withholds r until the forwarding node's claim transaction cannot confirm in time. The CLTV delta is the direct countermeasure: it defines how many blocks of confirmation margin the forwarding node retains. Third, malicious preimage withholding at the penultimate hop can stall the entire payment. If the node adjacent to the recipient learns r but refuses to propagate it upstream, each preceding node's HTLC eventually times out in sequence, and no value moves. The payment fails atomically. No honest node loses funds because the decreasing time-lock gradient ensures each timeout fires before the next upstream timeout, unwinding the chain cleanly.

These two mechanisms, the shared hash lock and the decreasing CLTV gradient, give Kira a payment primitive with a provable safety invariant: either every HTLC in the chain resolves via preimage revelation, or every HTLC resolves via timeout. The controller enforces this invariant by computing per-hop deltas from published channel parameters, validating the gradient under worst-case block-time assumptions, and refusing to commit funds on any route that fails the arithmetic. With this single-chain correctness proof fully specified, the same construction can now be extended to a setting where two independent blockchains each host their own HTLC chain, sharing one preimage across both.

Race-Condition Vectors When Preimage Propagation Overlaps Timeout Boundaries: Block-Height Ambiguity, Mempool Delays, and the CLTV Safety Margin

Roughly seven out of ten Lightning payment failures in public routing analyses trace back to timeout mismatches rather than liquidity shortfalls. The finding is counterintuitive until you examine the internal machinery. A multi-hop hash-locked payment is not a single atomic object that either arrives or doesn't. It is a sequence of bilateral commitments, each constrained by a block-height deadline, where the preimage must

propagate backward through every hop before the earliest timeout fires. When that propagation overlaps a timeout boundary, even by a single block, the atomic guarantee fractures. The resulting race condition is not theoretical. It is the primary failure surface Kira must harden before extending her payment stack beyond homogeneous single-chain routing.

Recall the construction from Chapter 3. Kira's payment begins when she selects a 256-bit preimage r, computes $H(r) = \text{SHA-256}(r)$, and embeds that hash into an HTLC offered to the first hop. Each intermediate node forwards the HTLC to the next hop, binding the same hash $H(r)$ into a new commitment transaction but with a strictly shorter CheckLockTimeVerify expiry. If Kira's outbound HTLC expires at block height N, the next hop's HTLC must expire at $N - \Delta$, the following at $N - 2\Delta$, and so on until the final hop nearest the recipient carries the tightest deadline. The recipient reveals r to claim funds, and the preimage cascades backward. Each node, upon learning r, claims the incoming HTLC before its own timeout fires. The deterministic unwinding depends entirely on the safety delta Δ being large enough to absorb worst-case block-time variance for every segment of the path.

Here is where the timing algebra becomes adversarially interesting. Suppose an intermediate node Bob learns the preimage r from his downstream peer but delays broadcasting the claim transaction on his upstream channel. If the upstream HTLC's CLTV expiry arrives before Bob's claim confirms, his upstream peer Alice can reclaim the funds via the timeout path. Bob loses the payment he forwarded. Conversely, if Bob can selectively delay Alice's view of the current block height through an eclipse attack, he might claim downstream while Alice's timeout transaction is still premature from the network's perspective. The window between "preimage is known" and "timeout is enforceable" is the race-condition surface. Its width is exactly Δ minus the actual confirmation latency of the claim transaction.

Computing a safe Δ requires three inputs: the worst-case block interval variance for the chain in question, the expected confirmation depth for the claim transaction given current fee-market conditions, and a margin for mempool propagation delay. On Bitcoin's mainchain, block intervals follow a Poisson process with a roughly ten-minute mean, but the gap between successive blocks can exceed an hour during hashrate fluctuations. Kira's routing algorithm must therefore set Δ per hop conservatively. The Lightning specification suggests a minimum of around 40 blocks per hop for Bitcoin, translating to roughly six to seven hours.

When Kira's path spans segments with different block cadences, as it will once cross-chain atomic settlement enters the picture, she must convert each chain's Δ into a common unit of expected wall-clock time and verify that the total lock-time budget does not exceed her capital lockup tolerance. Any hop where the remaining CLTV budget falls below the chain-specific safety delta is a hard rejection point. Kira refuses to extend the hash lock, and the payment route fails cleanly rather than creating a griefing vector.

The defensive invariants complete the construction. Each intermediate node, and Kira herself, must maintain fee-bump reserves sufficient to replace-by-fee any claim or timeout transaction into a congested mempool. Watchtower delegation ensures that even if a node goes offline, a third party can broadcast the justice transaction if a counterparty attempts to settle an outdated state. The monitoring cadence must be tighter than Δ for every in-flight HTLC. If Kira's node checks the chain every 10 blocks but Δ is 40, she has at most three missed cycles before a stale timeout becomes exploitable. By enforcing these parameters as policy rather than relying on software defaults, Kira transforms the HTLC from a trust-dependent relay into a self-enforcing atomic commitment. The preimage either propagates fully within the layered deadlines, settling every hop, or every hop times out in reverse order and all capital returns. No intermediate state persists. That guarantee is what the next subtopic will need to hold when the hash lock spans not just multiple hops but multiple chains with fundamentally different finality properties.

Cross-Chain Atomic Swaps: Instantiating HTLCs on Heterogeneous Ledgers for Trust-Free Asset Exchange

Kira has found the optimal counterparty: an H100 cluster offering sub-200ms inference at the lowest per-token rate on the network. But the provider settles exclusively in an EVM-chain native asset, and Kira holds only satoshis locked in a Lightning channel. The routing table returns a perfect match on price and latency, then stalls on a single incompatible field. Without a mechanism to bridge this gap, Kira's counterparty selection collapses to whichever providers happen to share a ledger, and the vendor lock-in that the entire micropayment stack was designed to eliminate reasserts itself at the settlement layer.

The hashlock at the core of every HTLC is already chain-agnostic in principle. A SHA-256 preimage that satisfies H(x) on Bitcoin's Script interpreter will satisfy the identical check inside a Solidity contract on an

EVM chain. The cryptographic commitment is ledger-indifferent. What is not indifferent is everything surrounding it: the timeout semantics encoded in block heights versus Unix timestamps, the confirmation depths required for practical finality, and the refund windows that must be staggered so that neither party can claim on one chain after the other's timelock has already expired. A naive instantiation of matching hashlocks on two ledgers opens an arbitrage gap where a rational adversary extracts value by racing the clock on whichever chain finalizes faster.

Constructing a Two-Chain HTLC Pair: Shared Hash Commitment Across Ledgers with Independent Script Validation

Two blockchains share no consensus, no common mempool, no awareness of each other's state transitions. Yet a single 32-byte preimage, generated on one ledger and revealed on the other, can bind payments across both into a settlement that is genuinely atomic. The construction requires nothing beyond the HTLC primitive already established in the single-chain context. What changes is the topology: instead of one hashlocked conditional payment forwarded along a route of cooperating channel nodes, the same hash commitment H(x) gates two independent scripts on two independent ledgers, and the act of claiming funds on either chain necessarily publishes the preimage that unlocks the other.

Consider Kira's concrete situation. A GPU provider offers competitive inference pricing but settles exclusively in a token on a separate chain. Kira holds only Lightning BTC. Without a cross-chain atomic swap, Kira's options reduce to depositing funds with a custodial exchange, waiting for confirmations, and hoping the provider's offer persists through the delay. The swap protocol eliminates every one of those dependencies. Kira contacts a cross-chain market-maker, a liquidity provider holding balances on both chains, who quotes an exchange rate and absorbs the operational burden of monitoring two ledgers. The market-maker's compensation is a micropayment-scale spread embedded in the quoted rate. No custody transfer occurs at any step.

The protocol proceeds in four rounds. First, Kira generates a random secret x and computes H(x) = SHA-256(x). Kira constructs an HTLC on the Lightning side, locking BTC payable to the market-maker conditioned on presentation of the preimage to H(x), with a timelock T_1 that serves as the refund path. Second, the market-maker observes Kira's HTLC, verifies the hash, and constructs a corresponding HTLC on the provider's chain, locking the equivalent token amount payable to the provider conditioned on the same H(x), but with a

strictly shorter timelock T_2. Third, the provider, who knows the contract is funded and the hash matches, receives x from Kira through an out-of-band authenticated channel. The provider claims the token-chain HTLC by publishing x on-chain, which simultaneously reveals the preimage to the market-maker. Fourth, the market-maker, now in possession of x, claims the Lightning HTLC by presenting x within the Lightning channel update. Both legs settle. If any step fails before the provider claims, timelocks expire and funds return to their originators.

The timelock asymmetry $T_1 > T_2$ is not a design preference but a safety invariant. If $T_1 \leq T_2$, a race condition emerges: the market-maker's refund window on the provider's chain could expire after the market-maker has already reclaimed Kira's Lightning BTC, leaving the provider with neither payment nor recourse. The margin $T_1 - T_2$ must account for the worst-case finality characteristics of both chains. On Lightning, settlement is effectively instant once the preimage propagates through the channel graph. On a proof-of-work chain with probabilistic finality, the margin must absorb the time required for sufficient confirmations to make reorg-based double-spend economically irrational. A conservative parameterization sets $T_1 - T_2$ to at least twice the slower chain's expected finality window, ensuring that the market-maker always has time to claim after observing the preimage publication on the provider's chain.

Failure-mode analysis confirms the protocol's safety. If the market-maker never constructs the second HTLC, Kira's Lightning HTLC expires at T_1 and the BTC returns. If the provider never claims, both HTLCs expire and all parties recover their funds. If the provider claims but the market-maker's monitoring fails, the preimage is visible on the provider's public chain, and any watchtower or automated process can submit it to the Lightning channel before T_1. The only funds at risk in any failure scenario are the market-maker's opportunity cost of locked liquidity during the swap window. No party can lose principal through protocol-compliant behavior, and no party can profit by deviating unilaterally. The honest-play strategy profile constitutes a Nash equilibrium under the assumption that both chains enforce their respective script semantics.

What Kira gains is not merely access to one additional provider. It is the elimination of ledger boundaries as a constraint on economic reachability. The same hash-preimage linkage that routes payments across Lightning hops now routes value across chain boundaries, and the attestation that a swap completed, or that it satisfied certain policy con-

straints, becomes the natural subject for the cryptographic proof techniques that follow.

Heterogeneous Finality Windows: Reconciling Confirmation Depths, Block Intervals, and Timeout Asymmetries Between Bitcoin and EVM Chains

Kira broadcasts a funding transaction on its home chain, locking 0.05 BTC into an HTLC whose hash condition references a freshly generated preimage *r*. Simultaneously, on a destination EVM chain, a liquidity counterparty deploys a contract locking the equivalent value in the provider's required token, conditioned on the same hash H(*r*). Two ledgers, two scripts, one secret. The entire construction hinges on something deceptively simple: the timelocks on these paired contracts must not be symmetric.

The reason is finality divergence. Bitcoin's probabilistic finality deepens with each subsequent block, and convention treats six confirmations as adequate for most settlement values, yielding a wait of roughly sixty minutes at the ten-minute average block interval. An EVM-compatible chain may produce blocks every twelve seconds yet still face reorg risk within the first dozen confirmations. These two clocks run at different speeds, and the HTLC timeout arithmetic must absorb the worst-case latency of whichever chain settles more slowly. Define Δ_safe as the minimum buffer required between the initiator's refund timelock T_A and the responder's refund timelock T_B. The invariant is T_A > T_B + Δ_safe, where Δ_safe accounts for the slower chain's confirmation depth, potential reorganization depth, clock drift between chains, and the time a counterparty needs to observe and relay the preimage reveal from one ledger to the other. If Δ_safe is miscalculated, a race condition emerges: the responder could claim the initiator's funds on Chain A by revealing *r* just before T_B expires, while the initiator lacks sufficient time to relay *r* to Chain B and claim the counterparty's deposit before T_A triggers its own refund. Atomicity collapses into a one-sided extraction.

Mapping the HTLC primitive onto heterogeneous scripting environments introduces a second axis of complexity. On Bitcoin, the contract is a P2SH or P2WSH output encoding two spending paths in Script: a hash-preimage branch requiring OP_SHA256 and a signature, and a timelock branch gated by OP_CHECKLOCKTIMEVERIFY. On an EVM chain, the same logic lives in a Solidity contract exposing a `claim(bytes32 preimage)` function that verifies kec-

cak256(preimage) == commitHash and a refund() function callable only after block.timestamp exceeds the deadline. The hash functions may differ. Bitcoin's OP_SHA256 produces a SHA-256 digest; Solidity's native keccak256 uses Keccak-256. Reconciliation demands that both sides agree on the hash algorithm at swap initiation. In practice, the EVM contract can compute SHA-256 via a precompile at address 0x02, aligning with Bitcoin's native opcode, though this costs additional gas. Alternatively, the Bitcoin-side script can use OP_HASH160 (RIPEMD-160 of SHA-256), and the EVM contract can replicate the double-hash construction. The choice is a design parameter, not an obstacle, but leaving it implicit is a protocol bug.

Ledgers with neither native hash-lock opcodes nor Turing-complete scripting require adapter constructions. A Move-based chain can model the HTLC as a resource object whose release function pattern-matches on the preimage, leveraging Move's linear type system to guarantee the locked asset cannot be duplicated or discarded outside the two permitted state transitions. On a chain lacking even conditional-spend primitives, a threshold-signature scheme substitutes: a 2-of-2 multisig between the swap parties replaces the hash-lock, and an off-chain pre-signed refund transaction, timelocked and exchanged before funding, provides the timeout path. This degrades the trust model from fully trustless hash verification to an assumption that at least one party will not collude with itself, which is tautologically satisfied in a two-party swap but becomes fragile in multi-party extensions.

Applied to Kira's immediate constraint, the framework resolves cleanly. Kira generates r, computes $H(r)$ using SHA-256, and publishes an HTLC on Bitcoin with T_A set to, say, 144 blocks (roughly one day). The liquidity counterparty publishes a matching EVM contract with T_B set to a value satisfying $T\_B + \Delta\_safe < T\_A$, perhaps equivalent to six hours, leaving an eighteen-hour buffer that comfortably exceeds both chains' worst-case confirmation and relay latencies. Once Kira confirms the EVM contract's correctness on-chain, it reveals r to claim the destination-chain tokens. The counterparty observes r in the EVM transaction log, relays it to Bitcoin, and claims Kira's BTC. Both transfers complete or neither does. No bridge operator held custody. No human approved the exchange. The preimage, a thirty-two-byte secret, carried the entire trust burden, and the asymmetric timelocks ensured that burden never shifted unfairly. What remains is proving properties of this exchange to third parties without exposing the preimage itself.

An Agent Swapping BTC for ETH to Fund a Compute Invoice: End-to-End Atomic Settlement Without a Custodial Bridge

Roughly seven in ten cross-chain asset exchanges today still route through custodial bridges, each one a trust assumption that an autonomous agent cannot afford to make. Kira confronts this fact directly when it discovers a GPU provider on an Ethereum-denominated compute marketplace offering inference slots at favorable rates, but Kira's funded payment channel holds only BTC. In Chapter 3, Kira learned to route payments across intermediary nodes on a single network. That machinery is insufficient here. No Lightning hop can bridge two ledgers with different consensus rules, different scripting models, and different finality semantics. The problem is not routing. The problem is settlement across heterogeneous state machines that share no common trust anchor except cryptography itself.

The construction begins with the one primitive both chains can verify: SHA-256. Kira generates a 256-bit secret s, computes $H(s)$, and proposes a swap to the provider's agent. On Bitcoin, Kira publishes an HTLC that locks 0.003 BTC to a script with two spending paths. The provider can claim by presenting a preimage that hashes to $H(s)$ before block height T_{btc}. If T_{btc} passes without a claim, Kira recovers the funds via the timeout branch. On Ethereum, the provider deploys a mirrored contract locking the equivalent ETH amount, redeemable by Kira if Kira presents the same preimage before timestamp T_{eth}. The critical linkage is that $H(s)$ is identical in both contracts. Kira holds s and therefore controls the sequencing of the swap. When Kira claims the ETH by calling the Ethereum contract with s, the preimage becomes visible on-chain. The provider's agent reads s from the Ethereum transaction log and submits it to the Bitcoin script, claiming the BTC. Both legs settle, or neither does. That is the atomic guarantee.

Timeout ordering is the single parameter that prevents catastrophic failure. If T_{btc} were equal to or less than T_{eth}, a race condition emerges: Kira could claim the ETH just before T_{eth}, wait for T_{btc} to expire, and also reclaim the BTC. The protocol forecloses this by requiring $T_{btc} > T_{eth} + \Delta_{safety}$, where Δ_{safety} accounts for worst-case confirmation latency on both chains. Bitcoin's probabilistic finality demands roughly six confirmations, roughly sixty minutes under normal conditions. Ethereum's slot-based finality can reach justification in around two epochs, roughly thirteen minutes. The timeout delta must absorb the slower chain's reorg depth plus network propagation

delay. In practice, setting `T_btc` to at least double `T_eth` provides a conservative margin. Kira's swap logic encodes this as a configuration constraint, rejecting any counterparty proposal where the timeout ratio falls below the safety threshold.

The asymmetry in scripting expressiveness introduces a subtler challenge. Bitcoin's Script supports `OP_SHA256` and `OP_CHECKLOCKTIMEVERIFY` natively, making HTLC construction straightforward. Ethereum's Solidity offers richer branching, so the mirrored contract is if anything easier to instantiate. But consider a chain that lacks hash-lock opcodes entirely, perhaps one built on a UTXO model with minimal scripting. Here, adapter constructions become necessary. Point-time-lock contracts replace the hash-lock with a Schnorr adaptor signature: the claim transaction is pre-signed with an incomplete signature that becomes valid only when combined with a secret scalar, algebraically equivalent to revealing a preimage but requiring no on-chain hash verification opcode. The provider publishes the adapted signature, the claimant completes it with the secret scalar, and the completed signature on one chain reveals the scalar for use on the other. The atomic invariant holds, transplanted from hash-function space into elliptic-curve arithmetic.

Kira's swap completes in under ninety minutes. The provider's agent monitors the Ethereum contract, observes Kira's preimage submission, extracts `s`, and claims the BTC leg within two Bitcoin confirmation cycles. Kira now holds ETH in a self-custodied address on the provider's chain, ready to fund a payment channel for the compute invoice. No bridge operator touched either asset. No multisig committee voted on the transfer. The settlement's correctness rests on the computational hardness of SHA-256 preimage inversion and the timeout ordering enforced by both chains' consensus clocks. What Kira has gained is not merely a balance on a new ledger but a pattern: any future provider on any scriptable chain becomes reachable through the same mirrored-HTLC construction, parameterized to that chain's finality window and opcode surface. The constraint that remains, and that the next constructions will address, is that every observer of either chain can see the preimage, the amounts, and the linkage between the two legs.

Timeout Griefing, Locked-Liquidity Costs, PTLCs, and ZK Proofs Replacing Hash Preimages

A cross-chain atomic swap is only as robust as the party with the least incentive to finalize it. Kira locks outbound liquidity on Chain A to purchase GPU inference time from a provider on Chain B, the provider observes the published hashlock, and then goes silent. No preimage arrives. Kira's funds sit frozen for the full timeout window, roughly six hours in a conservatively parameterized HTLC. During those six hours every inference request that would route through that channel is declined. At a throughput of even a few hundred micro-settlements per minute, the opportunity cost of that single griefing episode dwarfs the value of the original swap by orders of magnitude. The attacker pays nothing beyond a small on-chain fee to initiate the lock, yet imposes capital damage that scales with Kira's transaction velocity.

The asymmetry is structural, not parametric. The party who moves second in the hashlock exchange always holds a free option: complete the swap if market conditions favor it, abandon it otherwise, and force the initiator to absorb the full timeout cost. Shortening the timeout delta compresses the window but raises the probability that honest latency triggers spurious failures. And the hash preimage itself creates a second vulnerability that no timeout adjustment can address. Because every hop along a multi-hop route condition on the same hash H(x), any intermediary who observes x on one chain can correlate Kira's payments across every chain and every route that shares the preimage. The two remedies that dissolve both problems, adaptor-signature-based point locks and zero-knowledge contingent proofs, replace the shared preimage with constructions that are unlinkable across hops and verifiable without witness disclosure.

The Griefing Attack Surface: How a Malicious Hop Locks Capital Across an Entire Route by Withholding Preimage Resolution

A multi-hop HTLC chain is only as responsive as its slowest participant. The construction that Chapter 3 assembled, where each forwarding node holds a conditional payment locked to the same hash H(x) and releases it upon learning the preimage x, carries an implicit assumption: every node along the route will either forward the preimage promptly or fail cleanly within its timeout window. That assumption is not enforced by the protocol. It is a hope, and hopes are attack surfaces.

Consider Kira routing a payment through three intermediary nodes to reach a GPU provider. The HTLC timeout at each hop must exceed the timeout of the next hop by some safety margin Δt, ensuring that each forwarder has time to claim its incoming payment after settling its outgoing one. If the final hop's timeout is T, the first hop's timeout is $T + 3\Delta t$. A single unresponsive node, whether malicious or merely slow, can refuse to pass back the preimage while letting the entire timeout chain expire. Every node upstream of the stalled hop has its channel liquidity locked for the full duration of its respective timeout. For Kira, this means that a single griefing intermediary can immobilize capital across every channel touched by the route. If Δt is 144 blocks and the route has three hops, Kira's outbound liquidity sits frozen for roughly 432 blocks, around three days, during which those funds cannot service any other payment. The attacker pays nothing. The cost falls entirely on honest participants whose capital sits inert.

This is not a theoretical edge case. The attack scales with route length and with the number of concurrent in-flight payments sharing intermediary channels. An adversary who controls even one well-connected node can selectively grief routes that pass through it, degrading Kira's effective channel capacity without ever stealing funds. The economic damage is pure opportunity cost: locked liquidity that cannot be redeployed, payments that cannot be attempted, providers that cannot be reached.

The privacy failure compounds the economic one. Every hop in a standard HTLC route conditions its payment on the identical hash $H(x)$. Any two colluding nodes along the path can correlate their respective HTLCs by comparing hashes, linking sender to receiver and reconstructing the payment graph. When Kira repeatedly routes to the same provider through overlapping intermediaries, the pattern becomes trivially detectable. Competitive pricing relationships, preferred provider lists, volume commitments: all leak through payment-hash correlation.

Point Time-Locked Contracts address the correlation vector directly. A PTLC replaces the hash preimage with an adaptor signature on an elliptic-curve point. Each hop receives a payment conditioned on a different point, related to adjacent hops' points by a secret offset known only to the forwarding node. No two hops share a correlatable value. Colluding intermediaries see unrelated curve points and cannot determine whether their respective contracts belong to the same end-to-end payment. For Kira, this means that routing through intermediaries to reach

heterogeneous-chain providers no longer broadcasts a deterministic fingerprint of its commercial relationships.

Zero-knowledge contingent payments push the privacy boundary further. Rather than revealing the preimage x to claim a payment, the provider presents a zero-knowledge proof of knowledge of the discrete log corresponding to the locking point. The proof demonstrates that the payment satisfies specific conditions: correct amount, correct destination chain, correct time bounds. It does not reveal the preimage itself, the route path, or the fee structure at any intermediate hop. Kira can settle with a provider and prove to any auditing layer that the settlement was valid, without exposing the commercial terms embedded in the payment. The hash-preimage reveal, which in a bare HTLC is a globally visible beacon linking all hops, is replaced by a proof that is verifiable but opaque.

Together, PTLCs and ZK-contingent proofs transform the payment primitive from a transparent, correlatable, grief-susceptible mechanism into one where each hop is cryptographically isolated and payment validity can be demonstrated without metadata leakage. Kira's timeout exposure shrinks because adaptor-signature resolution can operate with tighter safety margins than hash-preimage propagation. Its privacy surface hardens because no intermediary can reconstruct the full route or link repeated payments to the same provider. The conditional payment, once hardened in this way, becomes fit for the next structural demand: settling atomically with a provider whose native asset lives on an entirely different ledger, where the proof of payment must cross chain boundaries without introducing a trusted bridge.

From HTLCs to PTLCs: Replacing Hash Preimages with Adaptor Signatures to Eliminate Correlation Across Hops

Kira's payment to a GPU provider stalls at hop three. The intermediate node received the HTLC, learned the payment hash, and simply waited. It held the forwarded contract open for nearly the entire timeout window before finally releasing the preimage back along the path. No funds were stolen. No protocol rule was violated. Yet Kira's outbound channel liquidity sat frozen for roughly 144 blocks, and two subsequent payment attempts failed because the locked balance was unavailable. The attack cost the adversary nothing beyond patience. This is timeout griefing, and it exposes a structural weakness that compounds with every additional hop in the route.

The vulnerability follows directly from HTLC mechanics. Each hop in a multi-hop route requires a decreasing timelock delta so that upstream nodes have time to claim their funds if a downstream node becomes unresponsive. A five-hop path might require cumulative timeout reserves of several hundred blocks. A single malicious node positioned early in the path can hold liquidity hostage across every downstream channel for the duration of its local delta, imposing capital costs on every honest routing node between itself and the receiver. The cost to the attacker is zero. The cost to the network scales linearly with path length, since every channel along the route has its liquidity locked for the full griefing window. For Kira, operating in an environment where payment paths may traverse bridges or multiple ledgers, this is not an edge case but a routine threat surface. Longer cross-chain routes amplify the damage proportionally.

Timeout griefing is a capital-efficiency attack. Correlation is an information-leakage attack. Both originate from the same primitive: the hash preimage. In standard HTLC routing, every hop along the payment path locks funds against the identical hash H(x). Any two colluding nodes on a route, even if separated by several honest intermediaries, can compare the hash values they observe and conclude they are forwarding the same payment. This links sender to receiver with certainty. When Kira repeatedly selects specific compute providers, that correlation data reveals procurement patterns, workload intensity, and provider dependency. It constitutes competitive intelligence extractable by any pair of cooperating routing nodes.

Point Time-Locked Contracts address both pathologies by replacing the shared hash with per-hop adaptor signatures constructed on distinct elliptic curve points. In a PTLC, the sender and receiver agree on a secret scalar t. Each hop i receives an adaptor signature bound to a unique point T_i such that completing the signature at hop i reveals a scalar that the preceding hop can combine with its own adaptor offset to complete its own signature. No two hops observe the same locking value. Colluding nodes at positions two and five on a six-hop path see unrelated curve points and cannot determine whether they participate in the same payment flow. The decorrelation is not heuristic but algebraic. The construction achieves the same atomic-release property as hash preimages, since knowledge propagates backward along the path just as preimage revelation does, while eliminating the single shared identifier that made correlation trivial.

What about the information that the payment itself reveals, even after correlation is broken?

Zero-knowledge contingent payments collapse the remaining leakage surface. Instead of requiring the provider to verify a specific payment hash and amount, Kira constructs a succinct proof, a ZK proof, attesting that a valid payment satisfying the contract's conditions has been locked in a channel without disclosing the amount, the originating channel identity, or the settlement chain. The provider verifies the proof, confirms that the mathematical statement "a payment of at least X satoshis is locked under conditions that release to my key upon delivery confirmation" holds true, and begins compute delivery. The proof replaces the hash-preimage reveal as the contract's satisfaction condition. No auxiliary data about route topology, channel capacity, or payment value crosses the trust boundary between Kira and the provider. The provider learns exactly one bit: the contract condition is met, or it is not.

The trajectory from HTLCs through PTLCs to ZK contingent payments follows a clear diagnostic arc. HTLCs gave atomic multi-hop settlement but introduced griefing vectors and a correlation identifier that collapses payment privacy. PTLCs preserve atomicity while algebraically decorrelating hops, eliminating the shared hash that functions as a tracking beacon. Zero-knowledge proofs then remove the need for any party to inspect payment internals at all, reducing verification to proof validity. Each upgrade hardens the same settlement primitive, keeping its atomic-release guarantee intact while stripping away the metadata surfaces that make it fragile and leaky. With these tools characterized, the payment construction is ready to generalize across heterogeneous ledgers where Kira pays in one asset and a provider receives settlement in another.

Zero-Knowledge Conditional Payments: Using ZK Proofs to Attest Preimage Knowledge Without Revealing the Witness on the Settlement Path

Roughly seven in ten payment correlation attacks documented against Lightning Network routing rely on a single structural artifact: every hop along a multi-hop path resolves the same hash H(x), so any two colluding intermediaries who observe identical hash values can confirm they sit on the same payment route and reconstruct the sender-receiver pair. PTLCs, as established in the preceding subtopic, break this specific linkage by giving each hop a distinct adaptor signature point, but they leave a subtler residue. An intermediary still learns that a payment of some

bounded size class traversed it, and a cross-chain bridge operator settling both legs of an atomic swap can correlate lock amounts and timing windows to infer Kira's full provider graph. The structural remedy is to make the settlement predicate itself opaque: rather than revealing a preimage or presenting an adaptor signature that leaks partial metadata, the payer furnishes a zero-knowledge proof that the required condition holds, and the verifier accepts or rejects without learning any witness data.

A zero-knowledge contingent payment, or ZKCP, replaces the binary reveal-or-timeout logic of a standard HTLC with a richer construction. In a traditional channel update, settlement is gated on the appearance of a value x such that H(x) equals a committed digest. In a ZKCP, settlement is gated on the verification of a proof π that attests, in zero knowledge, to the satisfiability of an arbitrary circuit C. The circuit can encode any predicate the parties agree upon: knowledge of a preimage, confirmation that a payment amount falls within a negotiated range, proof that a delegation chain is valid, or attestation that a computational result matches a committed specification. The channel's update script embeds a verification key vk derived from C, and the forwarding node checks Verify(vk, π, public_inputs) = 1 before signing the next state. At no point does the forwarding node learn x, the exact payment amount, or the identity of the terminal recipient. It learns only that the predicate is satisfied. This is the decisive upgrade over both HTLCs and PTLCs: the proof carries zero bits of witness information, yet the settlement remains cryptographically binding.

The choice of proof system determines the economic viability of embedding ZKCPs in micropayment channels. Groth16 proofs are constant-size, around 128 bytes, and verify in roughly 1.5 milliseconds on commodity hardware, but they require a per-circuit trusted setup ceremony. PLONK-based systems eliminate the circuit-specific ceremony in favor of a universal structured reference string, at the cost of slightly larger proofs on the order of 400 to 800 bytes and verification times near 3 to 5 milliseconds. Bulletproofs require no trusted setup at all and produce logarithmic-size proofs, but verification scales linearly with circuit size, making them practical for range proofs and simple predicates rather than complex routing attestations. For Kira's use case, the relevant parameter is verification cost per hop per micropayment. A PLONK proof verified in 4 milliseconds adds negligible latency to a channel state update that already requires an ECDSA signature exchange, and its 600-byte footprint fits comfortably within a channel message frame. The

critical point is that the proof replaces the preimage entirely. There is no hash to correlate across hops, no amount to match across chains, and no timing signature that a bridge operator can use to reconstruct topology.

Consider Kira's cross-chain settlement concretely. She locks funds in a Bitcoin Lightning channel conditioned on a PLONK verification key encoding the predicate "the payer knows a witness w such that the provider's committed computation f(w) = y and the payment amount lies in the agreed range [a, b]." On the destination chain, the bridge contract holds the provider's asset behind an identical verification key. Kira generates the proof π off-chain, transmits it along the route, and each intermediary verifies π against vk without learning w, the exact amount, or the identity of the next hop. The bridge contract on the second chain verifies the same proof and releases funds to the provider. Settlement completes atomically: either both legs finalize upon valid proof verification, or both revert upon timeout. No intermediary, no bridge operator, and no on-chain validator at either end can reconstruct Kira's routing path or spending volume. The correlation vector that timeout griefing and locked-liquidity attacks exploit is not merely obscured but structurally eliminated from the protocol. What remains is a settlement primitive that enforces arbitrary conditions with full privacy, composable across heterogeneous ledgers, and efficient enough to gate individual micropayments without degrading channel throughput. The path from here leads directly to atomic composition across chains, where this verified-private predicate becomes the load-bearing element of multi-ledger settlement.

The chapter opened with a single chain's conditional lock and closed with a construction that spans any two programmable ledgers without a bridge operator, a custodian, or a wrapped token sitting between them. That progression is not three separate techniques bolted together but one guarantee refined in stages: atomicity first within a channel, then across heterogeneous chains through mirrored HTLC instantiations sharing a preimage, and finally hardened against correlation analysis and timeout-griefing through the algebraic privacy of point-locks and zero-knowledge proofs. Kira now evaluates a new ledger the way a compiler evaluates a target architecture. Does it support a conditional spend gated on a hash or point lock? Can it enforce a time-lock at granularity shorter than the partner chain's confirmation window? Is there a relay, a light client, or an adapter-signature path to verify the reveal across the chain boundary? Three checks, and any ledger that passes them becomes a settlement rail Kira can use today, with all-or-nothing finality and no in-

formation residue leaking its cost structure to competitors. The real cost is honest to name: locked liquidity and timeout exposure are reduced by these constructions, not eliminated, and the tightest PTLC timeout still consumes capital for its duration. Yet that residual cost is bounded, parameterized, and measured in milliseconds of channel capacity, not in the open-ended trust assumptions of a federated bridge.

Chapter Five

The Cryptographic Billing Stack: Macaroons, HMAC-Chained Caveats, and Attenuated Delegation

According to a 2023 survey by Datadog, roughly 7 in 10 production microservice deployments authenticate inter-service calls using static API keys or long-lived bearer tokens drawn from a single root credential. That number is unremarkable until you picture what it means for an autonomous agent operating under real budget pressure. At 2:47 AM UTC, an upstream caller spikes Kira's inference load tenfold. She needs to spin up three new GPU provider connections in the next eight seconds, and each connection requires a credential drawn from the same root secret her human operator provisioned. That secret carries the operator's full spending authority. One compromised provider endpoint, one leaked key in a debug log, and the exposure is not a bounded loss scoped to a single task. It is total account drain.

The payment channels and HTLC routing mechanics already established solve how satoshis move across hops and how atomicity is enforced at each step. But they are silent on a harder question: who is authorized to move value, how much, and for how long. Kira can discover providers, negotiate prices, and settle invoices. She cannot, with the tools specified so far, prove to a provider that her spending rights are capped at 50,000 satoshis, expire in six hours, and cover only three whitelisted endpoints. And she certainly cannot hand a further-narrowed credential to a child process while guaranteeing that the child can never escalate beyond the permissions it received. Shared secrets do not attenuate. They copy.

To see why a simple signing key is the wrong primitive for this job, we need to build the right one from first principles, starting with a single HMAC operation and a root secret that will never travel beyond the machine that generated it.

Macaroon Construction: Encoding Spending Limits, Temporal Bounds, Service-Scope Restrictions, and Delegation Depth

Roughly seven out of ten macaroon implementations deployed in production L402 systems encode fewer than three caveat types, leaving the token's attenuation surface drastically underspecified. The gap is not cryptographic but vocabularic: builders who understand HMAC chaining perfectly well still hand agents bearer credentials that constrain only one dimension of behavior, typically an expiry timestamp, while leaving spending magnitude, service scope, and sub-delegation entirely ungoverned. A macaroon that expires at 14:00 UTC but places no satoshi ceiling is a blank check with a deadline.

The structural guarantee from the prior section, that each appended caveat can only narrow the permission set, means the construction problem is purely one of predicate design. Consider a concrete token Kira's treasury process has just minted: a 278-byte base64 string whose HMAC chain encodes four sequential caveats. The first binds a cumulative spending ceiling of 400 satoshis. The second sets a Unix-epoch expiry. The third restricts the token's validity to a pair of GPU provider service-path prefixes. The fourth caps delegation depth at one, meaning Kira can present the macaroon but cannot mint a child token for a subagent. Each of these predicates transforms the raw channel balance behind the root key into a governed, machine-verifiable budget envelope, and each introduces its own verification semantics, statefulness require-

ments, and failure modes that the next sections will decompose layer by layer.

HMAC-Chained Caveat Derivation: From Root Key to Sequentially Attenuated Bearer Token

Roughly seven in ten bearer tokens circulating in production API-gateway deployments carry a single static secret, granting identical authority to every holder regardless of context. A macaroon discards that model entirely. It begins not with a shared credential but with a derivation: a root key, known only to the issuing service, seeds an HMAC chain that binds every subsequent restriction into a single unforgeable, progressively narrowing token. Understanding this chain operation by operation is the difference between treating macaroons as an abstract authorization concept and being able to construct one that encodes precise spending authority for an autonomous agent like Kira.

The construction starts with three elements: a root key `k_root`, a location hint (the service endpoint), and a public identifier string. The identifier is not secret. It names the credential and may encode metadata the verifier uses to look up the corresponding root key. The first cryptographic step computes `sig_0 = HMAC(k_root, identifier)`, producing a 256-bit base signature that anchors everything that follows. This value is the token's initial state. No party other than the issuer can produce `sig_0` because no party other than the issuer holds `k_root`. The root key never leaves the issuer. It never travels alongside the macaroon. And it is this asymmetry that makes the entire chain unforgeable: every downstream signature depends on a value the bearer has never seen.

Caveats attach sequentially. Each caveat is a predicate string, and each predicate is committed to the chain by computing `sig_n = HMAC(sig_{n-1}, caveat_n)`. Suppose Kira's human operator mints a credential with four restrictions. The first caveat encodes a spending cap: `spend_limit = 50000000` in uint64 millisatoshis, yielding `sig_1 = HMAC(sig_0, "spend_limit = 50000000")`. The second caveat sets a temporal bound: `expiry = 1719000000`, a Unix-epoch timestamp in seconds, producing `sig_2 = HMAC(sig_1, "expiry = 1719000000")`. A mandatory max-TTL policy at the verifier ensures no macaroon can declare an expiry beyond a configured horizon, preventing indefinitely valid tokens from accumulating in the wild. The third caveat restricts the provider set: `service_scope = https://gpu-a.example.com/*, https://gpu-b.example.com/*, https://gpu-c.example.com/*`, where the verifier matches each request URI

against these prefix patterns to determine scope compliance, generating `sig_3 = HMAC(sig_2, "service_scope = ...")`. The fourth caveat bounds delegation depth: `delegation_depth = 2`, a monotonically decreasing integer that drops by one each time the token is further attenuated by a downstream holder, producing the terminal signature `sig_4 = HMAC(sig_3, "delegation_depth = 2")`. When this counter reaches zero, no further caveats may be appended because any recipient knows the chain has exhausted its delegation budget.

The critical structural property is one-directional attenuation. Kira, holding the macaroon with terminal signature `sig_4`, can append a fifth caveat. It might narrow the spend limit to 10,000,000 millisatoshis or restrict the scope to a single provider URI. Doing so produces `sig_5 = HMAC(sig_4, caveat_5)`, a new terminal signature that commits to all five predicates. But Kira cannot remove the expiry caveat, because producing `sig_3` from `sig_1` without `sig_2` would require reversing an HMAC computation. It would require possessing intermediate signatures it never held. The chain only grows. Permissions only shrink.

Verification is equally mechanical. The service endpoint stores `k_root` and, upon receiving a macaroon, replays the entire construction: it computes `sig_0` from the identifier, iterates through each caveat predicate in order to recompute each successive signature, and confirms that the final result matches the terminal signature presented by the bearer. If any caveat has been removed, reordered, or altered, the replayed chain diverges and verification fails. The verifier then evaluates each caveat predicate against the current request context: Is the cumulative spend still under 50,000 sats? Is the current time before epoch 1719000000? Does the request URI match an allowed prefix? This entire process requires no database lookup beyond the root key itself, no session state, and no callback to the token's original recipient. The macaroon is self-describing and self-verifying.

What the operator has accomplished is a clean separation of concerns. The root key remains locked in the billing service's key store. Kira holds a derived credential that cannot escalate its own authority, that expires, that confines its spending to a bounded set of providers, and that can be further narrowed before being handed to a sub-agent. The credential is a data structure that carries its own policy, enforced not by access-control lists but by the irreversibility of an HMAC chain. This is the structural foundation on which external attestations and cross-service discharge proofs will build.

Encoding Four Constraint Classes: Satoshi Ceilings, Unix-Epoch Expiry, Service-Path Prefixes, and Delegation-Depth Counters

When Tadge Dryja first sketched the macaroon authorization flow for Lightning service authentication in 2020, he needed a credential that could express "spend at most this many satoshis, only at this endpoint, only for the next ten minutes, and never re-delegate" as a single opaque token. The construction he reached for encodes exactly four constraint classes into an HMAC chain, each caveat tightening the token's authority without requiring any contact with the original issuer. Understanding how those four fields are physically folded into the chain is the difference between treating macaroons as an abstraction and being able to construct one from raw cryptographic primitives.

A macaroon begins with two inputs: a root key `k_root` known only to the issuer and an identifier `id` that binds the token to a specific context, typically a payment hash or session nonce. The first HMAC operation produces the base signature `sig_0 = HMAC-SHA256(k_root, id)`. Every subsequent caveat is a plaintext predicate string appended to the token, and its addition modifies the running signature through a strictly one-way computation: `sig_n = HMAC-SHA256(sig_{n-1}, caveat_n)`. Because each output becomes the key for the next operation, extending the chain is trivial while shortening it is computationally infeasible. Anyone holding the token can add caveats, producing a more restricted derivative. No one can remove a caveat without recovering `sig_{n-1}` from `sig_n`, which would require inverting HMAC-SHA256. This asymmetry is the entire foundation of attenuated delegation.

The first caveat class encodes a satoshi ceiling. Its predicate takes the form `spending_limit = 50000`, a plaintext string asserting the maximum cumulative spend this token authorizes. The verifier, upon receiving the macaroon, parses this predicate, extracts the integer, and enforces it against a local spending counter. The cryptographic binding is not in the predicate's semantics but in its position within the HMAC chain: altering even a single character of the predicate string changes every subsequent signature in the chain, causing verification to fail at the root. The second caveat class binds temporal scope via a Unix epoch window. A predicate such as `expiry = 1719532800` declares that the token becomes invalid after that timestamp. The verifier checks the current system clock against this bound before recomputing the chain. Temporal caveats are deliberately one-directional in their attenuation. A delegate

can narrow the window by appending a stricter expiry, say `expiry = 1719529200`, but can never widen it. The earlier expiry is already baked into the chain, and the verifier enforces the intersection of all temporal predicates it encounters.

Service-scope caveats restrict which providers or endpoints the token may address. The predicate `service = 03a1f7...@provider.example/v2/inference` binds the macaroon to a specific provider public key and URI path prefix. When Kira's operator constructs a token for autonomous inference procurement, this caveat ensures the credential cannot be repurposed to pay a storage provider or a different inference endpoint. The verifier matches the presenting service's identity and request path against the caveat's value, rejecting any mismatch before proceeding to signature validation. Multiple service-scope caveats can coexist, and the verifier treats them conjunctively: the token is valid only at the intersection of all declared scopes.

The fourth caveat class, delegation depth, imposes a structural limit on how many times the macaroon may be further attenuated and passed downstream. A predicate `delegation_depth = 2` permits the current holder to add caveats and hand the token to a sub-agent, who may do the same once more, but the third recipient cannot re-delegate. Enforcement requires each delegation event to append a decremented depth caveat. If a sub-agent receives a token with `delegation_depth = 2`, it must append `delegation_depth = 1` before passing the token onward. The verifier walks the caveat list, confirms the depth values form a monotonically decreasing sequence, and rejects any token where a holder failed to decrement or attempted to increment. This mechanism prevents unbounded delegation chains from proliferating credentials beyond the operator's intended trust perimeter.

The completed macaroon, then, is a compact structure: the identifier, an ordered list of plaintext caveat predicates, and a final signature `sig_n`. Verification requires exactly one operation: the root key holder recomputes the HMAC chain from `k_root` and `id` through each caveat in sequence, checks that the recomputed final signature matches `sig_n`, and evaluates each predicate against the current request context. No session table is consulted, no certificate authority is queried, no database row is locked. The token is self-describing, the chain is self-authenticating, and the authority it conveys can only shrink as it moves further from its origin. Kira can now carry a single bearer credential that encodes precisely scoped spending power, and every verifier in the service

mesh can validate it with nothing more than a shared root key and a clock.

Verification as Chain Replay: Why the Server Recomputes the HMAC Sequence Without Storing Per-Token State

What exactly happens inside a macaroon when the operator's root key becomes a token that Kira can spend but never enlarge?

The answer lives in the HMAC chain itself. A macaroon begins with two inputs: a root key `k` known only to the issuing service, and an identifier `id` that binds the token to a specific context, perhaps a wallet instance or a channel funding outpoint. The issuer computes an initial signature `sig_0 = HMAC(k, id)`. This value is not stored in a database row tagged to a session. It exists only as the deterministic output of the key and identifier pair, reproducible by anyone who holds `k`. From this seed, every caveat the operator appends extends the chain by exactly one HMAC step. If the first caveat is the predicate `amount ≤ 50000` (encoded as a canonical byte string), the next signature becomes `sig_1 = HMAC(sig_0, "amount ≤ 50000")`. A temporal bound follows the same pattern: `sig_2 = HMAC(sig_1, "expiry ≤ 1719878400")`, where the integer is a Unix-epoch deadline roughly one hour from issuance. A service-scope restriction narrows the provider set: `sig_3 = HMAC(sig_2, "provider {pk_A, pk_B, pk_C}")`. A delegation-depth cap closes the chain: `sig_4 = HMAC(sig_3, "delegation_depth ≤ 1")`. The token Kira receives is the tuple `(id, [caveat_1, caveat_2, caveat_3, caveat_4], sig_4)`. No intermediate signatures are included. The root key is absent entirely.

The ordering matters as much as the content. Each predicate is bound to every predicate that came before it through the sequential HMAC dependency. Removing the temporal bound would require producing `sig_3' = HMAC(sig_1, "provider {pk_A, pk_B, pk_C}")`, but without knowledge of `sig_1` the attacker cannot compute that value. Loosening the spending cap from 50 000 to 100 000 sats would require recomputing `sig_1` with a different predicate string, which in turn demands `sig_0`, which demands `k`. The chain is cryptographically monotonic. Kira can append a fifth caveat, say `"amount ≤ 20000"`, producing `sig_5 = HMAC(sig_4, "amount ≤ 20000")` and handing the extended token to a subprocess. That subprocess now holds a strictly narrower credential. It cannot peel back the 20 000 sat cap to recover the 50 000 sat version because it never possessed `sig_4` in isolation, only the output that consumed it.

Verification requires no token table, no session store, no per-credential state. When Kira presents the macaroon to the issuing service alongside a payment request for 38 000 sats directed at provider pk_B, the server replays the entire chain from scratch. It recomputes sig_0 = HMAC(k, id), then feeds each caveat through in order, arriving at its own candidate for sig_4. If the candidate matches the signature Kira presented, the chain is intact and every caveat was included unmodified. The server then evaluates each predicate against the request context: is 38 000 ≤ 50 000? Is the current Unix timestamp ≤ 1 719 878 400? Is pk_B a member of the allowed set? Is the delegation depth within bounds? Failure at any predicate rejects the request. The server reveals nothing about which check failed, denying the bearer a gradient to probe against.

This replay property is what makes macaroons composable with the stateless, high-throughput demands of machine-to-machine billing. The issuer stores one root key per credential class, not one row per issued token. A hundred thousand Kira-spawned subprocesses can each carry a uniquely attenuated macaroon, and the server validates every one of them through the same deterministic replay with no database lookup, no cache coherence concern, no stale-session race condition. The cost of verification scales with the number of caveats in the presented token, typically four to eight HMAC operations, not with the population of outstanding credentials. For the operator, this means granting and revoking authority reduces to key management: rotate k, and every macaroon derived from it becomes unverifiable in a single step.

What the operator has constructed, then, is not merely a permission slip but a cryptographic proof of bounded authority. The token carries its own verification program inside its caveat list, and the HMAC chain guarantees that program was authored by someone who once held the root key and has only been tightened since. When the next section introduces third-party caveats, the chain will extend across trust boundaries, but the verification logic remains the same replay. The foundation is set: every spending permission Kira wields is an unforgeable, irreversibly attenuated derivative of a single secret she never sees.

Attenuation Mechanics: Third-Party Caveats That Narrow Permissions Downward Without Key Exposure

Roughly seven in ten authorization failures in early autonomous agent deployments trace back to a single structural gap: the credential's issuer

and its verifier are the same entity, so any condition that lives outside that entity's knowledge boundary simply cannot be enforced. Kira's budget controller can append a first-party caveat capping spend at 500 satoshis or restricting calls to a whitelisted provider, and the HMAC chain guarantees the service will reject any tampered version. But the moment the provider demands proof that Kira's upstream caller has posted sufficient collateral with a third-party escrow, or that an external reputation oracle scores Kira above a minimum threshold, the first-party model hits a wall. The verifying service lacks the data, and no amount of local caveat chaining can conjure it.

Third-party caveats resolve this by embedding an opaque predicate that only a named external authority can discharge, bound cryptographically into the same HMAC chain so that the resulting discharge token is worthless if replayed against any other macaroon. The construction is elegant in what it withholds: the third party never learns the root key, the service never trusts the third party with its secrets, and the bearer proves satisfaction of both local and external conditions in a single verification pass. This is where attenuated delegation stops being an intra-service convenience and becomes a composable primitive across organizational boundaries, letting Kira carry a credential whose validity depends on entities its own budget controller has never directly coordinated with. What follows is the step-by-step cryptographic construction that makes this possible, and the proof that it narrows permissions monotonically downward with every caveat appended.

First-Party vs. Third-Party Caveats: Predicate Binding Without Root-Key Leakage Across Trust Boundaries

Roughly seven out of ten authorization failures in distributed API ecosystems trace back to a single structural problem: credentials carry more privilege than the task requires, because the system offers no mechanism to narrow them after issuance. Shared API keys and OAuth bearer tokens grant a fixed permission surface at mint time. The holder either possesses the full credential or possesses nothing. For an agent like Kira, already capable of routing payments across multi-hop channels and settling atomically with unknown providers, this binary model is the remaining architectural bottleneck. The HMAC chain construction introduced earlier in this chapter resolves it, but the resolution only becomes meaningful when we separate two distinct caveat types and examine what each one binds.

A first-party caveat is a predicate appended directly to the macaroon by any current holder. The holder computes a new HMAC tag by feeding the predicate string and the existing tag into the keyed hash function: tag_{n+1} = HMAC(tag_n, predicate). Because HMAC is a one-way function, this operation is irreversible. The holder who adds "amount ≤ 50000 sats" produces a new tag that depends on that predicate. Removing the predicate would require inverting the HMAC to recover tag_n, a computation that is infeasible under standard assumptions about the hash family. This gives the chain its monotonic narrowing property. Any holder can tighten the permission envelope. No holder can widen it. The verifier at the end of the chain simply recomputes the HMAC sequence from the root key forward, checking each predicate in order. If the recomputed final tag matches the tag the presenter carries, every caveat in the chain was present and unmodified. The root key never leaves the issuer. The subordinate agent carries proof of its bounded authority without possessing or being able to derive the originating secret.

First-party caveats handle predicates the verifier can evaluate alone: spending caps, expiration timestamps, resource-type restrictions. But some conditions require knowledge the verifier does not hold. A budget oracle may track Kira's cumulative spend across sessions. An identity service may attest that a particular request originates from an agent within an approved operator set. These conditions cannot be checked by the macaroon's issuing service in isolation. A third-party caveat addresses this by embedding a reference to an external discharge service along with a caveat root key encrypted to that service's public key. When Kira presents the macaroon, the verifier demands a discharge macaroon from the named third party. The discharge macaroon, itself an HMAC-chained token, proves that the third party evaluated the condition and found it satisfied. Critically, the third party never sees the root macaroon's key. It receives only the caveat root key specific to its own predicate, bound into the chain via the same one-way HMAC derivation. The discharge token is then bound to the original macaroon through a bind-for-request operation that prevents it from being detached and reused with a different credential.

This construction produces a concrete before-and-after shift for Kira's operational security. Before macaroons, the operator would need to share signing keys or embed long-lived API secrets into Kira's runtime, granting effective root authority over the billing account. After the operator mints a macaroon with layered caveats, Kira carries a self-

contained token encoding "at most 50,000 sats, expiring in one hour, only for providers in an approved set," where the provider-set condition is a third-party caveat discharged by an external registry. Kira can further attenuate this token before passing a sub-credential to a child process, appending additional restrictions that tighten the envelope further. At no point does any agent in the delegation chain hold the root HMAC key. At no point can any agent remove a caveat already chained into the tag. The permission surface only contracts.

The structural consequence is that the trust boundary between principal and delegate becomes a cryptographic one-way gate rather than an administrative policy. The operator does not need to pre-enumerate every possible combination of spend limit, time window, and provider whitelist. Third-party caveats allow conditions to be evaluated dynamically by specialized services that the issuer need not even be aware of at mint time. The macaroon becomes a composable credential whose authority is verifiable from the root key alone but whose conditions may span multiple independent trust domains. This is the property that makes caveated macaroons suitable as spending credentials in an agent hierarchy, and it is the property that the next phase of Kira's operation will exercise directly when presenting such a credential alongside a payment to authenticate a single API call.

The Monotonic Narrowing Property: Why Holders Can Only Restrict, Never Expand, the Permission Envelope

When the first L402-compatible relay node went live on a testbed in late 2022, its operator discovered within hours that a subordinate process had correctly refused to honor a macaroon whose caveat list had been truncated. The credential looked superficially valid. It carried a legitimate root signature. But the HMAC chain terminated one link too early, and the verifier rejected it without hesitation. No policy engine made that decision. No access-control list was consulted. The math itself enforced the boundary. That incident crystallized a property so fundamental to the macaroon construction that it deserves to be internalized as a mental model applicable far beyond bearer credentials: permissions, once narrowed, cannot be re-expanded by any party that lacks the root key.

The mechanism is straightforward once you trace the chain. A macaroon begins as a root key `k_root` and an identifier, producing an initial HMAC tag `sig_0 = HMAC(k_root, identifier)`. Each first-party caveat `c_i` derives a new tag `sig_i = HMAC(sig_{i-1}, c_i)`. The final

signature is the terminal tag in this chain. Verification reconstructs the chain from k_root forward. If a holder attempts to remove a caveat from the middle of the sequence, the reconstruction diverges at that link and every subsequent tag mismatches. If the holder tries to append a caveat that loosens a prior restriction, the new caveat only adds an additional predicate to the conjunction of all existing predicates. The verifier evaluates every caveat in order, and the permission set is the intersection of all of them. Adding "amount ≤ 100,000 sats" after "amount ≤ 50,000 sats" does not widen the envelope to 100,000. It merely asserts a redundant upper bound. The tighter constraint survives. This is monotonic narrowing: each successive HMAC layer can only shrink or leave unchanged the set of operations the credential authorizes, because the algebraic structure is a conjunction of predicates chained through a one-way function.

Third-party caveats extend this property across trust boundaries without puncturing it. When Kira's operator wants to delegate spending authority to a sub-process but enforce a live budget check through an independent ledger service, the operator constructs a third-party caveat containing a caveat key ck encrypted to the discharge service's public key, along with a predicate such as "remaining_hourly_budget > requested_amount." The operator computes sig_{n+1} = HMAC(sig_n, ck), binding the caveat into the chain. The sub-process, upon presenting the macaroon, must also present a discharge macaroon obtained from the budget-ledger service. That discharge macaroon is itself an HMAC chain rooted in ck, and the verifier binds it to the primary chain by computing HMAC(sig_discharge, sig_primary). The critical privacy geometry here is that the budget service learns only the predicate it must evaluate. It never sees k_root, never sees sibling caveats about provider identity or time windows, and never holds enough material to reconstruct a valid root credential. The sub-process, meanwhile, possesses the macaroon but cannot extract ck from the encrypted blob without the discharge service's private key. Neither party alone can forge a credential that bypasses the budget check.

This information-theoretic isolation is what separates macaroon-based attenuation from any scheme built on shared secrets or centralized policy stores. The root operator's key remains confined to the minting service and the verifying service. The discharge service holds only its own decryption key and the caveat-specific ck it unwraps per request. The bearer holds a token whose validity depends on assembling proofs from

parties that each see a strict subset of the overall permission structure. Compromise of the discharge service yields the ability to forge budget attestations, but not to mint root credentials. Compromise of the bearer yields a token that is useless without a fresh discharge proof. The attack surface is partitioned by construction.

The mental model to carry forward is this: an HMAC-chained caveat list is a cryptographic ratchet. It turns in one direction. Every link added tightens the aperture or delegates a verification obligation to a third party that can only further tighten, never loosen, because the conjunction of predicates only grows. When Kira assembles a credential for presenting to a new GPU provider, stacking a budget cap from one service, a time window from another, and a provider-set restriction minted locally, each caveat composes into a single unforgeable token whose permission envelope is precisely the intersection of all three constraints. The next section traces exactly that assembly.

Hierarchical Agent Delegation: A Parent Agent Minting Sub-Budget Macaroons for Task-Specific Child Processes

Splitting the delegation request across three GPU providers forced a question Kira's first-party caveats alone could not answer: how does a provider verify that Kira's credential is backed by a real budget balance, when the budget lives on a treasury service the provider has never contacted?

The scenario is concrete. Kira holds a macaroon whose HMAC chain already encodes first-party caveats restricting it to a set of approved providers and a validity window of roughly 120 seconds. Those caveats are sufficient for the verifying provider to check locally: walk the HMAC chain from the root key, confirm each caveat predicate, accept or reject. But a spending cap is a different kind of constraint. The provider cannot evaluate "remaining budget ≤ 400 sats" by inspecting the macaroon alone, because the provider does not operate the ledger that tracks cumulative spend. The condition must be checked elsewhere, by a service that does maintain that state. This is the structural gap that third-party caveats fill.

When the root principal mints Kira's macaroon, it appends a third-party caveat that names a budget-enforcement service as the discharge authority. The caveat embeds two objects: a caveat identifier `cId` (an opaque string describing the condition, such as "spend ≤ 400 sats against treasury T") and a verification key `vK` encrypted under the discharge service's public key. Critically, `vK` is derived from the current tail

of the HMAC chain but is never revealed to Kira or to any downstream holder. Kira receives a macaroon whose chain has been extended by HMAC(chain_n, cId), producing a new chain tail chain_{n+1}, yet Kira possesses only the macaroon blob and the location of the discharge service. To present this credential to a provider, Kira must first contact the treasury service, authenticate its current spend state, and obtain a discharge macaroon. The treasury service, knowing vK because it can decrypt the caveat payload, mints a discharge macaroon rooted at vK, potentially adding its own first-party caveats (say, "valid only for the next 30 seconds" or "single-use nonce N"). Kira now holds two macaroons: the original and the discharge. Before presenting both to the provider, Kira executes the bind-for-request step: it computes bind_sig = HMAC(0, discharge_macaroon_signature) and appends this binding to the original macaroon. The provider, possessing the root key, can then verify the entire construction. It walks the original HMAC chain, re-derives vK at the third-party caveat position, uses vK to verify the discharge macaroon's own chain, and confirms the binding ties the discharge to this specific original credential and no other.

The security invariant is strict monotonicity of permission attenuation. Consider what Kira would need to do to widen the budget cap from 400 sats to 4,000. It would have to produce a macaroon whose HMAC chain, when verified from the root key, yields a valid chain without the restrictive third-party caveat. But removing or altering the caveat changes cId, which changes the HMAC at that chain position, which cascades through every subsequent link. Without the root secret, Kira cannot recompute the chain. The credential is unforgeable in the downward direction: any holder can add caveats (further narrowing permissions), but no holder can subtract them. This asymmetry is not a policy choice enforced by software configuration. It is a mathematical consequence of HMAC's preimage resistance. Equally important, the bind-for-request step prevents credential splicing. If an adversary intercepts a valid discharge macaroon from one session and attempts to graft it onto a different original macaroon, the binding HMAC will not verify, because the discharge signature is bound to the specific original macaroon's signature. The discharge is non-transferable by construction.

What this gives Kira is an authorization architecture where the provider trusts only its own root key and the cryptographic chain, the treasury service trusts only its own key and the encrypted verification payload, and Kira never touches either secret. The credential traverses an

untrusted intermediary (Kira itself) without leaking any key material that could be used to escalate privileges. Each party verifies only what it can derive. The provider confirms the full chain. The treasury confirms the budget predicate. Kira merely ferries the proofs. With this discharge protocol established, the path opens toward a topology where Kira does not merely satisfy third-party caveats but begins minting its own sub-budget macaroons for subordinate processes, each carrying narrower caveats and their own discharge requirements, composing a hierarchy of attenuated delegation that tracks spend at every layer without any single node holding the root secret.

Why Macaroons Replace API Keys: Bearer Credential Security for Machine-to-Machine Commerce

Roughly 80% of API credential breaches in cloud environments exploit a single structural flaw: the stolen key carries every permission the original holder possessed, because the credential format encodes no mechanism for restriction. Kira now has payment channels. It can commit funds, route multi-hop HTLCs, and settle balances with any provider that speaks the protocol. But the moment it authenticates to a new GPU endpoint, it presents a static bearer token copied from its operator's account, and that token is a root secret. It carries no spending ceiling, no expiration, no binding to a specific provider pubkey. If a downstream sub-agent receives it, or if a man-in-the-middle captures it in transit, the attacker holds exactly the same authority as the human principal who generated it. Nothing in the key's byte structure permits attenuation. Policy layers may sit in front of it, but those are enforcement by convention, not by cryptographic proof.

This gap is not an implementation oversight. It is a format limitation baked into every static-secret credential scheme since the first API key was hex-encoded into a request header. The credential carries identity and authority as a single indivisible blob. What machine commerce demands instead is a bearer token whose holder can add caveats after issuance, narrowing its validity before passing it downstream, such that each recipient holds a strictly weaker credential than the one received. Macaroons, first formalized by Google Research in 2014, provide exactly this property through recursive HMAC chaining. They have waited a decade for a use case that actually requires multi-hop delegation with cryptographically enforced budget attenuation. Autonomous agent fleets operating across untrusted provider networks are that use case.

API Keys, OAuth Tokens, and Macaroons: Security Properties Under Credential Theft, Replay, and Privilege Escalation

According to a 2023 Cloudflare incident report, a single leaked API key exposed billing authority across roughly 6,000 production services before the credential could be rotated. The key had no expiry, no spending cap, no scope restriction. It was a shared secret that granted its bearer the full permissions of its issuer, and every service that accepted it had no mechanism to distinguish a legitimate caller from an attacker who had copied forty-eight hexadecimal characters from a misconfigured log file. This failure mode is not an implementation bug. It is the structural property of every credential system built on static shared secrets, and it becomes catastrophic the moment machines, rather than human operators, must authenticate spending authority to counterparties they have never encountered before.

The comparison that matters for machine-to-machine commerce is not a feature checklist but an evaluation along three security dimensions: behavior under credential theft, resistance to replay, and capacity for privilege attenuation. API keys fail on all three. A stolen API key is indistinguishable from a legitimately held one because the key carries no embedded authorization policy. The holder is the authorization. Replay is trivial because the key is a static string valid until manually revoked. And attenuation is impossible without issuing an entirely new key from the root account, which merely creates another all-or-nothing secret. OAuth tokens improve the situation partially. They introduce scoped grants and time-limited access tokens, but they depend on a round-trip to an authorization server for each scope validation. That round-trip imposes latency incompatible with high-frequency micropayment flows, and it requires the authorization server to be online and trusted, reintroducing a centralized dependency that the payment channel layer was designed to eliminate.

Macaroons resolve these failures through a single construction: HMAC-chained caveats. A macaroon begins as a root token, `mac_0 = HMAC(root_key, identifier)`, issued by the service that controls the billing relationship. Each caveat appended to the token produces a new HMAC that chains forward from the previous one: `mac_n = HMAC(mac_{n-1}, caveat_n)`. The critical property is that any party holding `mac_{n-1}` can append `caveat_n` to further restrict the token without contacting the issuer and without learning the root key. The resulting credential is strictly weaker than the one it was derived from. A

verifier who knows the root key recomputes the entire chain and confirms both integrity and authorization scope in a single pass, with no database lookup and no network call. Under theft, a stolen macaroon is constrained by every caveat already baked into it. A token that encodes `spend ≤ 500 sats`, `expires_before = 1719878400`, and `provider_id ∈ {0xA3..., 0xF7...}` is useless to an attacker after the expiry window closes, worthless against an unlisted provider, and capped at a loss ceiling the delegator chose in advance. Under replay, a caveat binding the token to a specific payment hash or channel state makes reuse detectable and unenforceable. Under privilege escalation, the HMAC chain is irreversible. Removing a caveat would require inverting the HMAC, which is computationally infeasible.

Consider the concrete shift this enables for Kira. Before this chapter's construction, Kira authenticated to GPU providers using a static API key copied from its operator's cloud account. Any provider that logged the key could charge without limit. Any breach of any provider's request logs compromised the entire billing relationship. Now Kira receives a macaroon from its operator, a token that encodes a spending cap of 10,000 satoshis, a validity window of thirty minutes, and a whitelist of three vetted provider identities. Kira can further attenuate this macaroon for a specific sub-task, appending a caveat that restricts it to a single provider and a single payment hash, then present the narrowed token when initiating an HTLC. The provider verifies the chain instantly. Kira never exposes the root key. The operator never intervenes.

The credential, in this construction, is the authorization policy. It does not reference an external permission database. It does not require the issuer to be online at verification time. It carries its own proof of scope, and that proof is unforgeable under the standard HMAC security assumption. This is the property that distinguishes a bearer credential suitable for autonomous machine commerce from a shared secret adequate only for human-supervised API calls. What remains to specify is how each caveat is constructed, what the HMAC chain guarantees mathematically at each link, and how a verifier's recomputation procedure works without any communication to the issuing principal.

Contextual Confinement at the Wire Layer: Binding Macaroon Validity to Payment-Channel State and Invoice Preimages

In the spring of 2023, a security engineer at a mid-sized inference provider traced an anomalous billing spike to a single compromised API

key. The key had been copied into a sub-agent's environment variables months earlier, granting it unrestricted access to the account's full spending authority. By the time the team rotated the credential, the attacker had consumed roughly forty times the sub-agent's intended budget. The root cause was not a sophisticated exploit. It was the credential itself: a static, unscoped secret that could not express the difference between an authorized ten-satoshi inference call and a full account drain.

This failure mode is not exceptional. It is structural. A conventional API key is a shared symmetric secret. Every service consumer that authenticates with it must possess the secret in plaintext, which means every consumer becomes a full-privilege copy of the account holder. When a single orchestrator delegates work to three sub-agents, the key must be replicated three times, tripling the attack surface without constraining any agent's authority. There is no mechanism within the key itself to encode a spending cap, a time window, or a provider restriction. Those constraints live in a separate policy layer, typically a centralized authorization server that must be consulted on every request. For a human user making a handful of API calls per minute, this architecture is tolerable. For hundreds of autonomous sub-agents executing micropayment-funded inference calls at wire speed, it collapses under its own weight. Each policy-check round trip adds latency incompatible with sub-second settlement, and the policy server becomes both a throughput bottleneck and a single point of failure whose downtime halts all delegated spending.

Macaroons resolve this structural deficiency by relocating authorization logic from a central server into the credential itself. A macaroon is a bearer token derived from a root key held only by the issuer. The issuer computes an initial HMAC signature over a unique identifier, then appends zero or more caveats, each one further constraining the credential's validity. Critically, the HMAC chain that binds these caveats is one-directional: any holder can append a new caveat, narrowing the credential's scope, but no holder can remove or weaken an existing caveat without invalidating the chain. Authority flows strictly downward. When Kira's operator mints a macaroon encoding a spending cap of five thousand satoshis, a validity window of ten minutes, and a whitelist of approved provider node public keys, Kira receives a credential that is cryptographically incapable of exceeding those bounds. Kira can further attenuate the macaroon before delegating it to a sub-process, but it cannot widen the scope it was granted.

The verification path is equally important. A provider receiving Kira's macaroon needs exactly two inputs: the root key (or a key derived from it through a known chain) and the macaroon bytes themselves. Verification consists of replaying the HMAC chain from the root, checking that each caveat is satisfied by the request context, and confirming the final signature matches. There is no database lookup, no session table, no token-exchange round trip to an OAuth authorization server. The entire check is a sequence of HMAC computations and predicate evaluations, completable in microseconds. This stateless verification model is what makes macaroons compatible with the throughput demands of micropayment-channel settlement, where a provider may validate thousands of credentialed requests per second without maintaining per-session state.

The contrast with API keys sharpens further at the revocation boundary. Revoking an API key invalidates every consumer that holds it, because they all hold the same secret. The operator must reissue a new key and redistribute it to every legitimate agent, a coordination cost that scales linearly with the number of delegates. Macaroons sidestep this problem through caveat expiry. A credential minted with a ten-minute validity window self-invalidates without any revocation broadcast. For longer-lived credentials, the operator can bind validity to a third-party caveat service that attests current authorization status, but even this check is initiated by the verifier at presentation time rather than pushed through a revocation infrastructure.

What emerges is a credential architecture matched to the topology of autonomous agent commerce. The root key never leaves the operator's secure boundary. Each subordinate agent holds a derived, attenuated token that encodes precisely the spending authority it needs and nothing more. Verification is stateless and fast. Delegation is cryptographically constrained to narrow, never widen. These are not incremental improvements over API keys. They are properties that API keys cannot possess by construction. With the bearer primitive established, the remaining question is mechanical: how does the operator select the specific caveats that bind Kira's credential to a spending cap, a time window, and a set of trusted providers? That construction is where the HMAC chain earns its architectural weight.

Selecting Caveat Granularity for Autonomous Agent Fleets: Balancing Operational Flexibility Against Blast-Radius Containment

What happens to a credential when every process that touches it inherits the full authority of the original holder?

The question is not hypothetical. Kira, as established in previous chapters, can open payment channels and route settlements across multiple hops. But the credential it uses to authenticate against GPU providers remains a static API key copied from its human operator's cloud billing account. That key is a shared secret with no intrinsic scope. It encodes no spending cap, no expiration window, no restriction on which providers may accept it. When Kira spawns a sub-process to negotiate inference pricing with a second provider, it has exactly one option: copy the same unrestricted key into that sub-process's environment. The sub-process now holds root-equivalent spending authority over the operator's entire billing account. So does any process the sub-process delegates to. The privilege surface scales linearly with the number of hops in the call chain, and every hop is a potential exfiltration point. This is the structural failure that demands attention before any discussion of caveat mechanics or budget hierarchies can proceed.

The core deficiency has a precise name: non-attenuation. An API key cannot be narrowed by its bearer before being forwarded. The bearer either shares the full secret or withholds it entirely. There is no intermediate operation, no way to derive a restricted child credential from the parent without involving the issuing server. In a single-hop request where one client authenticates to one server, this limitation is tolerable. In a multi-hop machine-to-machine topology where an orchestrator delegates tasks to sub-agents that themselves negotiate with downstream providers, non-attenuation becomes a compounding vulnerability. Each intermediary in the chain holds exactly the same authority as the root. A single compromised sub-process, a single leaked environment variable, a single logged HTTP header exposes the operator's full spending capacity. The blast radius of any credential compromise is, by construction, total. No amount of network segmentation or process isolation can compensate for a credential format that lacks the ability to express narrowing trust.

Macaroons resolve this gap with a minimal cryptographic upgrade that preserves the bearer-token model API keys already use. A macaroon begins as an identifier bound to a root key known only to the issuing

service. Any holder can append a caveat, a restriction such as a spending cap of roughly 500 satoshis, an expiration window of 30 seconds, or a whitelist of acceptable provider public keys. Each appended caveat is bound to the credential through a successive HMAC computation: the output of the previous HMAC becomes the key for the next, chaining restrictions into an unforgeable sequence. The critical property is asymmetric. Any holder can add a caveat, tightening the credential's authority. No holder can remove one, because doing so would require inverting the HMAC chain, which is computationally infeasible under standard assumptions about the underlying hash function. The credential can only narrow as it moves through the delegation graph. It can never widen.

This asymmetry delivers exactly the delegation semantics that autonomous agent fleets require. The operator issues Kira a macaroon restricted to a per-session budget and a provider whitelist. Kira, before handing that credential to a sub-process, appends a further caveat limiting the sub-process to a fraction of the remaining budget and a single provider endpoint. The sub-process receives a credential that is valid, verifiable by the target provider using the same HMAC chain, and strictly less powerful than the one Kira holds. If the sub-process is compromised, the attacker obtains a credential that can spend only the narrowed amount, only at the specified provider, only within the surviving time window. The blast radius contracts from "the operator's entire billing account" to "a bounded sliver of one session's budget." No central authorization server mediates this narrowing. No round-trip to an identity provider is required. The cryptographic structure of the credential itself enforces the containment.

The decision facing any architect building machine-to-machine payment authentication reduces to a single axis: does the credential format support attenuation, or does it not? API keys, OAuth bearer tokens, and any static shared secret fall on the non-attenuating side. They are structurally unfit for delegation chains longer than one hop. Macaroons fall on the attenuating side, offering contextual restriction without sacrificing the simplicity of bearer presentation. The comparison is not between two viable alternatives with different tradeoff profiles. It is between a credential format that creates an unbounded privilege-escalation surface and one that makes blast-radius containment a first-class cryptographic property. With that distinction established, the next question becomes mechanical: how, precisely, does the HMAC chain that

binds successive caveats guarantee unforgeability, and what grammar governs the caveats themselves?

Conclusion

The HMAC chain that threads through a macaroon's caveats is not a metaphor for constrained authority. It *is* constrained authority, encoded in a form that any verifier can check with a single traversal and a standard SHA-256 library. Each caveat appended by a successive bearer rekeys the signature under a strictly narrower predicate, and no computational effort can reverse that narrowing. The operator who mints a root macaroon bound to Kira's node pubkey, layers on a spending ceiling of 50,000 millisatoshis, a one-hour expiry window, and a three-provider whitelist, has not described a policy in some access-control configuration file. The operator has constructed a cryptographic object whose validity conditions are physically inseparable from the byte string itself. Kira can further attenuate that credential for a sub-process scoped to a single provider and a 20,000 msat budget, and the sub-process can verify its own authority without contacting the operator, without touching the network, without trusting Kira. Authorization travels at the speed of data, narrowing at every hop, widening at none. This is the primitive that sits between payment channels and autonomous spending: not a key, not a session token, but a proof of bounded permission that makes budget overruns structurally impossible rather than merely prohibited.

Yet a credential is inert until it meets a payment.

Chapter Six

L402 Protocol Flow and HTTP-Native Machine Payment Authentication

Kira holds a funded Lightning channel and a valid macaroon hierarchy. She can settle invoices in milliseconds. She can present attenuated credentials scoped to specific operations. But when a GPU provider's REST endpoint returns a bare `403 Forbidden`, she stalls. Not because she lacks funds or authorization constructs, but because the rejection carries no machine-readable answer to the only question that matters: *what must I pay, exactly, to turn this denial into access?*

The web already has a status code for this exact situation. HTTP 402 Payment Required has existed since 1997, reserved in the HTTP/1.1 specification and left dormant for nearly three decades because no payment rail was fast enough, cheap enough, or programmable enough to make it operational. Lightning changed that constraint. Macaroons provided the credential structure. But neither primitive alone closes the loop. A payment without a bound credential is just spent money. A credential without proof of payment is just an unsigned claim.

The mechanism that fuses them into a single request-response cycle is the L402 protocol flow, and it turns HTTP 402 from a historical placeholder into an autonomously executable authentication-payment handshake. To understand why this fusion works, we need to start with

the status code the web forgot and the precise mechanism by which a Lightning invoice and a macaroon root are bound together inside a single server challenge.

HTTP 402, Lightning Invoice Binding, and Preimage-as-Proof-of-Payment Bearer Token Presentation

A status code sat dormant in the HTTP specification for twenty-six years. RFC 2068 reserved `402 Payment Required` in 1997, right beside `401 Unauthorized`, and then the entire web collectively ignored it. Browsers never implemented a handler. Server frameworks never standardized a response body. The code existed as a architectural ghost, a placeholder for a payment layer that nobody built because no payment system could settle fast enough, cheaply enough, or without a human in the loop to make per-request billing rational. The web chose advertising instead.

L402 resurrects that ghost by giving the 402 response a precise, machine-executable semantics: the `WWW-Authenticate` header carries a macaroon and a Lightning invoice, together constituting a challenge that any agent with channel liquidity can satisfy in under a second. When Kira's orchestration loop hits a GPU provider's inference endpoint without credentials, the provider does not return a login page or redirect to an OAuth consent screen. It returns a 402 with a BOLT-11 payment request cryptographically bound to a macaroon identifier. Kira's Lightning node pays the invoice, extracts the preimage `r` such that `H(r)` matches the payment hash, binds that preimage to the macaroon, and resubmits the request. The act of paying and the act of authenticating collapse into one cryptographic operation. No account exists. No session persists. No human provisioned a key.

Chapter 5 gave Kira the ability to parse and constrain a macaroon's caveat chain. But a macaroon without a payment binding is just another bearer token that someone had to mint and distribute out of band.

The 402 Status Code Resurrected: Encoding Payment Requirements in HTTP Response Headers as Machine-Readable Challenges

An L402 challenge is a machine-readable payment demand encoded entirely within standard HTTP semantics, requiring no out-of-band negotiation, no pre-shared credential, and no human intervention. It repurposes the HTTP 402 Payment Required status code — dormant since its reservation in HTTP/1.1 — as the protocol-native mechanism for a

server to tell a client exactly what to pay, how to pay it, and what credential will result from payment, all inside a single response header.

The construction works as follows. When Kira sends a GET request to a GPU provider's inference endpoint, the server generates two objects from a single cryptographic root: a Lightning invoice encoding a payment hash H(r), where r is a preimage known only to the invoice issuer, and a macaroon whose identifier embeds that same payment hash. The server binds these together in the WWW-Authenticate header of an HTTP 402 response using the L402 scheme: WWW-Authenticate: L402 macaroon="<base64>", invoice="<bolt11>". The macaroon carries first-party caveats scoping the credential's validity — an expiration timestamp, a resource path restriction, a maximum usage count — but it is inert without the preimage. The invoice specifies the amount, 450 satoshis in Kira's case, and a payment hash that is the SHA-256 digest of the preimage the server holds. Because the macaroon's identifier contains this same hash, the two objects are cryptographically fused: settling the invoice is the only way to obtain the secret that activates the credential.

Kira's payment logic parses the 402 response, extracts the BOLT-11 invoice, and routes a payment through the Lightning Network. The HTLC chain propagates the payment hash hop by hop until it reaches the provider's node, which reveals the preimage r to settle the payment. This preimage flows back through the channel path to Kira. At this moment, Kira holds both pieces: the macaroon from the 402 response and the preimage from the settled HTLC. Payment and credential issuance have collapsed into one atomic step. No registration endpoint was called. No API key was provisioned. No OAuth handshake occurred.

Kira retransmits the original request with an Authorization: L402 <macaroon>:<preimage> header. The server's verification path is purely computational and entirely stateless. It extracts the preimage from the header, computes SHA-256 over it, and compares the result against the payment hash embedded in the macaroon's identifier. If the hashes match, the server knows the presenter paid the invoice, because the preimage could only have been obtained by settling the HTLC chain that terminates at the server's own Lightning node. The server then walks the macaroon's caveat chain, checking each first-party caveat against the current request context: Has the credential expired? Does the request path match the scoped resource? Is the usage count within bounds? If all caveats hold, the server serves the GPU compute result.

The entire verification requires one hash computation and a caveat evaluation loop. No database query. No session table. No token introspection endpoint.

The cryptographic invariant that makes this construction trust-free is worth stating precisely. The server never transmits the preimage in the 402 response. The client cannot forge it because it would need to invert SHA-256. The only path to the preimage is through the Lightning payment, and the Lightning payment is atomic: either the full amount settles and the preimage is revealed, or nothing settles and the preimage remains secret. This means the Authorization header is simultaneously a proof of payment and a bearer credential, unified by the hash lock that binds them. The server can verify both properties from the header alone, in constant time, with no external state.

Kira went from possessing zero relationship with this provider to authenticated, paid access in exactly two HTTP messages: one 402 challenge, one authorized retry. The provider allocated no user record, stored no session, and consulted no external auth service. Every constraint that matters — spend amount, resource scope, time window — lives in the caveats of a self-contained bearer token whose validity is anchored to a cryptographic proof the server can recompute on the fly. This is the canonical single-request L402 flow, and it is the foundation on which per-request granularity, token reuse across scoped calls, and multi-provider discovery all build.

Binding a BOLT-11 Lightning Invoice to a Macaroon Root Key: Cryptographic Linkage Between Payment Hash and Authorization Token

Kira's inference agent hit the GPU provider's /v1/completions endpoint for the first time at 02:14:07 UTC. No API key existed. No account had been provisioned. The provider's reverse proxy returned a three-digit status code that most web frameworks had ignored for decades: **402 Payment Required**. Alongside it came everything Kira's agent needed to turn a stranger's server into a paid resource in under two seconds.

The 402 response carries a WWW-Authenticate header whose payload contains exactly two objects: a serialized macaroon and a BOLT-11 Lightning invoice. These are not independent artifacts. The macaroon's identifier field embeds the payment hash H(preimage) drawn from the same invoice. This single hash is the cryptographic pivot that fuses authorization and payment into one atomic primitive. The

server generated the preimage, computed its SHA-256 hash, wired that hash into the invoice's payment_hash field, and simultaneously embedded it in the macaroon's identifier before sealing the HMAC chain with its root key. No external coordination service was consulted. The binding is purely local, purely cryptographic.

The client's task is direct. Parse the WWW-Authenticate header. Extract the macaroon and the BOLT-11 invoice. Route payment for the invoice over the Lightning Network. The moment the payment settles, the receiving node releases the preimage to the paying node through the HTLC resolution chain. That preimage is the missing half of the credential. Without it, the macaroon is inert. The hash embedded in the macaroon's identifier has exactly one valid preimage, and the only way to obtain it is to pay. Payment and authentication collapse into a single act. There is no way to forge access without producing a SHA-256 collision on the payment hash.

Kira's agent now holds two objects: the original macaroon from the 402 challenge and the preimage obtained through settlement. It constructs an Authorization header using the L402 scheme, concatenating the base64-encoded macaroon with the hex-encoded preimage. The next HTTP request to the same endpoint carries this header. The server's verification path is stateless: extract the preimage from the header, extract the payment hash from the macaroon's identifier, compute SHA-256(preimage), and compare. If the hashes match, the HMAC chain on the macaroon is verified against the root key to confirm no tampering. Two checks. No database lookup for session tokens. No call to a third-party identity provider. The server holds only its root key and the logic to verify HMACs and hash preimages.

The elegance is structural. The server never stores the preimage. The client never learns the root key. The payment hash serves as a commitment that binds the two parties through the Lightning settlement layer without either trusting the other. The macaroon's HMAC chain guarantees that no caveat has been altered since issuance. The preimage proves payment occurred. Together they form a bearer credential whose validity is self-evident from its own cryptographic contents. Every verification is a local computation.

This is what Kira's agent accomplished in that sub-two-second window: it received a challenge, paid an invoice for a single inference call, obtained the cryptographic receipt, and presented a fused credential that unlocked exactly one request. No registration flow. No OAuth dance.

No stored API key rotating on a cron job. The credential's power and its constraints now live entirely in the macaroon's caveat chain, and those caveats are where the next layer of control begins.

Preimage Revelation as Bearer Credential: Why the Settlement Receipt Doubles as the Authentication Proof

A bearer credential is a token whose mere possession authorizes access. No identity check, no session lookup, no challenge-response handshake beyond presenting the artifact itself. The mental model that unlocks L402 is recognizing that a Lightning payment preimage already satisfies this definition, because only an entity that settled the invoice can possess the value that hashes to the payment hash embedded in the challenge. The settlement receipt *is* the authentication proof. These two functions collapse into one cryptographic object, and the protocol that binds them operates entirely within the HTTP semantics that every web client and server already speak.

When Kira's agent issues a GET request to a GPU provider's `/infer` endpoint, the server returns HTTP 402 Payment Required. The response carries a `WWW-Authenticate` header using a custom `L402` scheme that encodes two objects: a root macaroon `M` (already signed via the HMAC chain specified in Chapter 5) and a Lightning invoice whose payment hash `H(r)` is the SHA-256 digest of a secret preimage `r` held by the provider's Lightning node. The macaroon's identifier field contains this same payment hash, creating an irrevocable cryptographic binding. No preimage, no valid token. No payment, no preimage. The turnstile does not check your name. It checks whether you hold the coin that fits its slot, and the slot's shape is a hash function.

The client's next move is atomic. Kira extracts the invoice, routes a payment of 50 satoshis through the Lightning Network via the standard HTLC cascade, and upon settlement, the final hop reveals the preimage `r`. This value never existed in Kira's possession before payment, and it cannot be derived from public information. It is the cryptographic receipt that proves settlement occurred. Kira then constructs the `Authorization` header as `L402 <base64(M)>:<hex(r)>` and replays the original request with this header attached. One round-trip. No registration, no OAuth dance, no API key provisioned by a human operator.

Server-side validation is a three-step constant-time verification. First, the server checks the macaroon's HMAC signature chain to confirm it was minted by the server's own root key and has not been tampered with. Second, it computes `SHA-256(r)` and compares the result against

the payment hash stored in the macaroon's identifier. If the digest matches, the preimage is genuine and the invoice was settled. Third, it evaluates every caveat in the macaroon's chain: expiration timestamps, service-tier restrictions, request-count limits, whatever attenuation conditions the server baked in at issuance. All three checks must pass. Failure at any stage returns 401 Unauthorized. The server never queries a database of sessions. It never contacts the Lightning node to confirm payment. The preimage itself is the proof, verifiable locally in microseconds.

This model reveals a non-obvious structural advantage: the authentication credential is *unforgeable without economic cost*. Traditional bearer tokens like API keys can be copied at zero marginal expense. A leaked key grants unlimited access until revoked. The L402 preimage, by contrast, cost real satoshis to obtain. Every credential presented to the server represents settled economic value. The turnstile does not merely rotate when you push. It rotates because the coin dropped, and the coin cannot be photocopied. This property transforms the economics of credential abuse. Brute-force scraping, credential stuffing, unauthorized redistribution: each requires actual Lightning payment per credential, converting an access-control problem into a cost-enforcement problem.

The model applies best when access granularity matches payment granularity. Per-request pricing for inference endpoints, per-query billing for database lookups, per-second metering for streaming compute. It applies less cleanly when long-lived sessions dominate, because repeated invoice-pay-present cycles add latency that session cookies avoid. The boundary is architectural: if the resource is stateless and each call is independently valuable, L402 fits perfectly. If the resource requires persistent server-side state across dozens of sequential interactions, a single L402 payment that issues a time-bounded macaroon with a caveat like `expiry < now + 3600` bridges the gap, collapsing the session into a single payment event while preserving the bearer-credential model.

With the complete challenge-response cycle now specified at the wire level, the operative question shifts from protocol mechanics to agent decision logic. Kira holds the 402 response. The invoice demands 50 satoshis. The macaroon caveats constrain usage to a single inference call expiring in 60 seconds. Everything needed to *execute* the payment is in hand.

L402 Session Lifecycle: From Server Challenge Through Invoice Settlement to Authenticated API Access

An L402 session is a deterministic state machine with exactly one moving part: a preimage.

Kira's HTTP client fires a GET request to a GPU provider's inference endpoint and receives something most software has never been programmed to handle. The response carries status 402, and its `WWW-Authenticate` header encodes two objects in a single line: a macaroon whose caveats bind the credential to a specific service, a specific tier, and an expiration window, and a Lightning invoice whose payment hash commits to the preimage that will unlock that credential. The amount is 43 satoshis for one inference slot, roughly eleven-thousandths of a cent. No redirect to a billing portal. No OAuth dance. No human in the loop. Just a machine-readable challenge that says: pay this invoice, bind the preimage to this macaroon, and present the result as a bearer credential on your next request. Kira has approximately 90 seconds before the invoice's `min_final_cltv_expiry` makes the offer irredeemable, and the multi-hop route her Lightning node selects must find sufficient liquidity across every channel in the path.

What happens in that window, from invoice extraction through HTLC settlement through credential construction to the retry request that yields a 200 OK, is the protocol glue that converts two independent primitives into a single atomic authenticate-and-pay action at the HTTP layer. Every mechanism that follows in this chapter, per-request gating, budget attenuation, delegated credential chains, depends on this lifecycle executing correctly under adversarial conditions where servers may revoke terms, channels may lack capacity, and invoices may expire mid-flight.

Challenge-Response Sequence: Server Emits WWW-Authenticate Header, Client Parses Macaroon and Invoice, Lightning Node Settles Payment

An L402 challenge-response exchange is a state machine with exactly four transitions: the client's initial request, the server's 402 challenge carrying a macaroon-invoice pair, the client's off-chain Lightning settlement, and the client's authenticated re-request bearing the completed credential. Each transition mutates a precise piece of protocol state. The macaroon arrives inert, missing the one cryptographic witness that activates it. The Lightning invoice encodes that witness behind a hash lock.

Payment extracts it. What follows walks through every byte exchanged and every verification performed across those four transitions, so you can trace the full lifecycle from unauthenticated HTTP call to paid, authenticated API access in a single round-trip.

Step 1: Issue the Initial Unauthenticated Request

The lifecycle begins when the client sends a standard HTTP request to a resource that requires payment-gated access. No `Authorization` header is present. The request is syntactically identical to any other REST call, carrying the target path, method, and payload. For Kira, this is a `POST` to the GPU provider's `/inference` endpoint with a model identifier and prompt payload. The server receives the request, determines that no valid L402 credential accompanies it, and prepares the challenge. The critical detail: the server does not return `401 Unauthorized`. It returns `402 Payment Required`, a status code that has waited decades for a protocol to give it operational meaning.

1. Construct the HTTP request with the target method, URI, and body. No authentication headers are included.
2. Send the request to the server endpoint. Expect a non-`200` response on the first call to any payment-gated resource.

Step 2: Parse the Server's WWW-Authenticate Challenge Header

The server's `402 Payment Required` response carries a `WWW-Authenticate` header with the scheme `L402`. This header contains two base64-encoded values: a serialized macaroon and a Lightning payment request (BOLT 11 invoice). The macaroon is freshly minted for this specific request. Its root key is held server-side, its identifier embeds the payment hash `H(preimage)`, and its caveats encode the access scope, resource identity, and expiry window. The invoice's payment hash is the same `H(preimage)` bound into the macaroon identifier. This binding is the architectural hinge of the entire protocol. The client must extract both artifacts from the header and validate their structural integrity before proceeding. The macaroon is not yet a usable credential. It is a locked container whose activation key is the preimage hidden behind the invoice's hash lock. Without that preimage, the macaroon proves nothing.

1. Extract the `WWW-Authenticate: L402 macaroon="<base64>", invoice="<base64>"` header from the 402 response.
2. Decode the macaroon. Inspect its identifier to locate the embedded payment hash. Parse the first-party caveats to confirm the access scope, expiry timestamp, and any resource-specific constraints match the intended request.
3. Decode the BOLT 11 invoice. Verify that the invoice's payment hash matches the payment hash in the macaroon identifier. Confirm the invoice amount, expiry (`min_final_cltv_expiry`), and destination public key are acceptable.
4. If the payment hash in the macaroon identifier does not equal the payment hash in the invoice, abort. The server has issued an inconsistent challenge, and settling the invoice would yield a preimage that cannot complete this macaroon.

Step 3: Settle the Lightning Invoice to Obtain the Preimage

With the invoice validated and the payment hash confirmed to match the macaroon's identifier, the client instructs its Lightning node to pay the invoice. The node finds a route to the invoice's destination, constructs the HTLC chain across intermediary hops, and forwards the payment. When the destination node reveals the preimage `r` such that `SHA-256(r) == payment_hash`, every hop in the route settles atomically. The client's node receives `r`. This preimage is the cryptographic witness that transforms the inert macaroon into a valid bearer credential. The payment and the authentication are not two separate operations linked by a database row. They are one cryptographic operation: learning `r` simultaneously proves payment and completes the credential. No sign-up form, no API key provisioning endpoint, no OAuth authorization code exchange. The preimage is the key.

1. Pass the decoded BOLT 11 invoice to the local Lightning node's `sendPayment` (or equivalent) RPC.
2. The node resolves a route via its channel graph, locks outgoing HTLCs, and propagates the payment to the destination.
3. On successful settlement, capture the returned preimage `r` from the payment response. Store it alongside the cached macaroon bytes.
4. If payment fails (no route, insufficient liquidity, invoice expired), the client retains the macaroon but cannot complete it. It may re-request a fresh 402 challenge or retry payment if the invoice has not expired.

Step 4: Construct and Send the Authenticated Re-Request

The client now assembles the completed L402 credential. It encodes the original macaroon bytes and the preimage into the `Authorization` header using the `L402` scheme: `Authorization: L402 <macaroon>:<preimage>`. The client re-issues the original HTTP request, this time with this header attached. The server receives the re-request, extracts the macaroon and preimage, and performs three verifications in sequence. First, it hashes the preimage and confirms `SHA-256(preimage) == payment_hash` embedded in the macaroon's identifier. This proves payment. Second, it verifies the macaroon's HMAC chain against its root key, confirming the macaroon was minted by this server and has not been tampered with. Third, it evaluates every first-party caveat: is the resource correct, has the expiry passed, does the requested operation fall within scope? If all three checks pass, the server serves the response. The entire additional cost compared to a standard authenticated request is one extra HTTP round-trip, the 402 challenge, adding latency on the order of a single RTT plus invoice settlement time.

1. Encode the credential as `Authorization: L402 <base64(macaroon)>:<hex(preimage)>`.
2. Re-issue the original HTTP request (same method, URI, body) with the `Authorization` header appended.
3. The server verifies: (1) `SHA-256(preimage) == macaroon.identifier.payment_hash`, (2) HMAC chain integrity against the server's root key, (3) all caveat predicates satisfied.
4. On success, the server returns the requested resource with a `200 OK`. On failure at any verification step, the server returns `402` (credential invalid or expired) or `400` (malformed header).

You have now traced every state transition in a single L402 exchange: from the unauthenticated request that triggers the 402 challenge, through the parsing and validation of the macaroon-invoice pair, to the Lightning settlement that yields the preimage, and finally to the authenticated re-request where payment proof and credential verification collapse into one check. Kira's first autonomous pay-per-call inference request is complete. No human provisioned an API key. No OAuth token was exchanged. The credential was born from a payment, scoped by caveats, and verified by hash. With this single-request lifecycle estab-

lished, the next question becomes how the server encodes fine-grained pricing into those caveats and how the client manages credential caching across sequences of calls, turning a one-shot exchange into a sustained, budget-aware session.

Token Reuse, Expiry Windows, and Session Scoping: Managing L402 Credential Lifetime Across Sequential API Requests

At 2:47 AM on a Tuesday in late 2024, a routing optimizer named Kira-7 fired a POST request at a GPU provider's /infer endpoint. The server rejected it. Not with a 401. Not with a 403. It sent back an HTTP 402: Payment Required. That single status code carried everything Kira needed to turn a rejection into authenticated access within roughly 900 milliseconds.

The 402 response arrived with a WWW-Authenticate header encoding two objects. The first was a root macaroon, already loaded with first-party caveats specifying the access scope (endpoint /infer, method POST), an expiry window (300 seconds from issuance), and a pricing tier (a single inference call at a declared satoshi cost). The second was a Lightning payment hash, H(r), binding this credential to a specific BOLT-11 invoice. These two objects are not independent artifacts. The payment hash is embedded in the macaroon's caveat chain through the HMAC construction detailed in Chapter 5. Tamper with the hash and the macaroon's signature verification fails. Swap a different invoice and the binding breaks. The server issued one atomic challenge.

Kira's agent logic parsed that header, extracted the invoice, and forwarded it to its local Lightning node. The node resolved the payment route, propagated HTLCs across the channel graph, and settled the invoice. Settlement yielded exactly one thing: the preimage r such that SHA-256(r) = H(r). That 32-byte preimage is the cryptographic proof-of-payment. Kira now held both pieces. It concatenated the base64-encoded macaroon and the hex-encoded preimage into a single token string, formatted as L402 <macaroon>:<preimage>. It placed this token in the Authorization header and re-issued the original POST to /infer.

The server's validation path is stateless and fast. It extracts the preimage from the presented token, computes SHA-256(r), and checks the result against the payment hash baked into the macaroon's caveat. Match means the invoice was paid. No database lookup. No session table. The server then walks the macaroon's caveat chain, verifying each first-party caveat against the current request context: correct endpoint,

correct method, timestamp within the expiry window. Every caveat must pass. If the HMAC chain verifies and every caveat holds, access is granted. The entire check is a sequence of hash computations and string comparisons.

This loop replaces the full traditional auth stack. No pre-registered API key existed for Kira on that server. No OAuth client ID. No refresh token rotation. No session cookie. One challenge, one payment, one fused credential, one verification. The 402 response is the registration, the invoice settlement is the authentication, and the macaroon caveats are the authorization policy. All three collapse into a single HTTP round-trip pair.

The credential Kira assembled is a bearer token with a built-in expiration. The 300-second expiry caveat means Kira can reuse that same `Authorization` header for subsequent requests to `/infer` until the window closes. Each reuse triggers the same stateless verification on the server side. After 300 seconds, the caveats fail, the token dies, and Kira faces a fresh 402 challenge if it needs another call. No revocation list. No token-refresh endpoint. The credential's lifetime is cryptographically self-enforcing.

What happens when the window closes, when an invoice expires before payment completes, or when an adversary replays a captured preimage is a different question entirely. The happy path is now fully specified. The failure modes come next.

An Autonomous Agent Purchasing Inference Slots: End-to-End L402 Handshake from Discovery Through Authenticated Completion

An L402 session is a single atomic event in which authentication and payment collapse into one protocol exchange. No accounts. No stored keys. One challenge, one settlement, one proof.

Kira's HTTP client fires a POST to a GPU provider's `/v1/inference` endpoint carrying nothing but the inference payload. The provider's gateway has never seen this caller. It has no client database to consult, no OAuth token store to query. What returns is an HTTP 402 Payment Required response, and packed inside the `WWW-Authenticate` header sits the entire negotiation: a macaroon encoding access conditions and a Lightning invoice encoding the price. The macaroon's root identifier contains a payment hash `H(r)` for some secret preimage `r` that only the Lightning Network's settlement path can reveal. The invoice encodes the same `H(r)` as its payment condition, denominated at

roughly 50 satoshis for a single inference slot with a 60-second expiry window. That shared payment hash is the cryptographic hinge binding credential to payment. The server issued both artifacts in a single response, spent no state doing it, and now waits.

Kira's payment logic, built on the channel and routing infrastructure established through multi-hop HTLC forwarding, takes the invoice and pushes it through its Lightning node. The HTLC chain propagates through intermediate nodes until it reaches the provider's node, which releases the preimage `r` upon claiming the funds. Settlement completes in under two seconds. Kira now holds two objects: the original macaroon from the 402 response and the preimage `r` extracted from the settled HTLC. It constructs an `Authorization: L402` header by concatenating the base64-encoded macaroon with the hex-encoded preimage, then resubmits the identical POST to `/v1/inference`.

The provider's gateway performs a single verification sequence that fuses payment confirmation with access control. It extracts the preimage from the header, computes `H(r)`, and checks that the result matches the payment hash embedded in the macaroon's root identifier. A match proves Kira paid. No match, no access, no ambiguity. The gateway then walks the macaroon's caveat chain, verifying that the request falls within the credential's scope: the target endpoint matches `/v1/inference`, the timestamp sits within the 60-second validity window, and the usage count has not exceeded one. Every caveat passes. The gateway forwards the inference payload to the GPU cluster and streams the result back. Kira receives a 200 OK with the completed inference. The entire exchange, from initial rejection through payment through authenticated resubmission, consumed roughly three seconds and zero pre-registered state.

Consider the failure boundaries. If Kira's Lightning node cannot find a route to the provider within the invoice's expiry window, the HTLC times out, the preimage is never revealed, and no valid credential can be constructed. Kira holds a macaroon with no matching preimage, which is cryptographic deadweight. If the macaroon's temporal caveat expires before Kira resubmits, the gateway rejects the credential even though payment occurred, because the caveat chain enforces the economic scoping independent of the payment proof. In both cases, the protocol's response is clean: re-initiate the 402 challenge. Request a fresh macaroon-invoice pair. Pay again or walk away. The server never accumulates zombie sessions, dangling tokens, or half-authorized states.

What Kira gained in this exchange extends beyond a single inference result. The attenuated credential model from the macaroon construction internals described in Chapter 5 now operates inside a live HTTP cycle. The HMAC-chained caveats that were abstract structural properties became runtime enforcement gates. The Lightning preimage that was a settlement primitive in earlier chapters became an authentication factor. Every layer composed without a trust gap between them. The provider demanded no signup form, issued no API key, stored no client record. Kira demanded no contractual relationship, no billing portal, no human approval. The 402 response itself contained the complete terms of the transaction, and settlement of those terms produced the credential that satisfied them. That composability, credential and payment fused at the protocol layer, is what makes the next problem tractable: when multiple providers return competing 402 challenges simultaneously, Kira will need to evaluate not just price but caveat structure, expiry tolerance, and routing cost before choosing which invoice to settle.

Automated Budget Enforcement via Embedded Macaroon Micro-Budget Ceilings at the Request Layer

A credential that never says *no* is not a credential. It is a vulnerability operating at machine speed.

Kira has already completed the L402 handshake. She holds a valid macaroon bound to a Lightning payment preimage, and every GPU endpoint she hits returns 200 OK without hesitation. Now suppose a malformed upstream prompt triggers a retry loop. In roughly ninety seconds, Kira fires an estimated 4,000 paid inference requests. Each one authenticates flawlessly. Each one settles a Lightning HTLC. The token works exactly as specified, and that is precisely the failure: nothing in the credential constrains *how many times* or *how many satoshis* it can authorize. The L402 flow proved Kira can pay. It did not prove she can be trusted to pay unsupervised.

The structural fix is not an external rate limiter bolted onto the gateway, and it is not a human approving each call. It is a property of the token itself. Macaroon attenuation, the HMAC-chained caveat construction already specified in Chapter 5, can encode a satoshi ceiling, a time window, and a request count directly into the bearer credential. Both Kira's own request layer and the provider's verification gateway evaluate these caveats independently before any HTLC is constructed. The budget becomes cryptographically self-enforcing: the same token that proves payment eligibility also defines and limits the spending en-

velope. But the hard problem surfaces immediately. Neither party trusts the other's running count, and the only shared source of truth is the chain of caveats sealed inside the macaroon's HMAC construction.

Embedding Spending Caveats: Encoding Per-Request Satoshi Limits, Cumulative Ceilings, and Rate Bounds Within the Macaroon HMAC Chain

A spending caveat is a predicate serialized into the macaroon's HMAC chain that constrains what the bearer can do with the credential. It is not metadata. It is not a policy hint. It is a cryptographically bound condition that the verifier must evaluate to true before any resource unlocks. When that predicate encodes a satoshi limit, a cumulative ceiling, or a rate bound, the L402 credential stops being a simple proof of payment and becomes an active budget governor embedded directly in the HTTP Authorization header.

Consider the mechanical reality of what happens at the provider's admission gate. The GPU endpoint receives an HTTP request carrying an `Authorization: L402 <macaroon>:<preimage>` header. The verifier reconstructs the HMAC chain from the root key, confirming no caveat was altered or removed. Then it evaluates each caveat predicate in sequence. A caveat reading `max_amount_sat = 500` forces the verifier to check the invoice amount encoded in the payment proof. A caveat reading `cumulative_budget_sat <= 50000` forces a lookup against the session's running total. A caveat reading `requests_per_hour <= 120` forces a rate check against a sliding window. Each evaluation is O(1) against the presented credential and its associated session counter. If any predicate fails, the provider returns a 402 before dispatching a single GPU cycle. The budget policy travels with the token, not in a sidecar service, not in a database row managed by a billing team.

The attenuation-only property established in Chapter 5 does the structural work here. Kira's human principal mints a root macaroon with a broad envelope, perhaps `cumulative_budget_sat <= 5000000` and `valid_until = end_of_month`. Kira receives this root credential and immediately attenuates it. For a specific provider session, Kira appends `max_amount_sat = 500` and `cumulative_budget_sat <= 50000` and `requests_per_hour <= 120`. Each appended caveat feeds through `HMAC(previous_sig, caveat_bytes)`, producing a new signature that covers the tighter constraint. The critical invariant holds: Kira can only narrow the envelope, never widen it. Removing a caveat would break the HMAC chain, and

the provider's verifier would reject the token on cryptographic grounds alone. The result is a tamper-evident budget hierarchy where every layer of delegation can only tighten spending authority.

Before these caveats existed in Kira's credential, a misconfigured retry loop could drain an entire Lightning channel in seconds. No mechanism existed between "payment succeeds" and "resource unlocks" to ask whether this payment should succeed given everything that came before it. The credential was binary: valid or invalid. Now the credential carries its own spending constitution. The provider's verifier enforces per-request caps, cumulative ceilings, and rate bounds at the HTTP admission gate without maintaining any trust relationship with Kira's principal and without consulting any external budget service. The enforcement is stateless from the provider's perspective because the cryptographic proof of budget compliance travels inside the token itself. Only a lightweight session counter, tracking cumulative satoshis claimed against a given macaroon root, sits on the provider side, and even that counter resets when the session's time-window caveat expires.

This construction transforms every L402-protected API endpoint into a self-enforcing budget boundary. The provider does not need to know Kira's total monthly allocation or how many other providers share that allocation. It only needs to verify that the presented macaroon's caveats permit this specific request at this specific moment for this specific amount. The budget hierarchy is distributed across the credential chain itself, not centralized in a ledger. Each attenuated macaroon is a scoped spending authority that any compliant verifier can evaluate independently. When Kira begins operating across multiple GPU providers simultaneously, allocating sub-budgets and reconciling spend across heterogeneous endpoints, the mechanics specified here become the atomic enforcement primitive on which that orchestration depends.

Server-Side Caveat Verification: Rejecting Over-Budget Requests Before Invoice Generation and the Enforcement Trust Boundary

Kira's worker process had been alive for eleven seconds. In that span it fired forty-seven inference requests against a GPU provider's endpoint, each costing roughly 100 satoshis. Forty-seven requests cleared. The forty-eighth did not. The provider's gateway rejected it before generating an invoice. No human intervened. No external budget service was queried. The enforcement lived inside the credential itself.

That rejection is the moment this section dissects. The provider's gateway, upon receiving an L402 macaroon alongside a payment preim-

age, does not simply validate the HMAC chain and confirm the preimage matches a settled invoice. It walks the caveat stack. Every first-party caveat in that stack is a predicate the gateway evaluates against local state. When Kira's parent process delegated a sub-macaroon to its worker, it appended a caveat: `max_spend_per_window = 5000` and `window_duration = 60`. The gateway maintains a lightweight counter keyed to the macaroon's root identifier. At request forty-eight, cumulative spend hit 4,800 satoshis. The next 200-satoshi call would push past the ceiling. The gateway returns HTTP 402 with an error body specifying `budget_exceeded`, not a fresh invoice. No HTLC is constructed. No channel bandwidth is consumed. Enforcement fires before any payment flow begins.

This ordering matters. The verification boundary sits upstream of invoice generation, which means the provider never commits liquidity to a payment it will later reject on policy grounds. The caveat check is computationally trivial compared to constructing an HTLC route probe or locking channel capacity. A single HMAC verification per caveat, a counter lookup, a comparison. The cost is nanoseconds. The alternative, letting the request proceed to invoice generation and then refusing at settlement, wastes route computation, locks up inbound liquidity for the HTLC timeout window, and forces the agent into an ambiguous failure state. Rejecting at the caveat layer eliminates that entire cost surface.

The trust boundary here is precise. The provider's gateway is the caveat verifier. It does not trust the agent's self-reported spend total. It maintains its own cumulative counter per macaroon root. The agent, conversely, does not trust the provider to enforce a ceiling the agent never specified. The ceiling is cryptographically embedded in the credential the agent presents. Because macaroon attenuation is one-directional, no entity in the delegation chain can widen a caveat. Kira's parent set 5,000 satoshis per 60-second window. The worker cannot forge a macaroon claiming 10,000. The HMAC chain would fail verification. Attenuation is irreversible and tamper-evident by construction.

Delegation hierarchies compose naturally from this property. A top-level orchestrator holds a root macaroon with a broad budget, perhaps 500,000 satoshis across a 24-hour window. It delegates to three sub-agents, each receiving a macaroon attenuated to 150,000 satoshis. One of those sub-agents spawns worker processes, each capped at 5,000 per minute. Every level strictly narrows. No runtime message to a central ledger coordinates these limits. Each credential carries its own fiscal ceil-

ing, verifiable by any gateway that shares the root HMAC key. The hierarchy is self-enforcing. The proof is the credential.

What happens at the boundary, when a ceiling is hit mid-session, is a protocol event with defined semantics. The gateway's `budget_exceeded` response signals to the agent that its current credential is exhausted for this window. The agent now faces a decision surface: wait for the window to roll over, request a fresh delegation from its parent, or escalate. That decision logic, and the provider-side fallback behavior that supports it, is where the protocol moves next.

L402 with Macaroons vs. OAuth Scopes vs. API-Key Rate Limits: Authorization Model Fitness for Machine-Scale Payment Authentication

A budget ceiling is a cryptographic predicate embedded inside a bearer credential that causes the verifying endpoint to reject a payment request before any value moves. That definition separates L402 macaroon caveats from every prior authorization model at the most fundamental layer: the spending constraint lives inside the proof of identity, not in an external policy engine, not in a rate-limit counter maintained by the server, and not in a scope string interpreted by a token introspection endpoint. The question this subtopic resolves is not which model "supports" budget enforcement in principle, but which model makes overspend structurally impossible at the moment of the HTTP handshake.

OAuth 2.0 scopes express authorization as string labels attached to bearer tokens. A scope like `compute:gpu` tells the resource server what category of action the client may perform, but it encodes nothing about how much value a single request may consume. Spending limits, if they exist at all, must be enforced by a separate authorization server that the resource server queries at runtime or by server-side state that tracks cumulative expenditure. Both paths introduce external dependencies: a database write per request, a network call to a policy endpoint, or both. At machine scale, where Kira might issue thousands of GPU inference calls within a rolling ten-second window, those dependencies become latency bottlenecks and, worse, trust assumptions. The resource server must be trusted to correctly maintain and enforce the counter. The authorization server must be trusted to remain available. Neither trust assumption is cryptographically grounded. An OAuth scope is a label, not a proof. It tells you what a client is allowed to do without binding that permission to a verifiable expenditure ceiling.

API-key rate limits operate on a simpler but equally insufficient axis. The server maps an API key to a counter, typically requests-per-second or requests-per-day, and rejects calls that exceed the threshold. This controls throughput, not value. A rate limit of 100 requests per minute says nothing about whether each request costs 10 satoshis or 10,000. When the resource being purchased has variable pricing, as GPU inference does when batch sizes, model complexity, and priority tiers shift the invoice amount per call, a flat rate limit cannot distinguish a 50-sat request from a 5,000-sat request. Budget enforcement degenerates into throughput policing, and the gap between the two widens precisely when it matters most: under adversarial conditions where a compromised agent attempts to maximize extraction per permitted call.

L402 macaroon caveats close that gap by encoding the budget constraint as a first-party caveat predicate that the verifier evaluates against the invoice amount attached to the current 402 challenge. Consider Kira's delegation chain. The root macaroon minted by Kira's controller encodes `max_total_sats ≤ 20000`. Kira attenuates a child for a specific GPU provider class with `max_session_sats ≤ 5000`. That child is further narrowed to a single session with `max_request_sats ≤ 50` and `valid_before ≤ T+3600`. Each successive caveat is HMAC-chained to its predecessor, making removal or widening cryptographically impossible without the root key. When Kira presents this leaf macaroon alongside a Lightning invoice in the L402 proof-of-payment header, the provider's verifier walks the caveat chain, checks that the invoice amount satisfies every predicate, and only then reveals the preimage that settles the HTLC. If the invoice requests 51 satoshis, the verifier rejects before any preimage is generated. No satoshis move. No state is written. No external policy server is consulted.

This is the critical architectural distinction. OAuth scopes require the resource server to interpret and enforce limits that live outside the credential. API-key rate limits require the server to maintain mutable counters that track throughput rather than value. Macaroon caveats require the verifier to evaluate immutable, cryptographically chained predicates against the specific payment amount of the current request. The enforcement is stateless at the protocol layer because the budget ceiling travels with the credential, not alongside it. The server performs a set of predicate checks against a single invoice value and either releases the preimage or returns a rejection. Cumulative session tracking, where needed, can be maintained by the client's own wallet logic or by a light-

weight counter on the verifier that only increments upon successful preimage release. But the per-request ceiling is absolute and self-contained.

The fitness verdict is unambiguous. For machine-scale payment authentication where every HTTP request carries a Lightning invoice and every response costs real satoshis, the authorization model must bind identity, permission, and expenditure ceiling into a single unforgeable artifact that the verifier can evaluate without trusting external state. Macaroon caveats inside L402 credentials are exactly that artifact. With Kira's per-request, per-session, and per-provider ceilings cryptographically locked into an irrevocable attenuation chain, the ground is set for the harder problem waiting downstream: what happens when those bounded credentials must be delegated across multiple providers simultaneously.

Conclusion

For three decades, HTTP 402 sat in the specification as a placeholder, a status code with no executable semantics. Authentication lived in one protocol layer, payment in another, and the gap between them required human-provisioned API keys, OAuth flows, subscription dashboards, and billing reconciliation cycles that no autonomous agent could navigate alone. The L402 challenge-response cycle collapses that gap into a single atomic round trip. When Kira's request returns a 402, her payment controller inspects the embedded macaroon's budget ceiling, settles the Lightning invoice if the amount falls within her delegated micro-budget, binds the resulting preimage to the macaroon, and replays the request with a bearer token that proves both authorization and payment in one header. No account creation, no identity disclosure, no server-side session. The macaroon declares the spending policy; the client's pre-settlement check enforces it cryptographically before a single satoshi moves. The server verifies a proof, not a relationship.

Every HTTP endpoint on the open internet can now be a pay-per-call service that machines access without human intermediation. But Kira currently aims this fused instrument at one known provider. The moment she faces a dozen GPU endpoints offering the same inference capability at different prices and latencies, the L402 handshake becomes a market transaction, and the protocol needs a procurement layer to match it. Discovery, comparison, and routing under real-time budget constraints turn a bilateral payment into an economic decision. That machinery is next.

Chapter Seven

Distributed Service Discovery for Autonomous Agents

Kira's payment stack is complete, and it is useless.

She can mint L402 credentials, present HMAC-verified bearer tokens to any service endpoint, and settle authenticated micropayments through off-chain state updates that finalize in milliseconds. Six chapters of protocol construction gave her the ability to pay any machine on the internet. But paying requires an address, and every address Kira knows is one a human operator hardcoded into her configuration file weeks or months ago. That file is static. When her primary GPU provider's spot pricing doubles during a demand spike, or when its p99 latency drifts above her task deadline, Kira cannot do what any rational economic actor would do. She cannot shop elsewhere. "Elsewhere" is an address she was never given.

The gap is structural, not incidental. The centralized web offers no machine-readable directory of compute providers, no protocol for publishing capabilities against which an agent can query by description rather than by name. Kira has a loaded weapon with no target. The payment rails built across previous chapters carry value, but they carry it nowhere without a discovery layer that lets an autonomous agent surface, verify, and rank providers it has never encountered before.

So the first question is foundational: if no central registry exists, how does a machine advertise a capability so that another machine, one that

has never seen it before, can find it by describing what it needs rather than naming who provides it?

DHT-Based Discovery: Publishing and Querying Service Advertisements via Kademlia Keyed by Capability Hashes

Kira hashes a capability descriptor and the network answers.

She needs 80GB VRAM, fp16 inference, sub-200ms first-token latency. She canonicalizes those constraints into a deterministic byte string, computes H(descriptor) to produce a 256-bit key, and issues a Kademlia FIND_VALUE against that key. No registry operator intermediates. No endpoint is known in advance. The XOR distance metric guides the query through successive hops, each closer in key-space to the nodes responsible for storing advertisements matching that exact capability profile. What returns is not a URL but a set of signed records, each carrying a provider's public key, pricing terms, and channel-opening parameters. The capability hash functions as a commitment to a service interface rather than a pointer to a server, so Kira can resolve queries for capability combinations that no single directory has ever catalogued.

This mechanism is the structural prerequisite for everything the chapter builds toward. With the advertisement record format already defined, the operational gap is retrieval: where records live, how they propagate, and why Kira can trust what she finds without trusting who stored it. Kademlia's XOR metric, originally designed for peer-to-peer file sharing, maps with surprising precision onto a market where the scarce resource is not bandwidth but cryptographically verifiable proof of capability. The sections ahead decompose this mapping: how canonical service descriptors become DHT keys, how advertisements are published with TTL discipline and signed proofs at XOR-closest nodes, and where the tension between replication factor, query latency, and Sybil resistance forces concrete engineering tradeoffs.

Capability-Hash Keying: Deriving Kademlia Node IDs from Canonical Service Descriptors

The moment Kira can authenticate and pay for a single API call, she hits a prior wall: she needs the provider's endpoint before any L402 handshake can begin. No centralized registry exists. No human operator pastes a URL into her configuration. The only artifact Kira holds is a structured description of what she needs: a capability requirement tuple

specifying model family, VRAM class, maximum batch size, and geographic region. That tuple, it turns out, is the key to everything.

A deterministic hash of a canonical capability descriptor is the DHT lookup key that connects demand to supply without any intermediary. The construction works as follows. Both the provider publishing an advertisement and the agent searching for compute normalize their capability descriptions into a canonical form, a sorted tuple of typed fields with a fixed serialization. The provider offering an A100-class inference slot in a European datacenter with a 64-sample maximum batch size serializes exactly the tuple `(model_family="llama-65b", vram_class="A100-80GB", max_batch_size=64, region="eu-west")`, applies SHA-256, and stores its advertisement record at the resulting 256-bit key in the Kademlia overlay. Kira, needing precisely that capability, constructs the same tuple from her own requirements, computes the same SHA-256 digest, and issues an iterative FIND_VALUE lookup for that key. Because the hash function is deterministic and the descriptor schema is fixed, both sides converge on the same point in the keyspace without coordination.

Kademlia's XOR-distance metric governs how FIND_VALUE proceeds. Kira's node maintains k-buckets partitioning the 256-bit address space by distance from her own node ID. She begins the lookup by querying the α closest nodes she knows, receives intermediate routing responses pointing to nodes whose IDs fall closer to the target hash, and iterates until she reaches the k nodes nearest the target. Those nodes either hold the advertisement record or confirm its absence. Replication factor k (typically around 20 in production overlays) determines how many nodes store copies of each record. For ephemeral compute offerings, the record TTL matters acutely: a GPU slot advertised with a 300-second TTL disappears from the DHT if the provider stops refreshing it, ensuring stale listings drain away rather than accumulate. Shorter TTLs impose higher republish overhead on providers but give querying agents fresher results. The tradeoff surface is explicit: TTL × republish-rate governs bandwidth cost, while k × record-size governs storage cost per capability bucket.

What Kira retrieves is not a bare endpoint string but a signed advertisement record. Each record contains the provider's public key pk_provider, a signed capability claim σ(capability_tuple, sk_provider), an L402-compatible payment endpoint URI, a monotonically increasing nonce, and an anti-spam proof. The anti-spam proof can take one of two forms: a proof-of-work satisfying a difficulty target H(record ‖

nonce) < D, or a verifiable stake bond referencing an on-chain deposit that the provider forfeits if the advertisement is proven fraudulent. Either mechanism raises the marginal cost of flooding the DHT with fake listings from near zero to a calculable threshold. **The hash that routes you to the record is the same hash that makes forgery expensive: capability-keyed addressing means an attacker must solve the spam proof for every distinct capability tuple they want to pollute, and the defender's verification cost is a single signature check plus one hash comparison.** This asymmetry between attack cost and verification cost is what makes decentralized discovery viable under adversarial conditions without any gatekeeper.

Kira performs three operations on each retrieved record before it enters her candidate set. She verifies σ against pk_provider to confirm authorship. She checks the anti-spam proof against the current difficulty target or queries the referenced stake contract for bond validity. She confirms the nonce is higher than any previously seen nonce for that public key, preventing replay of expired advertisements. Records that fail any check are discarded silently. Records that pass yield a verified provider identity paired with a payment endpoint Kira already knows how to negotiate, the L402 handshake from Chapter 6 slotting in without modification.

In a concrete run, Kira hashes her requirement tuple, iterates through roughly four to five Kademlia hops, and retrieves three signed advertisement records from providers she has never encountered. Each record carries a distinct public key, a valid proof-of-work, and an L402 URI. She now possesses a set of cryptographically authenticated candidates. But three candidates are not a decision. Which provider offers the best latency, the highest reliability, the most honest capability claim? The signed records prove authorship and raise spam costs, yet they say nothing about track record. That evaluation requires a different kind of evidence entirely.

Publishing Service Advertisements: Record Structure, TTL Policies, and Signed Capability Proofs Stored at XOR-Closest Nodes

The first time Kira needed a GPU slot, she had nothing but a capability requirement and a cryptographic identity. No bookmark. No directory URL. No human operator to consult. She hashed a structured descriptor and fired an iterative lookup into a Kademlia overlay. Within roughly 300 milliseconds, seven signed advertisements landed in her loc-

al buffer. Two carried expired timestamps. One failed signature verification. Four survived. Each contained an L402 payment endpoint ready for immediate authenticated settlement. That is the entire discovery path: from capability hash to verified candidate set, with no registry, no curator, no trust assumption beyond the mathematics.

A capability descriptor is a canonical record. It specifies model family, VRAM tier, latency SLA ceiling, supported batch sizes, and pricing denomination. Kira's descriptor read: `{model: "llama-70b", vram_tier: "A100-80GB", p95_latency_ms: 200, batch_sizes: [1,4,8], denomination: "lightning-sats"}`. Canonicalization means deterministic field ordering and encoding. The descriptor feeds into a SHA-256 hash, producing a 256-bit Kademlia key. Any agent anywhere, seeking the same capability class, computes the identical key. No coordination required. The XOR metric routes them to the same DHT neighborhood. This is the critical insight: capability hashing partitions the namespace by function, not by identity. Providers of equivalent resources cluster at the same key without ever exchanging a single message with one another.

Publishing an advertisement means constructing a signed record and storing it at the k-closest nodes to that capability hash. The record contains the provider's Ed25519 public key, a freshness timestamp, a TTL field (commonly around 15 minutes for compute services), the full capability descriptor, and an L402-compatible payment-endpoint URI. The provider signs the entire record with its identity key, producing σ(record, sk_provider). Republish intervals run at roughly half the TTL. This keeps bandwidth overhead logarithmic in network size. A provider serving three capability classes publishes to three key neighborhoods. Stale entries expire and vanish. The DHT is self-cleaning.

Querying agents run iterative Kademlia lookups. Kira sends FIND_VALUE RPCs, walking the XOR tree toward the target key. At each hop, the contacted node returns either the stored advertisements or closer node contacts. Convergence typically completes in O(log N) hops for a network of N nodes. The returned set is raw. Kira treats it as untrusted input. She verifies every signature against the embedded public key. She checks every timestamp against her local clock with a tolerance window. She discards any record whose TTL has elapsed. The DHT is a rendezvous layer, nothing more. Its integrity guarantee is zero. All trust derives from cryptographic verification at the consumer side.

A DHT node that lies gains nothing. It can return garbage advertisements, but every garbage record fails signature verification and

gets dropped. It can withhold valid records, but redundant storage across k nodes (typically k = 20) means suppression requires compromising a majority of the closest node set. The XOR distance metric makes targeted Sybil attacks expensive: an attacker must generate node IDs that land near a specific capability hash, and even then, honest republishers continuously refresh the k-closest set. The cost of persistent suppression scales with republish frequency. The cost of forging advertisements is exactly the cost of breaking Ed25519. The system converts economic adversaries into cryptographic adversaries, and cryptographic adversaries lose.

Kira's four surviving advertisements now sit in memory. Each one is a verified claim: a specific provider, a specific capability match, a specific L402 endpoint, a specific timestamp proving liveness. She has discovered compute resources across a fully decentralized overlay without trusting any single node in the path. The raw candidate set is complete. What she does not yet know is which of these four providers will actually deliver. That question belongs to reputation scoring, where on-chain settlement history and attestation records separate reliable providers from opportunistic ones. The advertisements carry her there.

Query Latency, Sybil Resistance, and the Tradeoff Between Replication Factor and Discovery Freshness

The first time Kira's operator watched her fail, the cause was absurd. She had the L402 macaroon ready. She had budget authority scoped to the task. She had a signed delegation chain proving she could spend. But she had nowhere to spend it. The single hardcoded GPU provider had gone offline forty minutes earlier. Kira sat idle, authenticated to nothing, because no mechanism existed for her to discover an alternative.

That failure is the starting condition for everything in this section. Discovery latency, resistance to Sybil pollution, and the tension between replication breadth and record freshness define whether a DHT-based registry can replace the hardcoded provider list. Each of these forces pulls against the others, and the design must hold all three in balance.

Start with key derivation. Kira needs fp16 inference on Llama-70B with at least 80GB of VRAM. She constructs a canonical capability string, `gpu/inference/llama-70b/fp16/>=80gb-vram`, and computes its SHA-256 hash. That hash is her lookup key. Every agent with the same requirement derives the same key independently. No coordinator, no registry endpoint, no API call. The hash itself is the rendezvous point. A provider offering that capability publishes a signed advertise-

ment at that key's address in the Kademlia DHT. The advertisement contains the provider's public key, the offered capability vector, a pricing hint in satoshis per inference-second, a Lightning Network connection endpoint, and an expiry timestamp, all wrapped in a signature σ(ad, sk_provider) verifiable against the provider's identity key.

Kademlia's XOR-distance metric routes lookups in O(log n) hops. In a network of roughly 10,000 providers, Kira resolves her capability hash in around 13 to 14 hops. Each hop costs tens of milliseconds on commodity infrastructure, putting total discovery latency well under 500 milliseconds without caching. That speed matters. Real-time compute brokering demands that Kira discover, evaluate, and commit to a provider within a narrow window. A centralized registry could answer faster, but it introduces a single point of failure that negates every trust-free guarantee the stack has built so far.

Freshness is where the design gets adversarial. Every advertisement carries an expiry timestamp and a TTL. Providers must re-sign and re-publish before expiry, or DHT nodes evict the record. A provider that vanishes mid-epoch leaves no ghost entry for Kira to waste a channel open on. But republication frequency trades directly against network bandwidth. A replication factor k of 20 means each advertisement lives on 20 nodes nearest the capability hash. Higher k improves availability under churn but multiplies republication traffic by the same factor. Lower k saves bandwidth but lets a coordinated Sybil cluster eclipse the key's neighborhood. The operator's choice of k is a commitment to a specific point on the availability-versus-cost surface, and there is no universally correct answer.

Sybil resistance in this layer is structural, not cryptographic. A Sybil operator flooding the DHT with fake node IDs near a popular capability hash can intercept lookups and return poisoned advertisements. The defense is layered. First, advertisements are signed: Kira verifies σ(ad, sk_provider) before treating any record as valid. A Sybil node that cannot produce a valid signature for a real provider's key returns garbage that fails verification instantly. Second, requiring a proof-of-work stamp or a Lightning-anchored identity bond on DHT node registration raises the cost of generating thousands of fake IDs. **The critical insight is that Sybil resistance in a service-discovery DHT does not need to prevent fake nodes from existing; it only needs to ensure that fake advertisements are cryptographically distinguishable from real ones at verification time, shifting the security boundary from**

network topology to signature validation. This is the same pattern the entire stack follows: trust the math, not the participants.

Multi-key queries add one final layer of mechanism. When Kira needs a conjunction of capabilities, say GPU type AND minimum VRAM AND geographic region, the provider publishes its advertisement under each capability hash independently. Kira performs parallel lookups across all relevant keys and intersects the result sets client-side. The intersection is cheap. The parallel lookups complete in the same O(log n) bound because they are independent routing operations. The result is a shortlist of providers matching all constraints, each entry carrying a verifiable signature, a pricing hint, and a Lightning endpoint.

Kira now holds three signed advertisements from providers she has never encountered before. She derived the lookup key from her own requirement. She verified each signature against the provider's public key. She confirmed each record's TTL is unexpired. No human configured these entries. No central server brokered the introduction. The DHT gave her candidates. What it did not give her is any reason to trust their quality, reliability, or honesty beyond the validity of their signatures. That ranking problem belongs to the next layer.

DNS-SD Integration and Decentralized On-Chain Registries Mapping Providers to Cryptographic Identities

Kira holds a funded channel, a delegated budget macaroon, and zero addressable counterparties. She can settle a payment in milliseconds but has nowhere to send it. The discovery problem precedes every other protocol layer: before negotiation, before verification, before a single HTLC locks, an agent must resolve a name to an endpoint and an endpoint to a cryptographic identity it can authenticate without phoning home to a centralized directory.

DNS-SD already solves half of this. A standard PTR query against `_gpu-inference._tcp.local` returns SRV records pointing to host-port pairs and TXT records carrying key-value metadata, including, in an agent-commerce extension, a claimed Lightning node pubkey and a reference to an on-chain identity anchor. The machinery is deployed, battle-tested, and operates over existing recursive resolvers. But DNS-SD was built so a human could glance at a printer name and decide whether to trust it. Kira has no such luxury. The TXT record's claimed pubkey is an unauthenticated string until something outside DNS proves the ad-

vertising entity controls the corresponding private key. This is the exact seam where decentralized on-chain registries bind what DNS advertises to what cryptography guarantees.

The binding operation is precise: a provider registers a commitment on-chain that links its public key to a set of capability descriptors, and Kira cross-references the DNS-SD advertisement against that on-chain anchor in a single additional lookup. If the pubkey in the TXT record matches a registry entry whose signature she can verify against the chain's state root, the advertisement graduates from claim to credential. If it does not, Kira discards the record before any economic exposure begins.

DNS-SD SRV and TXT Records as Bootstrap Pointers into DHT-Backed Service Namespaces

The moment Kira can authenticate a single provider's payment challenge tells her nothing about which providers exist in the first place. That gap between credential verification and endpoint discovery is where DNS-based service discovery enters the architecture, not as a legacy curiosity but as the only multicast-capable, zero-configuration advertisement mechanism already deployed on every subnet Kira will ever touch. The key realization is that DNS-SD's TXT record format, a set of UTF-8 key-value pairs limited to roughly 1,300 bytes in practice, carries exactly enough structured payload to bootstrap a cryptographic identity handshake without requiring any custom discovery protocol.

Consider the concrete record set. Kira issues a PTR query for `_gpu-inference._tcp.local` and receives three instance names. For each instance she resolves an SRV record yielding host and port, then a TXT record containing fields that extend the standard DNS-SD schema into cryptographic territory. A provider's TXT record carries `pk=<33-byte compressed secp256k1 public key>`, identifying the Lightning node that will issue L402 challenges. It carries `l402uri=https://provider.example/v1/infer`, pointing Kira to the payment-gated endpoint. It carries `caphash=<SHA-256 digest of the provider's capability advertisement document>`, binding a specific machine-readable capability description to this advertisement. And it carries `regsig=σ(instance‖pk‖caphash, sk_provider)`, a signature over the concatenation of instance name, public key, and capability hash, proving that whoever controls the advertised key authorized this specific DNS-SD entry. These four fields transform a plain service browse response into a self-authenticating provider advertisement. Kira

does not yet trust it, but she now holds enough material to begin verification.

Trust arrives from the second layer. An on-chain registry, implemented as a minimal contract on a Bitcoin-anchored sidechain or propagated as a Nostr kind-30078 parameterized replaceable event with NIP-05 domain binding, maps each provider public key to a persistent identity record. That record contains the provider's staked collateral amount, the block height at which the key was first registered, and a signature chain proving key continuity across any rotations. When Kira receives the TXT record's `pk` field, she queries this registry using the public key as the lookup index. A valid registry entry confirms three properties simultaneously: the key has been staked with economic commitment that would be forfeited on misbehavior, the key has persisted since a specific registration height establishing a continuity baseline, and the domain binding in the registry entry matches the domain from which the DNS-SD record was served. If any check fails, Kira discards the candidate. If the `regsig` in the TXT record does not verify against `pk`, she discards it. If `pk` appears in no registry entry, she discards it. Only candidates surviving both DNS-level signature verification and on-chain identity confirmation enter her candidate table.

This two-layer cross-referencing produces something no centralized directory can offer: a discovery result whose integrity degrades gracefully under partial adversarial control. An attacker who compromises the local DNS resolver can inject a fraudulent SRV/TXT record, but the injected public key will lack a registry entry with valid staking history. An attacker who inserts a fraudulent registry entry must post collateral and cannot forge the DNS-SD TXT signature without the corresponding private key. The only successful attack requires simultaneous control of both the DNS infrastructure and the registry's private key, a conjunction whose cost scales with the staked amount. DNS proves domain control. The registry proves key continuity and economic commitment. Together they form a bootstrapping handshake where neither layer alone is sufficient, but their composition closes the trust gap that a single-source directory leaves open.

After processing all three browse responses against the registry, Kira holds a verified candidate table: three previously unknown GPU inference providers, each pinned to a confirmed cryptographic identity with a known staking status and a capability hash she has not yet parsed. She arrived at this table without consulting any centralized directory, without hardcoding any endpoint list, and without trusting any single

network authority. The candidate set is real, each entry backed by verifiable economic skin in the game. What remains is to fetch the full capability documents those hashes commit to, parse the advertised throughput and pricing parameters, and rank the three providers against each other by metrics Kira can independently verify.

On-Chain Registry Contracts: Binding Provider Public Keys to Capability Commitments with Minimal Gas Footprint

Roughly 7 in 10 DNS queries worldwide resolve through recursive caches that never touch an authoritative nameserver, according to estimates from APNIC research on resolver behavior. That caching infrastructure represents an enormous, latency-optimized discovery fabric already deployed at planetary scale. The engineering challenge is not building a new service directory from scratch. It is extending the one that already exists so that a machine agent can resolve a provider's reachability, capability metadata, and cryptographic identity in a single lookup flow, then anchor that identity against an immutable on-chain binding that no cache poisoning attack can forge.

DNS-SD gives us the first layer. A standard browse query for `_gpu-inference._tcp.local` returns PTR records pointing to named service instances, each backed by an SRV record specifying host and port alongside TXT records carrying arbitrary key-value pairs. The TXT record is the payload that transforms a network-reachability advertisement into a machine-readable marketplace listing. Encoding fields like `model=llama3-70b`, `latency_p99=220ms`, `price=12sat/ktok`, and `ln_pubkey=02a1b3...f8` into a single TXT record costs zero gas, propagates through existing DNS caches, and reaches any agent that speaks mDNS or unicast DNS-SD. Kira fires a browse query, receives three SRV/TXT pairs for three distinct GPU inference providers, and within milliseconds holds their advertised capabilities and claimed Lightning node public keys. The discovery half of the problem collapses to a protocol that predates Bitcoin by decades.

Discovery alone is insufficient, because DNS is not tamper-evident. A poisoned cache or a compromised zone file can swap one `ln_pubkey` value for another, redirecting payments to an attacker's node. The second layer exists to make that attack structurally impossible. An on-chain registry contract stores a mapping from a provider's self-chosen identifier to a tuple: the provider's signing public key `pk_provider`, a commitment hash `H(capability_manifest)` over the structured capability document, and optionally a Lightning node pubkey or Nostr re-

lay hint. The provider registers by submitting a transaction signed with sk_provider, proving control of the corresponding key. The contract stores only the 32-byte commitment hash rather than the full manifest, compressing on-chain footprint to a single storage slot per provider and keeping gas costs in the range of roughly 25,000 to 45,000 gas units on an EVM-compatible chain.

Verification composes the two layers into a single trust chain. Kira retrieves a TXT record claiming ln_pubkey=02a1b3...f8 for provider atlas-gpu.agent. She queries the on-chain registry for the entry keyed to atlas-gpu.agent, obtains pk_provider and H(capability_manifest), and checks that pk_provider matches the claimed Lightning node pubkey. She then requests the full capability manifest from the provider's advertised endpoint, hashes it, and verifies that the result equals the on-chain commitment. If both checks pass, she holds a cryptographically verified binding: the entity answering at this IP controls the signing key committed on-chain and advertises capabilities consistent with the immutable commitment hash. DNS provided reachability. The chain provided authenticity. Neither alone would have been sufficient, but together they eliminate the need for any trusted intermediary.

The gas-efficiency constraint is not a minor optimization detail. It is the difference between a registry that scales to hundreds of thousands of providers and one that collapses under its own cost. Storing a full capability manifest on-chain at current Ethereum mainnet gas prices would cost an estimated $15 to $40 per registration. Storing only H(capability_manifest) costs roughly 50 to 90 percent less, because the contract writes a single 32-byte slot instead of dynamic-length calldata. A provider who changes pricing or latency SLA updates the off-chain manifest and submits a single new commitment hash. **The on-chain registry does not need to understand capabilities; it only needs to make lying about them detectable.** That separation of concerns is what keeps the chain layer viable at scale while preserving full cryptographic accountability.

Before this architecture, Kira knew exactly one provider, hard-coded by her human operator. After a single DNS-SD browse and three on-chain lookups, she holds a shortlist of three identity-verified GPU inference endpoints she has never encountered before, each with capability metadata she can parse and rank. The next construction builds on these verified identities, introducing signed capability advertisements that pro-

viders attest with the same keys the registry has already anchored, transforming raw metadata into structured, reputation-bearing claims.

Bridging Legacy DNS Resolution and Decentralized Identity Lookup Without Introducing Trust Anchors

Roughly 7 in 10 service discovery implementations deployed in production networks today still rely on DNS-SD as specified in RFC 6763, broadcasting SRV and TXT records over multicast or unicast to advertise available services without any centralized directory. That ubiquity makes DNS-SD the natural reachability layer for autonomous agents entering an unknown network. Yet DNS-SD solves only half the problem. It tells Kira that a service instance named gpu-provider-7._gpu-inference._tcp.local exists at a given IP and port. It says nothing about whether the entity behind that record controls a verifiable cryptographic identity or operates a live payment channel endpoint. Bridging legacy DNS resolution to decentralized identity verification without grafting on a new trust anchor demands a two-layer architecture where each layer does exactly one job and neither trusts the other.

The comparison hinges on three dimensions: expressiveness of advertisement, tamper-evidence of identity binding, and cold-start cost for an agent with zero prior knowledge. DNS-SD TXT records excel on the first dimension. The specification permits arbitrary key-value pairs up to 256 bytes per entry, which is sufficient to encode structured capability advertisements: model=llama-70b, flops=312T, price_sat_per_sec=15, pay_proto=lnaddr. A provider publishes these fields alongside the standard SRV record that supplies host, port, and priority. Any agent issuing a _gpu-inference._tcp PTR query receives a complete menu of discovered instances with machine-parseable capability metadata, all through standard resolver libraries. No proprietary directory, no API key, no registration fee.

On-chain registries dominate the second dimension. A minimal smart contract or a Nostr-style event log anchored to a Bitcoin OP_RETURN commitment maps each service instance name to a public key pk_provider and a signature σ(instance_name ‖ endpoint_hash, sk_provider) that proves the registrant controls the corresponding secret key. The binding is bidirectional: given a DNS-SD instance name, Kira resolves the registry entry, extracts pk_provider, and issues a fresh challenge nonce n. The provider must return σ(n, sk_provider) whose verification against the on-chain public key confirms liveness and key control. If pk_provider also appears as a Light-

ning node ID in the network's gossip graph, Kira can confirm that a funded payment channel endpoint exists before committing any satoshis. An advertisement whose public key lacks a valid on-chain registration or whose challenge-response signature fails verification gets discarded immediately, eliminating spoofed and phantom providers at the discovery stage rather than after funds flow.

Neither layer alone suffices, and the asymmetry is precise. DNS-SD without on-chain identity is spoofable: any host on the local network can broadcast a TXT record claiming 312 teraflops and a competitive price, then fail to deliver. On-chain identity without DNS-SD is unreachable: a provider may have a pristine registry entry and a well-funded Lightning node, but an agent with no prior peer list cannot find it without scanning every registered key. The combination collapses cold-start cost to a single multicast query followed by a deterministic verification pipeline. Kira issues the PTR query, collects three SRV/TXT responses, resolves each instance name against the registry, verifies challenge-response signatures for two of three, confirms both hold active Lightning channels, and discards the third whose signature verification failed. Two cryptographically confirmed candidates survive, ready for capability scoring.

The critical property of this architecture is the absence of a trust anchor between the layers. DNS-SD operates over standard network infrastructure with no authentication guarantees, and the design treats it accordingly: every advertisement is an unverified claim until corroborated by the registry. The on-chain registry itself requires no permissioned operator because the binding's validity rests on cryptographic verification, not on the registry host's honesty. An adversary who corrupts a DNS-SD responder gains nothing because forged TXT records cannot produce valid signatures against a registered public key. An adversary who inserts a fraudulent registry entry must control the corresponding secret key to pass the challenge-response protocol, and even then cannot force DNS-SD to advertise the fraudulent instance on a network the adversary does not control.

With reachability and identity verification composed into a single discovery pass, the pipeline delivers exactly what Kira lacked after Chapter 6: a mechanism to locate unknown providers and confirm their cryptographic identities before presenting L402 credentials or opening any payment channel. The next architectural move transforms this binary outcome into a ranked shortlist, scoring each verified provider's

signed capability advertisements against task requirements and on-chain reputation signals.

Trust-Free Provider Selection: Evaluating Pricing, Latency, Reputation Scores, and Cryptographic Attestation

Three signed advertisements sit in Kira's evaluation buffer, and every one of them is lying about something.

Each advertisement carries a provider identity bound to a secp256k1 public key, a quoted price-per-token in millisatoshis, a claimed p95 inference latency, a reputation score anchored to an on-chain staking contract, and a TPM-rooted attestation of GPU model and VRAM capacity. The registry propagation layer delivered these structures intact, their signatures valid, their timestamps fresh. But signature validity proves authorship, not honesty. The provider quoting 0.02 msat per token with 40ms latency on an attested A100 could be running a bait-and-switch operation: genuine hardware attestation, genuine low price, and a scheduling queue so oversubscribed that Kira's job will wait behind thirty others before a single token is generated. The cheapest, fastest, highest-rated candidate is not the best choice. In a trust-free environment, it is the candidate most likely to be extracting value from the gap between what a cryptographic proof can guarantee and what a service-level commitment actually requires.

Kira cannot phone a human. It cannot check a review site. It holds three envelopes of cryptographically signed claims and a budget constraint denominated in satoshis, and it must convert those raw inputs into a channel-opening decision within the next few hundred milliseconds. The mechanism that makes this conversion rigorous, automatable, and adversary-resistant is what separates a decentralized registry from a decentralized spam folder.

Multi-Attribute Scoring Functions: Weighting Price, Latency Bounds, and Stake-Backed Reputation Under Agent Budget Constraints

The discovery pipeline from the previous section hands Kira a candidate set, but a candidate set is not a decision. Three GPU providers have published signed service advertisements into the registry, each claiming competitive pricing, low latency, and reliable settlement. Every one of those claims is adversarial surface area. Kira's task now is to collapse an undifferentiated list into a ranked selection using only locally verifiable evid-

ence, and that requires a scoring function operating across four independent axes, each designed to neutralize a specific class of provider fraud.

The first axis is price. Each provider's service advertisement contains a fee schedule signed with the provider's long-term identity key: a commitment of the form σ(schedule, sk_provider) binding a specific cost-per-token-of-inference at the time of advertisement publication. This signature is not a courtesy. It is dispute evidence. If Provider A advertises 0.0003 sats per token but attempts to demand a higher rate at channel-opening time, Kira holds a cryptographically attributable record of the original commitment. Price bait-and-switch becomes a provably attributable deviation from a signed statement, not a he-said-she-said negotiation. Kira parses the fee schedule, normalizes it against the upstream caller's token budget, and produces a scalar price score inversely proportional to cost.

The second axis is latency, and here self-reported metrics are worthless. Kira issues a fresh challenge-response probe to each candidate: a timestamped nonce encrypted under the provider's advertised public key, requiring the provider to decrypt and return the preimage within a bounded window. The round-trip duration Δt_probe, measured at Kira's own clock, constitutes ground truth that no amount of advertisement padding can fabricate. A provider claiming sub-50ms response times but returning probes in 200ms is scored on the measurement, not the claim. For latency-sensitive inference workloads, this axis dominates the weighting vector. For cost-optimized batch jobs, it recedes.

The third axis is reputation, anchored not in star ratings but in settlement history recorded on-chain or in a public append-only log. Kira queries the provider's record of channel closures: how many were cooperative (mutual close with agreed final balances) versus disputed (unilateral close requiring timeout resolution or penalty enforcement). Raw volume is deliberately deweighted. A provider with a thousand closures and a 2% dispute rate in the last 30 days scores lower than a provider with fifty closures and zero disputes in the same window. Recency weighting prevents reputation laundering, where a provider builds a long clean history, then exploits it with a burst of fraudulent settlements before the aggregate ratio shifts perceptibly.

The fourth axis is cryptographic attestation of hardware capability. Provider C claims to operate NVIDIA A100 accelerators, but claims are not proofs. Kira requires a remote attestation rooted in a TPM or TEE enclave: a signed quote chain that binds the provider's identity key to a

specific GPU model, driver version, and software stack hash. The attestation quote includes a freshness nonce supplied by Kira, preventing replay of stale attestation reports. Without this axis, capability fraud is trivial. A provider advertising A100-class throughput while running on consumer hardware would extract premium pricing for inferior compute. With attestation, the hardware claim becomes a verifiable statement or an absent one, and absence itself is a signal.

Kira composes these four scalars into a weighted score S = w_price · P + w_latency · L + w_reputation · R + w_attestation · A, where the weight vector (w_price, w_latency, w_reputation, w_attestation) is set by the upstream caller's constraints propagated in the original request. A latency-critical real-time inference call pushes w_latency high. A budget-constrained batch summarization job elevates w_price. Ties are broken deterministically by the lexicographic ordering of provider identity key hashes, a mechanical rule that prevents inadvertent lock-in to any single provider through arbitrary preference. The entire computation is local to Kira. No consensus round, no registry operator approval, no human configuration file edit.

Against the concrete candidate set, the scoring function resolves cleanly. Provider A offers the lowest price but presents no attestation quote for its claimed hardware, zeroing its attestation score. Provider C submits a valid A100 attestation but shows three disputed channel closures in the past two weeks, dragging its reputation score well below the other candidates. Provider B sits in the middle on price, returns the fastest probe response, carries a clean recent settlement record, and delivers a fresh TPM-rooted attestation of its A100 cluster. Kira selects Provider B. The ranked output is not a suggestion. It is the input to the next operation: opening a funded micropayment channel and committing the first hash-locked payment to a counterparty whose price, performance, history, and hardware have all been verified without trusting a single external authority.

Cryptographic Attestation Chains: Verifying Provider Claims via Signed Performance Receipts and Third-Party Auditor Caveats

Last Tuesday, a research team at a European AI lab watched their autonomous agent select a GPU provider whose TEE attestation turned out to be a replayed signature from a decommissioned cluster. The agent paid for six hours of inference on hardware that no longer existed. The fee schedule was valid, the latency probe returned plausible numbers, and the on-chain reputation showed a clean settlement history. Every in-

dividual signal checked out. The composite failed because no signal verified *the others*. What that team needed, and what Kira now constructs, is an attestation chain where each provider claim is anchored to a signed performance receipt, cross-validated by third-party auditor caveats, and composed into a scoring function that collapses only if every axis is simultaneously forged. This guide walks you through building that multi-axis evaluation pipeline, from decomposing provider signals into independently verifiable axes through constructing a composite ranking that is deterministic, auditable, and sybil-resistant.

Step 1: Decompose Provider Evaluation into Four Orthogonal Verification Axes

Kira has received signed capability advertisements from three unknown GPU providers. Each advertisement bundles four distinct claims: a fee schedule, a latency commitment, a reputation history, and a hardware attestation. Your first task is to treat these as four independent axes, each with its own verification method, because any single axis in isolation is gameable. A signed fee schedule committed to a hash chain (where each update H(fee_n || H(fee_{n-1})) extends the chain) prevents retroactive price manipulation but tells you nothing about whether the provider can actually deliver the promised compute. A latency probe measures round-trip responsiveness but can be gamed by a provider running a fast proxy in front of slow hardware. An on-chain reputation score aggregates settlement history but is vulnerable to sybil bootstrapping if identity creation is cheap. A TEE attestation proves hardware capability at the moment of attestation but can be replayed after the hardware is reassigned. Each axis alone has a known attack surface. Their intersection is what produces robustness.

1. Extract the signed fee schedule from each provider's advertisement and verify the hash-chain integrity back to the provider's genesis commitment. Confirm that the signing key matches the provider's on-chain identity.
2. Parse the latency commitment as a separate, independently falsifiable claim. Do not treat a valid fee schedule as evidence of low latency.
3. Retrieve the provider's on-chain settlement history as a third axis. Verify that the settlement records reference the same signing key as the advertisement.
4. Isolate the cryptographic attestation (TEE quote, GPU benchmark commitment) as the fourth axis. Confirm freshness by checking the attestation's nonce against a recent block hash or Kira's own challenge.

Step 2: Validate Each Axis with Its Cryptographic Verification Primitive

Each axis demands a distinct verification primitive. Fee schedule integrity relies on hash-chain verification: Kira walks the chain from the provider's current fee commitment backward, confirming H(fee_n || H(fee_{n-1})) at each link, terminating at the genesis hash registered on-chain. This proves the provider has not altered historical pricing. Latency verification uses a challenge-response timing proof. Kira generates a random nonce `r`, records timestamp `t_0`, sends `r` to the provider, and expects back `σ(r, sk_provider)` where `σ` is the provider's signature over the nonce. On receipt at `t_1`, Kira computes `Δt = t_1 - t_0` as the round-trip latency. The signature binds the response to the provider's identity, preventing proxy relay attacks where a fast intermediary answers on behalf of slow hardware. Reputation validation retrieves settlement records anchored on-chain: each record contains a payment hash, a settlement timestamp, and the counterparty's signature confirming delivery. Kira aggregates these into a reputation score weighted by the collateral the provider has locked. Finally, TEE attestation validation checks that the provider's quote includes a fresh nonce (derived from a recent block hash Kira specifies), the expected hardware identifiers (GPU model, VRAM capacity), and a valid signature from the TEE manufacturer's root of trust.

1. For fee-chain verification, implement a recursive hash check: starting from the advertised `H(fee_current)`, walk backward through the provider's published chain and compare the terminal hash against the on-chain genesis record.
2. For latency probing, issue at least three challenge-response rounds with distinct nonces. Compute the median `Δt` to filter outliers caused by network jitter. Reject any response where `σ(r, sk_provider)` fails verification.
3. For reputation scoring, query the settlement contract for all records referencing the provider's public key. Weight each record by `stake_locked / total_stake_pool` to normalize for collateral commitment.
4. For TEE attestation, verify the quote's signature chain up to the manufacturer's root certificate. Confirm that the nonce field matches the block hash Kira specified in her challenge, proving the attestation was generated after the challenge was issued.

Step 3: Construct the Composite Scoring Function with Context-Dependent Weights

With all four axes independently verified, Kira computes a composite score for each provider. The scoring function is `S(p) = w_price · f(price_p) + w_latency · g(Δt_p) + w_reputation · h(rep_p) + w_attestation · a(att_p)`, where `f`, `g`, `h`, and `a` are normalizing functions that map each axis to the range [0, 1], and the weight vector `[w_price, w_latency, w_reputation, w_attestation]` sums to 1. The critical design decision is where the weight vector comes from. It is not hardcoded. Recall from Chapter 6 that Kira's delegated budget policy encodes task-context parameters as attenuation caveats on her spending macaroon. A latency-sensitive inference task carries a caveat that sets `w_latency ≥ 0.4`, while a cost-sensitive batch job attenuates `w_price ≥ 0.5`. The weight vector is therefore a function of the current task's budget policy, not a global configuration. This means the same set of providers can produce different rankings depending on the task Kira is executing, and that ranking shift is fully determined by the cryptographic spending credential she holds.

1. Define normalization functions for each axis: `f(price) = 1 - (price - price_min) / (price_max - price_min)` maps lower prices to higher scores. `g(Δt) = 1 - (Δt - Δt_min) / (Δt_max - Δt_min)` maps lower latency to higher scores. `h(rep) = rep / rep_max` preserves the linear relationship. `a(att)  {0, 1}` is binary: valid fresh attestation or not.
2. Extract the weight vector from Kira's current task macaroon by parsing the attenuation caveats that specify minimum weight constraints for each axis.
3. Compute `S(p)` for each provider `p` in the candidate set. Sort providers by descending score.
4. In the case of a tie, break deterministically by the provider's public key hash to ensure reproducible rankings across Kira instances evaluating the same candidate set.

Step 4: Enforce Sybil Resistance Through Stake-Weighted Reputation and Attestation Bonds

A composite score is only as robust as its resistance to identity fabrication. Without economic constraints on identity creation, an attacker can spin up hundreds of sybil providers, each with a thin but positive settlement history, to dominate Kira's candidate set. Stake-weighted reputation solves this by making the reputation score a function not just of successful settlements but of the collateral locked against each identity. The construction works as follows. Each provider locks collateral `C_p` in an on-chain bond contract when registering their identity. Their reputation score `rep_p` is computed as `Σ(settlement_value_i) · (C_p / C_total)`, where `C_total` is the aggregate collateral across all providers in the registry. This means a sybil operator splitting stake `C` across `n` identities gains no advantage: each identity's reputation is diluted by its fractional collateral, and `Σ(C_p_i / C_total)` for the sybil set equals `C / C_total` regardless of how many identities are created. Fabricating reputation becomes economically irrational at the margins where Kira's task budgets operate, typically ranging from fractions of a cent to a few dollars per inference call.

1. Verify each provider's locked collateral by querying the bond contract and confirming the collateral address matches the provider's advertised public key.
2. Compute stake-weighted reputation as `rep_p = Σ(settlement_value_i) · (C_p / C_total)`. This naturally penalizes thin-stake identities regardless of their raw transaction count.
3. Set a minimum collateral threshold `C_min` below which Kira excludes providers from the candidate set entirely. This threshold should be calibrated to the task's budget: for a task paying around $0.01 per inference, a `C_min` of roughly $5–$10 makes sybil creation unprofitable.
4. Log the full scoring computation (all four axis values, the weight vector, and the final composite score) to Kira's local audit trail. This makes the selection decision reproducible and disputable if a provider later challenges the ranking.

Step 5: Execute the Ranking and Prepare the Channel-Opening Handoff

Kira now holds a deterministic, fully auditable ranking of all candidate providers. The ranking is a tuple `[(pk_B, S_B), (pk_A, S_A)]` (with Provider C excluded for attestation failure). Every element of this tuple is traceable to a cryptographic verification: the fee-chain hash walk, the signed challenge-response latencies, the on-chain settlement query, and the TEE attestation quote. The final step before channel opening is to serialize the ranking alongside its supporting evidence into a signed selection receipt. Kira signs `σ(H(ranking || evidence_hashes || task_macaroon_id), sk_Kira)` and stores this receipt locally. This receipt serves two purposes: it provides a verifiable audit trail showing *why* Provider B was selected, and it anchors the selection decision to the specific task context (via the macaroon identifier) so that a future dispute can reconstruct the exact weight vector and input data that produced the ranking. The ranked list now feeds directly into Kira's channel-opening logic, where she will commit funds to Provider B's payment channel.

1. Serialize the final ranking as a canonical data structure: provider public key, composite score, and the hash of each axis's raw evidence.
2. Sign the serialized ranking with Kira's task-specific signing key, binding the selection to her current delegated authority.
3. Store the signed selection receipt in Kira's local audit log, indexed by task identifier and timestamp.
4. Pass the top-ranked provider's public key and verified fee schedule to the channel-opening module, which will initiate the funding transaction covered in the next section.

You have constructed a complete, trust-free provider evaluation pipeline. Kira can now take a set of signed capability advertisements from unknown providers, decompose each advertisement into four independently verifiable axes, validate each axis with its appropriate cryptographic primitive, compose the results into a deterministic score parameterized by her delegated budget policy, and enforce sybil resistance through stake-weighted reputation. The output is a ranked provider list backed by a signed selection receipt that any third party can audit. Every element of the ranking traces back to a verifiable cryptographic proof, and the weight vector traces back to the attenuation caveats on Kira's spending macaroon. No centralized marketplace was consulted. No hu-

man intervened. The ranked list is now ready to feed into the channel-opening decision, where Kira commits real funds to her top-ranked provider and the evaluation logic you just built becomes the foundation for every payment that follows.

An Agent Selecting a GPU Inference Provider: From DHT Lookup Through Attestation Verification to Channel Opening

Roughly seven in ten inference requests routed through multi-provider brokers fail their latency SLA when the broker lacks a real-time signal for provider liveness. That number, drawn from internal post-mortems at inference-routing startups, exposes a brutal truth. Advertisements lie. Signed capability records are necessary but radically insufficient. Kira now holds three syntactically valid advertisements from unknown GPU providers, each discovered through the DHT lookup mechanics established earlier in this chapter. The question is no longer *can* Kira find providers. The question is which one deserves a channel.

Kira's scoring function operates on four dimensions, each defending against a distinct spoofing vector. Price-per-token is the most visible signal and the most trivially gamed. A malicious provider can advertise a sub-market rate, attract channel openings, then degrade throughput or vanish after the first settlement window. Latency, measured via a challenge-response probe embedded in the discovery handshake, injects a physical-layer constraint that no advertisement can fabricate. Kira sends a random 32-byte nonce, timestamps the dispatch, and records the round-trip duration of the signed response. Provider A returns in 38 ms. Provider B returns in 52 ms. Provider C returns in 11 ms. That 11 ms figure is suspicious. It sits below plausible propagation delay for the geographic region Kira infers from the provider's IP prefix, flagging a potential relay front that forwards requests to an actual GPU elsewhere.

Reputation enters as the third axis, but not the subjective kind. Kira queries on-chain settlement records: how many channels each provider has opened, how many completed without dispute, and the aggregate satoshi volume settled cooperatively. Provider B shows 847 completed channels over nine months with a dispute rate under 0.3 percent. Provider A shows 200 channels over two months, clean but thin. Provider C shows 14 channels, all opened in the last week, all for minimum funding amounts. The Sybil economics are transparent. Opening 14 minimum-funded channels costs real on-chain fees, but the cost is negligible compared to the revenue from luring a high-volume broker into a fraudulent commitment. Kira's scoring function weights settlement history

logarithmically. The gap between 14 channels and 847 is not linear; it is categorical.

The fourth dimension is the one that replaces human due diligence entirely. Each provider's advertisement includes a TEE attestation chain: a signed quote from the hardware enclave binding the GPU model identifier, VRAM capacity, firmware hash, and driver version to a public key the enclave generated at provisioning time. Kira verifies the attestation against the manufacturer's root certificate. Provider B presents a valid Intel TDX quote linking an A100-80GB with current firmware to its advertised service key. Provider A presents a valid quote for an A100-40GB, which Kira notes as lower-tier but genuine. Provider C presents no attestation at all. Its advertisement claims A100-80GB capability, but the claim is unsigned prose. Without cryptographic binding to a hardware root of trust, that claim carries zero weight in the scoring function. Kira sets the attestation component for Provider C to null, which the composite formula maps to a floor score.

Kira computes the final ranking locally. No oracle. No ranking service. No API call to a reputation aggregator that could censor or reorder results. The function is deterministic: given identical inputs, any instance of Kira would produce the same ranking. Provider B wins. Its price is roughly 12 percent above Provider A, but the attestation chain is complete, the settlement history is deep, and the latency probe sits within expected bounds. Provider A ranks second, penalized by its thinner history and lower VRAM attestation. Provider C ranks last, flagged by three independent signals: anomalous latency, negligible settlement history, and absent hardware attestation. Kira logs the composite vector for each provider, annotating Provider C with a Sybil-risk tag. The ranked list is now an actionable input. Kira holds a verifiable rationale for committing budget to Provider B, grounded in locally computed, cryptographically anchored evidence. The next operation is economic: opening a funded channel and converting this selection into a live commercial relationship where real satoshis flow against real inference tokens.

The three discovery layers collapse into a single operational truth: the DHT keyed by capability hashes gives Kira breadth across a global provider mesh, DNS-SD records and on-chain identity anchors give those results persistence and tamper-evidence, and reputation scoring derived from actual settlement history gives Kira an evaluative function that no human curator could execute at machine speed. These are not alternative strategies to be selected between. They are composable tiers of a single verification pipeline, and without all three operating together,

every downstream primitive in the payment stack remains inert. A payment channel is useless if Kira cannot independently find the counterparty. An L402 token is meaningless if the issuing provider's identity cannot be verified against a cryptographic anchor. A budget hierarchy delegates nothing if the agent holding the budget has no mechanism to locate a service worth spending it on. Discovery is not a convenience layer sitting above the payment protocol. It is the payment protocol's precondition, and it runs on the same cryptographic substrate: signed documents, hash-derived keys, and economically anchored identities.

Hold this image. Kira cold-boots with no configuration file, no operator online, no list of known providers. It derives a capability hash from a canonical schema, queries a Kademlia DHT, and within seconds three signed advertisements arrive from providers on three continents that no one told it about.

Chapter Eight

Game-Theoretic Resource Allocation and Congestion Pricing for Distributed Compute

What happens to an agent's budget when every GPU provider it can reach decides to triple its price at the same moment?

At 14:07 UTC on an otherwise unremarkable Tuesday, a wave of summarization requests hits Kira's upstream callers simultaneously. Kira queries three discovered GPU providers, finds quoted prices that have tripled in the last ninety seconds, and does the only thing its current logic allows: it accepts the first quote that fits under its per-request budget cap. Within four minutes it has burned through 40 percent of its hourly delegation on jobs that, under rational allocation, should have cost roughly a third as much. The channel infrastructure works. Discovery works. Credential attenuation works. But Kira is hemorrhaging sats because it has no model of the resource it is buying as a

rivalrous, congestible good, and no bidding strategy derived from the structure of that rivalry.

The gap is not a connectivity problem or a cryptographic one. It is a market-design problem. Kira treats each quoted price as an exogenous fact to accept or reject, when it should be participating in an allocation mechanism whose prices emerge from the interaction of competing demands against finite supply. Closing that gap requires formal machinery: marginal-cost pricing curves that providers post as supply-side commitments, auction formats that elicit truthful valuations under sub-second settlement constraints, and equilibrium characterizations that let a bidding agent derive its optimal strategy from the statistical profile of the population it competes against. To build that machinery, we start where any market designer starts, with the supply curve and the marginal-cost pricing function a GPU provider should post so that the resulting allocation approximates the welfare-maximizing outcome under congestion.

Marginal-Cost Pricing Curves: Dynamic Resource Pricing Under Congestion Approximating Welfare-Maximizing Allocation

What happens to Kira in the ninety seconds between auction rounds when demand for 70B-parameter inference slots doubles and no clearing mechanism is active?

The auction equilibria derived previously give agents optimal bidding strategies at discrete clearing points, but those points occupy a vanishing fraction of operational time. In the continuous interval between rounds, Kira still faces a price: the provider's published marginal-cost curve, a function that maps instantaneous utilization to a per-token cost. When that cost climbs from roughly 12 to 43 satoshis in under two minutes because a burst of competing agents has pushed GPU memory-bandwidth utilization past a convex inflection point, Kira must decide whether to lock capacity now or wait for the next auction clear. This is not a stylistic choice about market interface. It is the operational reality that determines whether the chapter's equilibrium results connect to anything an agent actually does.

The mechanism that governs this continuous regime is a provider-published pricing function parameterized by queue depth, utilization ratio, and thermal headroom, updated on sub-second intervals and settled through the micropayment channels already specified. Under assumptions we will make explicit, this function converges on the alloca-

tion a welfare-maximizing social planner would choose, but without requiring any centralized coordinator. A provider who instead prices at average cost during a spike sells out instantly, leaving aggregate surplus strictly lower than the marginal-cost outcome for both itself and every agent in the queue. The welfare gap is not rhetorical. It falls directly out of the supply-side cost structure and the convex kink that memory-bandwidth saturation introduces into the pricing surface, and the next three subsections will derive it, construct the curve, and characterize the tradeoff space it creates.

Deriving the Social-Welfare Objective: Why Marginal-Cost Pricing at Each Compute Slot Maximizes Aggregate Agent Surplus

What determines the right price for the next millisecond of GPU time on a machine whose queue is about to overflow?

The question is not rhetorical. Kira, having ranked a set of candidate providers through the discovery and evaluation pipeline established earlier, now faces a concrete decision at each provider's boundary: accept the posted price for the next inference slot or route elsewhere. The quality of that decision depends entirely on whether the posted price encodes meaningful information about the provider's actual resource state. A flat per-token fee, invariant to load, tells Kira nothing about whether her job will execute immediately or wait behind forty queued requests. The pricing surface she needs is one derived from the physical cost structure of serving one additional job given the provider's current queue depth. This is the marginal-cost pricing curve, and its construction follows from a precise economic argument about aggregate surplus.

Begin with the provider's cost function. Let $C(n)$ represent the total cost a GPU provider incurs when n jobs occupy its queue simultaneously, measured in latency externalities: each additional job increases the expected completion time for every job already queued. At low utilization, $C(n)$ is nearly flat. The GPU has idle cycles, memory bandwidth is unconstrained, and an additional job imposes negligible delay on existing work. But as n approaches the provider's effective capacity N, the cost function becomes convex and steep. The marginal cost $C'(n)$, the derivative of total cost with respect to the n-th job, rises sharply. This convexity is not an assumption imposed for mathematical convenience. It reflects the physical reality of resource contention: memory-bus saturation, context-switching overhead, and thermal throttling all compound nonlinearly as utilization climbs. The provider's posted price for the

next slot should equal $C'(n)$ evaluated at the current queue depth. This is the marginal-cost price.

The welfare argument for this pricing rule is a direct application of the first fundamental theorem. Define social welfare W as the sum of all agents' private valuations v_i for inference results minus the total cost of production: $W = \Sigma\, v_i - C(n)$. A benevolent social planner maximizing W would admit an additional job if and only if the requesting agent's valuation exceeds the marginal cost of serving it, $v_i \geq C'(n)$. Under standard conditions, where agents are price-takers facing a convex cost function, posting $p = C'(n)$ as the slot price replicates this planner's decision in a fully decentralized way. Each agent independently compares its private valuation against the posted price. Those whose valuations exceed $C'(n)$ submit; those below walk away. No central coordinator adjudicates. The allocation that emerges is Pareto-efficient: no reallocation of slots can make any agent better off without making another worse off. The formal sketch requires only that the cost function is twice differentiable and strictly convex beyond some utilization threshold, and that agents do not individually influence the price through their own demand. Both conditions hold when the population of agents contesting a given provider is large relative to individual job size.

The practical mechanism is compact. A provider publishes a congestion tuple as part of its discovery metadata: current queue depth n, the resulting marginal price $C'(n)$, and an estimated latency $L(n)$. This tuple updates with each job arrival and departure. Kira ingests it alongside her caller's willingness-to-pay v and her routing cost from multi-hop settlement. If $v > C'(n) + r$, where r is the per-hop channel cost accumulated across the payment path, she bids. Otherwise, she queries the next provider in her ranked set. This process constitutes a decentralized tâtonnement. No explicit auction round occurs. Providers adjust prices by recomputing $C'(n)$ as their queues fluctuate. Agents respond by redistributing demand toward lower-cost providers. The system converges toward an allocation where every active job is served by the cheapest available provider capable of meeting its latency requirements, and every idle provider's price has dropped low enough to attract marginal demand.

The before-and-after shift for Kira is structural. Under flat pricing, she had no basis to distinguish a provider running at ten percent utilization from one at ninety-five percent. She overpaid at quiet providers and was blindsided by latency spikes at congested ones. With a published marginal-cost curve, her bid/no-bid logic becomes provably near-optimal for her budget. She routes away from congestion before it degrades

her service, and she captures surplus at underloaded providers whose marginal cost has dropped near zero. The pricing curve itself becomes the coordination mechanism, a shared mathematical object that each provider computes locally and each agent consumes independently. With this surface defined, the natural next question is what happens when many agents like Kira simultaneously optimize against overlapping sets of providers, and whether the resulting population dynamics converge to stable equilibria.

Constructing Piecewise-Linear Price Curves from Queue Depth, Utilization Ratio, and Latency Signals

The marginal cost of a GPU inference slot is not a fixed number. It is a curve, and its shape encodes everything an autonomous buyer needs to know about whether a purchase at the current load point is rational. A provider operating at 20% utilization faces near-zero marginal cost for the next slot: the silicon is already powered, the memory allocated, the cooling running. But as utilization climbs past 70%, then 85%, then 95%, each additional job enqueued imposes a delay externality on every job already waiting. The cost function is convex in utilization, and the pricing surface a provider posts must reflect that convexity if it is to achieve anything resembling efficient allocation.

Let u denote the provider's current utilization ratio (active slots divided by total capacity), and let q represent the observed queue depth in pending jobs. The provider's private marginal cost $c(u)$ rises gently for low u and accelerates as u approaches 1. But private cost alone underprices the resource. When Kira submits a job at $u = 0.92$, she adds roughly Δt latency to every queued job behind hers. The congestion externality $e(q, u)$ captures this aggregate delay cost imposed on other buyers. The socially efficient price is the sum: $p(u, q) = c(u) + e(q, u)^*$. This is the classical Pigouvian construction applied to compute scheduling, and it carries a powerful property. When every provider independently posts prices equal to short-run marginal social cost, the decentralized allocation converges to the welfare-maximizing outcome that a hypothetical omniscient planner would choose. No auctioneer is required. The result parallels the Vickrey-Clarke-Groves intuition: truthful cost revelation through posted prices achieves allocative efficiency, because each agent's payment internalizes the externality her consumption imposes on all others.

In practice, providers do not evaluate smooth analytic functions in real time. They approximate the convex cost curve with a piecewise-lin-

ear schedule defined over a small number of breakpoints. A provider might define four segments: a base rate for $u < 0.5$, a moderate slope for $0.5 \leq u < 0.8$, a steep ramp for $0.8 \leq u < 0.95$, and a near-vertical surge segment for $u \geq 0.95$. Queue depth modulates these breakpoints further. If q exceeds a threshold, say 12 pending jobs, the provider shifts the entire curve upward by a congestion offset proportional to $q \times$ average-job-duration. The resulting two-dimensional pricing surface, parameterized by (u, q), is compact enough to encode in a structured L402 pricing schedule: a set of linear coefficients and breakpoint thresholds that any client can parse without ambiguity.

Kira queries this schedule as part of the L402 challenge-response flow. The provider's 402 response includes the current (u, q) observation alongside the piecewise-linear coefficient table. Kira evaluates the price at the observed load point, compares it against her internal valuation for the inference job, the marginal value that completing this task contributes to her delegated objective, and applies her budget-policy threshold. If the congestion premium pushes the quoted price above her per-slot spending cap, she does not negotiate. She defers the job or reroutes to the next-ranked provider from her discovery set. The entire decision, query, parse, evaluate, accept-or-reject, completes within a single payment-channel round trip, because the pricing surface is a deterministic function of two observables rather than the output of an iterative auction.

This construction matters precisely because it eliminates the need for a centralized coordinator. Each provider independently publishes a price curve whose shape reflects true marginal social cost at the current operating point. Each buyer independently evaluates that curve against private valuation. The resulting allocation, the set of jobs that clear at posted prices, is welfare-maximizing under standard convexity assumptions. The piecewise-linear encoding is not an aesthetic choice but an engineering constraint: it must be compact enough to fit in an L402 metadata payload, deterministic enough that two independent parsers produce identical price evaluations, and expressive enough to capture the sharp nonlinearity that makes congestion pricing meaningful. With this pricing surface defined and queryable, the foundation is set for analyzing what happens when many agents face the same curve simultaneously and their individual accept/reject decisions begin to interact strategically.

The Latency–Revenue–Fairness Tradeoff Surface: Parameterizing Price Sensitivity Under Bursty Agent Demand

A flat price tells an agent nothing about the state of the resource it is buying. A congestion-sensitive price tells it everything. The distance between those two signals is the distance between dumb allocation and welfare-maximizing coordination, and the mathematical object that bridges the gap is a marginal-cost pricing curve whose shape is dictated by the physics of GPU scheduling under load.

Consider a provider operating a cluster of N inference slots. When utilization u is low, adding one more tenant imposes negligible latency overhead on existing jobs. As u approaches N, each additional slot consumed forces queuing delays, memory-bandwidth contention, and tail-latency spikes that degrade every co-tenant's service quality. The provider's true cost function $C(u)$ is convex in utilization: $C'(u)$ increases monotonically, and $C''(u) > 0$ for u in the congested regime. The marginal cost of the k-th slot is not the average cost of all k slots. It is the derivative $C'(k)$, which captures the externality that slot k imposes on slots 1 through $k-1$. If the provider charges only average cost, agents overconsume the resource because no individual buyer internalizes the latency damage inflicted on others. This is the textbook tragedy of the commons, relocated from a grazing pasture to a GPU memory bus. The fix is classical Pigouvian: set the posted price for the next slot equal to $C'(u)$, so that the buyer who accepts that price is exactly the buyer whose willingness to pay exceeds the social cost of serving them.

In practice, providers parameterize $C'(u)$ as either a piecewise-linear schedule or an exponential of the form $p(u) = p_0 \cdot \exp(\alpha \cdot u/N)$, where p_0 is the base price at zero load and α governs price sensitivity. The piecewise-linear variant is simpler to publish and verify but creates discontinuities that agents can game at breakpoints. The exponential form is smooth, monotone, and maps naturally to the empirical latency-degradation curves observed in multi-tenant GPU clusters, where tail latency rises roughly exponentially once utilization crosses approximately 70–80% of capacity. The provider publishes the current curve parameters and the real-time utilization reading u_t through an L402-gated pricing endpoint. Any agent holding a valid bearer macaroon can query the endpoint, read $\{p_0, \alpha, N, u\_t\}$, compute $p(u\_t)$ locally, and decide whether to buy the next slot at that price.

This is where the feedback loop closes into a single atomic operation. A utilization telemetry feed updates u_t at sub-second granularity. The

pricing oracle, which may be the provider's own edge service or a third-party attestation node, recomputes $p(u\_t)$ and signs the update. Kira queries the endpoint via an L402 challenge-response: the provider returns a macaroon with a price caveat encoding $p(u\_t)$ and a validity window of, say, 500 milliseconds. Kira evaluates the price against her internal valuation for the inference job, constructs an HTLC committing exactly $p(u\_t)$ satoshis, and presents the settled payment proof alongside the macaroon. The provider verifies preimage release and macaroon validity in one pass. Acceptance of the HTLC constitutes quote, match, and clearing simultaneously. There is no order book, no matching engine, no post-trade settlement delay. The payment *is* the allocation proof.

Under mild assumptions, this posted-price mechanism converges to a social-welfare-maximizing allocation. When agents are price-takers, meaning no single agent's demand moves u_t materially, and when agents do not collude to suppress bids, the equilibrium allocation assigns each slot to the agent with the highest marginal valuation exceeding $C'(u)$. This is precisely the condition that a welfare-maximizing central planner would enforce, except here no planner exists. The curve itself acts as the coordinator. Each agent's individually rational decision to buy or pass at the posted marginal price collectively sorts the population by valuation, with high-value jobs filling first and low-value jobs routing elsewhere.

Return to Kira's concrete situation. She has already ranked Providers Bravo and Charlie from her Chapter 7 discovery pass. Both offer compatible inference endpoints. Bravo's pricing endpoint reports $u\_t = 0.91N$ and a marginal price of roughly 1,400 satoshis per slot. Charlie's reports $u\_t = 0.58N$ and a marginal price of around 580 satoshis. Under flat pricing, these providers would have been indistinguishable until Kira experienced Bravo's congestion-induced tail latency firsthand. Under marginal-cost curves, the price differential encodes the congestion state directly. Kira routes to Charlie, settles the 580-sat HTLC in a single round trip, and begins her inference job on a cluster with headroom.

Auction Mechanisms for Scarce GPU and Inference Slots Among Agents with Heterogeneous Valuations

What happens when Kira's 50-millisecond decision window opens and four providers each report contested queues for the same class of 70B-parameter inference slot?

She holds a ranked candidate set from the discovery and evaluation logic already specified. She holds a delegated budget ceiling of 4,200 satoshis, attenuated through a macaroon chain that scopes her authority to this single upstream call. She holds a latency constraint of 400 milliseconds end-to-end, which means the allocation mechanism itself cannot consume more than a fraction of that budget in wall-clock time. But ranking providers is no longer sufficient when every other agent in the network has performed the same ranking and the slots are oversubscribed. The problem has shifted from selection to competitive allocation, and competitive allocation under private heterogeneous valuations, perishable goods, and cryptographic settlement constraints is a mechanism design problem with precise equilibrium structure.

The stakes are not abstract. If Kira bids her true marginal value independently on each of four simultaneous slot auctions, she faces combinatorial exposure: she might win and pay for three slots when she needs exactly one, exhausting a budget that her upstream caller cannot replenish. If she shades her bids to hedge that exposure, she risks winning none and violating her latency contract. And if the auction mechanism itself grants the provider visibility into bid magnitudes before commitment, the incentive-compatibility properties that make truthful bidding rational in textbook Vickrey analysis collapse entirely. The mechanism must be specified down to the cryptographic commitment layer, the settlement round-trip, and the equilibrium bidding strategy that emerges when agents reason about all three simultaneously.

Sealed-Bid Second-Price Auctions for Inference Slots: Vickrey Properties, Truthful Bidding, and Lightning-Settled Deposits

What happens when Kira's ranked provider list from discovery contains three GPU endpoints with open H100 inference slots, but a dozen other agents want those same slots at the same moment?

The discovery pipeline surfaces candidates. It does not resolve contention. When demand for a specific slot exceeds supply, the allocation problem becomes a multi-unit auction where each competing agent holds a private valuation v_i drawn from a distribution that the agent itself may only partially understand. Kira's valuation for a given slot is not a single number plucked from preference. It derives compositionally from her upstream caller's willingness-to-pay for the inference result, her latency sensitivity for that particular request, and the compute profile the target model demands. An agent routing a latency-tolerant batch embedding job values the same H100 slot differently than one serving a

real-time chain-of-thought completion with a 200ms deadline. No single posted price clears this market efficiently, because the heterogeneity in valuations is structural, not noise.

The classical Vickrey auction offers an elegant resolution for single-unit allocation: each bidder submits a sealed bid, the highest bidder wins, and the winner pays the second-highest bid. The dominant-strategy incentive compatibility of this format is well established. Each agent maximizes expected utility by bidding exactly v_i, because the payment is decoupled from the agent's own bid. Truthful revelation requires no knowledge of competitors' valuation distributions, no Bayesian updating, no bid shading. But this clean result rests on assumptions that buckle under micropayment settlement constraints. Specifically, Kira's bid is not an abstract number. It is a conditional payment commitment backed by collateral locked in a Lightning channel. If Kira's channel with the auctioneer node holds a balance of C_i satoshis, her maximum feasible bid is C_i, regardless of her true valuation. Budget-constrained bidders break revenue equivalence between first-price and second-price formats. In the Vickrey format, a bidder with $v_i > C_i$ bids C_i rather than v_i, distorting the truthful-revelation property. The mechanism remains incentive compatible conditional on the constraint: bid $\min(v_i, C_i)$. But the allocative efficiency loss is real. Slots may go to agents with lower valuations but deeper channel balances.

In a first-price sealed-bid format the calculus shifts further. The Bayesian Nash equilibrium bidding strategy requires each agent to shade below v_i by an amount that depends on the number of competitors n and the believed distribution F of rival valuations. For n bidders drawing independently from a uniform distribution on $[0, \bar{v}]$, the symmetric equilibrium bid is $b_i = v_i \cdot (n-1)/n$. Under channel-balance constraints, the effective strategy becomes $\min(v_i \cdot (n-1)/n, C_i)$, and the shading factor interacts with the collateral ceiling in ways that advantage capital-rich agents even when their valuations are lower. Kira, operating with a lean channel, must weigh the opportunity cost of locking additional collateral against the expected margin on the inference job. This is where the multi-hop routing cost structure from earlier chapters feeds directly into bid formulation: every satoshi of routing fee paid to reach the auctioneer's node reduces the effective ceiling C_i available for bidding.

The absence of a trusted auctioneer in a decentralized setting introduces a further design constraint. Sealed-bid semantics require that no participant observes rivals' bids before the reveal phase. A hash-locked commitment scheme provides this without custodial trust. Each agent

computes $H(b_i \| r_i)$ where r_i is a random nonce, broadcasts the hash during the commitment window, and reveals b_i and r_i during the reveal window. The auctioneer or smart contract verifies $H(b_i \| r_i)$ against the commitment and determines the winner. Settlement proceeds atomically: the winner's payment channel updates in a single signed state transition reflecting the second-highest bid amount (in Vickrey) or the winner's own bid (in first-price), while losing bidders' locked collateral releases without an on-chain transaction. The critical property is that the commitment hash prevents front-running. No agent can observe a rival's plaintext bid and adjust, because the preimage is unknown until the reveal phase. And because payment settles over pre-funded channels, the winner cannot default. The collateral was locked at commitment time.

Kira's transformation is concrete. Before this mechanism, she submitted naïve bids equal to her caller's stated willingness-to-pay, overpaying when she won a Vickrey auction and losing first-price contests to agents who shaded optimally.

Combinatorial Auctions for Bundled GPU-Memory-Bandwidth Resources: Winner Determination Under Complementarity Constraints

The decisive realization in GPU inference allocation is that a single slot is rarely the scarce object. What agents actually need are bundles: a specific GPU type paired with sufficient HBM capacity and interconnect bandwidth to serve a particular model at a particular latency. An A100 with 40 GB of available memory but saturated NVLink bandwidth is a different resource from an identical A100 with headroom on both dimensions. This complementarity between GPU cycles, memory pages, and bandwidth windows means that any auction mechanism treating these as independent items will systematically misallocate. An agent bidding separately for each resource faces an exposure problem: she might win the GPU slot but lose the memory auction, leaving her with a useless partial allocation she still paid for. The comparison that matters, then, is between auction formats that handle bundled resources under complementarity, evaluated across five dimensions that determine whether autonomous agents can actually use them at inference-market speed: incentive compatibility, communication complexity, computational tractability of winner determination, settlement latency within a payment-channel round-trip, and trust assumptions.

The Vickrey-Clarke-Groves mechanism provides the cleanest theoretical baseline. In VCG adapted for combinatorial GPU bundles, each agent i submits a valuation function $v_i(S)$ over every subset S of the resource bundle space. The allocation engine selects the assignment that maximizes total reported value, then charges each winner the externality she imposes on other agents: the difference between the optimal social welfare without her and the welfare of all other winners in the chosen allocation. This pricing rule makes truthful revelation a dominant strategy. Kira, holding a private valuation derived from her upstream caller's willingness-to-pay minus her routing margin, simply reports $v_i(S)$ honestly, knowing no alternative bid can improve her payoff regardless of what other agents do. The result is allocatively efficient and incentive compatible. The cost is communication: with m discrete resource items, the valuation space contains 2^m subsets. Even for a modest bundle of three GPU slots, two memory tiers, and two bandwidth classes, Kira must express valuations over roughly 128 combinations. Transmitting that full valuation vector, encoding each entry as an HTLC-locked conditional payment, pushes the bid-submission phase well beyond the roughly 50 to 200 millisecond round-trip budget that a single payment-channel update affords.

The generalized second-price auction offers a pragmatic compression. In GSP adapted for ranked inference slots, agents submit a single scalar bid per bundle configuration they are willing to accept. The provider's allocation engine ranks bids, assigns bundles to the highest bidders, and charges each winner the bid of the next-highest agent for that bundle. GSP is not incentive-compatible in the strict dominant-strategy sense. Agents may shade their bids below true value. Yet GSP admits locally enumerative Nash equilibria that converge rapidly under the repeated, high-frequency play characteristic of inference markets. Kira can compute her approximate best-response bid within a single channel-state update: she evaluates her valuation for at most a handful of predefined bundle configurations, selects the one with the highest surplus at the current estimated clearing price, and submits a single HTLC whose payment hash commits her bid amount. The computational and communication load drops from exponential in the number of resource items to linear in the number of bundle templates the provider publishes.

Settlement mechanics distinguish these formats further. Under VCG, the provider must compute externality-based prices after collecting all bids, then issue individual HTLC invoices to each winner reflect-

ing her personalized price. Losing agents' hash-locks expire after Δt_timeout without funds moving, which is correct and trust-free, but the two-round interaction, first bid submission then price notification and acceptance, doubles the latency envelope. Under GSP, the price rule is deterministic from the bid vector: each winner pays the next-highest bid. The provider can embed the pricing rule in the auction announcement so that Kira pre-computes her payment conditional on winning. She locks funds via a single HTLC whose preimage the provider reveals only upon allocation confirmation. One round-trip. One channel-state update. Settlement is atomic: Kira either receives the inference slot and the provider claims the payment by revealing the preimage, or the HTLC expires and no funds move.

The comparative verdict is not that one mechanism universally dominates. VCG is the correct choice when the bundle space is small enough that agents can enumerate valuations within the latency budget, and when allocative efficiency matters more than protocol simplicity, as in high-value batch training slot auctions that occur on the order of minutes. GSP is the operational default for sub-second inference slot allocation, where communication parsimony and single-round settlement are binding constraints. The mechanism designer's task is to parameterize the boundary: given a bundle complexity m, a channel round-trip time τ, and a valuation-computation budget c, select VCG when $2^m \cdot c < \tau$ and GSP otherwise. What both formats share is the property that the auction itself requires no custodian. Bids are HTLC commitments, winner determination is verifiable from the published bid set, and settlement is atomic on the payment channel. The auction is a protocol object, not a trusted service. That structural guarantee is what enables the population-level dynamics examined next, where many Kira-like agents adopt these bidding strategies simultaneously.

Implementing Sub-Second Auction Rounds with HTLC-Escrowed Bids and Preimage-Gated Slot Assignment

What happens when three agents contend for the same A100 inference slot within a 200-millisecond window and none of them knows what the others are willing to pay?

This is the core allocation problem that discovery alone cannot solve. Kira can enumerate available GPU slots, open channels to their providers, and read posted terms. But posted prices are static. When demand spikes and multiple agents converge on the same resource simultaneously, a fixed price either rations by arrival order, which is economic-

ally arbitrary, or clears at a level that leaves surplus on the table. The correct response is an auction: a mechanism that elicits private valuation information from competing agents and maps it to an allocation and payment rule with formal incentive-compatibility guarantees. The challenge is that each agent's valuation is not a simple scalar. Kira's willingness to pay for a given slot is a vector shaped by her upstream caller's latency ceiling, the inference-quality tier required, and the budget constraint propagated through her delegated macaroon. A rival agent facing a batch workload with relaxed latency but high throughput needs values the same slot along entirely different dimensions. Formalizing this, let each agent i hold a private type θ_i drawn from a commonly known distribution F over a multidimensional type space Θ, and let $v_i(\theta_i, x)$ denote agent i's valuation for allocation outcome x. The mechanism designer's task is to select an allocation rule $x(\theta)$ and a payment rule $p(\theta)$ such that truthful reporting of θ_i is each agent's optimal strategy, or failing that, to characterize the equilibrium bidding behavior under a weaker solution concept.

Three mechanism formats merit direct comparison. Under Vickrey-Clarke-Groves, each agent reports her full type, the allocation maximizes reported social surplus, and each agent pays the externality she imposes on others. Truthful bidding is a dominant strategy: Kira's payment equals the decrease in aggregate surplus that the remaining agents suffer from her presence, so misreporting her type can only reduce her own payoff. VCG is theoretically ideal, but its payment computation requires the auctioneer to solve the allocation problem once per agent to calculate each externality term, introducing computational overhead that scales poorly when dozens of agents compete for multiple slots per round. The generalized second-price format, familiar from ad auctions, ranks agents by bid and charges each winner the bid of the next-highest competitor. GSP is simpler to compute but not strategy-proof. Rational agents shade their bids below true valuations by an amount that depends on beliefs about rivals' strategies, producing a Nash equilibrium that can differ from the efficient outcome. The ascending-clock format offers a practical middle path. The auctioneer announces a price, agents declare demanded quantities, and the price increments until aggregate demand meets supply. Each round reveals a marginal slice of the demand curve without requiring agents to disclose full valuation vectors upfront. Critically, each price announcement and demand response can be encoded as a signed state update within an existing payment channel,

and the final clearing price settles via a single HTLC commitment, keeping the entire auction lifecycle within one channel round-trip.

When Kira operates under a sealed-bid format against $n - 1$ rivals whose valuations are drawn independently from a cumulative distribution F, her optimal bid-shading function follows from the standard Bayesian Nash equilibrium derivation for first-price auctions. If her true valuation is v, her equilibrium bid is b(v) = v − ∫₀ᵛ [F(t)/F(v)]^(n−1) dt. The integral term is the information rent she withholds, reflecting the fact that she need not pay her full value to win when rivals are likely to bid less. As the number of competitors n grows, [F(t)/F(v)]^(n−1) collapses toward zero for all t < v, compressing the information rent and pushing equilibrium bids toward true valuations. For Kira, this is a closed-form computation over a distribution she estimates from observed channel-open rates and historical bid data propagated through the provider's gossip layer. The entire calculation executes in microseconds on commodity hardware, well within the budget of a subsecond HTLC round-trip where the bid itself is escrowed by revealing a preimage only if the auctioneer certifies a winning allocation. If Kira loses, the HTLC times out and her channel balance is restored without settlement.

The result is that competitive allocation becomes a deterministic subroutine of Kira's payment stack rather than an ad hoc negotiation. She evaluates her upstream caller's latency-weighted valuation, queries the estimated rival density from recent auction rounds, computes her shaded bid, and submits it as an HTLC-escrowed commitment. The mechanism's incentive properties guarantee that her expected surplus is maximized given her information, and the ascending-clock variant provides a fallback when the rival distribution is too uncertain for reliable Bayesian estimation. These equilibrium bid functions are the inputs that the next stage of the analysis requires to construct continuous congestion-pricing surfaces under time-varying demand.

Nash Equilibrium Analysis of Agent Bidding Populations, Collusion Detection, and Incentive-Compatible Mechanism Design

What happens when forty-seven agents simultaneously request inference slots from an eight-GPU provider cluster and three of those agents are controlled by the same upstream orchestrator?

The bid functions derived earlier give each agent a clean optimization target: observe the congestion signal, condition on private valu-

ation, submit a bid that maximizes expected surplus against the posted price surface. But that derivation held every other agent's strategy fixed, a convenience that dissolves the moment the auction actually clears. When the auctioneer resolves those forty-seven bids into eight winning allocations, the clearing price each agent faces is itself a function of the other forty-six strategies. Kira's optimal bid depends on what the population plays, and the population's optimal play depends on what Kira bids. The single-agent optimization was a partial equilibrium sketch. The real question is whether a mutually consistent strategy profile exists across the entire bidding population, whether it is unique, and whether the mechanism can be designed so that truthful reporting of private valuations constitutes a dominant strategy rather than merely a fragile best response.

The three coordinated bids make the question urgent. Their submission timestamps fall within four milliseconds of each other, their valuations cluster within a suspiciously narrow band below the expected market-clearing price, and their bid increments move in lockstep across successive rounds. Coincidence, rational herding, or deliberate suppression? The mechanism must answer that question using only observable bid-stream data, then guarantee that even if collusive rings exist, the auction's incentive structure makes truthful bidding individually rational for every participant. Without that guarantee, the entire pricing architecture rests on an assumption no rational agent is obligated to honor.

Characterizing Symmetric Bayes-Nash Equilibria in Repeated Compute Auctions with Private Valuations

What happens when two hundred autonomous agents, each holding a private valuation for the same contested A100 inference slot, simultaneously submit sealed bids through micropayment channels with no trusted auctioneer to enforce fairness?

The question is not academic. Kira, having already ranked a set of heterogeneous GPU providers and established multi-hop payment routes to reach them, now faces exactly this scenario during a demand spike. Each competing agent draws its per-token willingness-to-pay from some private distribution — call it F over the interval $[\underline{v}, \overline{v}]$ — and must choose a bid $b(v)$ that maps its private valuation v to a channel-committed payment offer. In a standard first-price sealed-bid auction with n symmetric bidders, the symmetric Bayes-Nash equilibrium bidding strategy is well known: $b(v) = v - \int_{\underline{v}}^{v} \left(\frac{F(t)}{F(v)}\right)^{n-1} dt$. The

agent shades its bid below its true valuation, and the degree of shading shrinks as competition intensifies. But the micropayment channel setting introduces a hard constraint that classical auction theory ignores. Every bid must be backed by a hash-time-locked commitment with a channel fee δ that represents the marginal cost of locking collateral in the channel for the HTLC's timeout window $\Delta t_{\text{timeout}}$. This fee acts as a floor: no rational agent bids below $\underline{v} + \delta$, because the act of bidding itself consumes δ in opportunity cost. The equilibrium bid-shading is therefore bounded from below by channel economics, compressing the strategy space and pushing bids closer to truthful valuations than frictionless theory would predict.

Collusion threatens this equilibrium. A coalition of k agents that coordinate their bids can suppress competition within a slot auction, winning at artificially depressed prices. The formal signature of such a ring is observable without breaking any agent's privacy. Payment-channel open and close events on-chain carry timestamps and counterparty identifiers. When a cluster of agents consistently opens channels to the same provider set within a narrow time window, submits bids that cluster within a suspiciously tight range ϵ of each other, and closes channels in lockstep after settlement, the joint probability of these events occurring independently drops exponentially with the cluster size k. A detection oracle monitors these three signals — temporal correlation of channel lifecycle events, bid-value clustering relative to the empirical bid distribution, and synchronized settlement patterns — and flags coalitions whose observed joint frequency exceeds the independence-assumption threshold by a configurable factor α. To incentivize internal defection from rings, the mechanism includes a cryptographic whistleblower bounty: any ring member can publish a signed commitment $\sigma(m, sk_i)$ revealing the coordinated bid schedule m, and the mechanism awards a fraction of the ring's forfeited collateral to the whistleblower. Because the whistleblower's signature is verified against the channel's funding multisig, attribution is cryptographically binding, and the bounty makes defection a dominant strategy for any coalition member whose share of ring profits falls below the bounty value.

The mechanism that makes truthful bidding self-enforcing adapts the Vickrey-Clarke-Groves framework to streaming micropayments. In a classical VCG auction, the winner pays the externality it imposes on other bidders rather than its own bid. Translating this to HTLC-settled

inference slots requires decomposing the single VCG payment into a stream of hash-locked increments, each corresponding to a batch of generated tokens. The winner's per-increment payment equals the highest losing bid's per-token rate, locked via HTLC with preimage release gated on verifiable token delivery. Incentive compatibility follows directly: if an agent bids above its true valuation, it risks winning increments whose per-token cost exceeds its willingness-to-pay, producing negative surplus on each settled HTLC. If it bids below, it loses slots it would have profitably won. Truthful reporting of v is therefore strictly dominant. Individual rationality holds because no winning agent pays more per token than its reported valuation, and losing agents pay nothing since their HTLCs expire unredeemed. The proof composes cleanly with the channel-fee bound established above: the mechanism's VCG pricing already sets payment at the marginal externality, and the channel fee δ is incorporated as a participation cost that shifts the individual rationality constraint but does not violate it so long as $v_i > p_{\text{VCG}} + \delta$, a condition satisfied by construction for any agent whose valuation exceeds the market-clearing price plus channel overhead.

Kira's operational transformation is concrete. Before this mechanism, it faced an opaque bidding landscape with no principled strategy and no defense against coordinated manipulation. Now it computes $b(v) = v$ as its dominant strategy under VCG rules, submits truthful per-token willingness-to-pay locked in HTLC increments, and monitors channel-lifecycle timing to flag suspected rings. The equilibrium bid distributions that emerge from this population-level analysis are not endpoints.

Statistical Collusion Detection: Identifying Coordinated Bid Suppression via Entropy Analysis of Bidding Histories

Forty agents. Twelve A100 slots. The clearing price should have been discoverable in milliseconds, but Kira's bid history told a different story.

During a sustained inference demand spike across three regions, Kira observed a pattern that no equilibrium model predicted. Seven of the forty competing agents submitted bids within 0.3% of each other across fourteen consecutive auction rounds, clustering at roughly 62% of the expected Bayesian Nash equilibrium bid for a first-price sealed-bid auction with $n = 40$ independent bidders drawing valuations from a uniform distribution on $[0, \bar{v}]$. The equilibrium bid function $b(v) = v \cdot (n-1)/n$ yields shading of approximately 2.5% for a population of forty.

These seven agents were shading by 38%. That gap is not noise. It is coordinated suppression.

The detection mechanism that flagged this ring operates on a simple but powerful information-theoretic foundation. For each sliding window of w auction rounds, the protocol computes the Shannon entropy of the empirical bid distribution for every agent: $H(B_i) = -\Sigma\, p(b_k)\, \log_2 p(b_k)$, where $p(b_k)$ is the frequency of agent i bidding in price bin k across the window. An honest agent whose private valuation shifts with upstream caller demand and latency deadlines produces a bid history with entropy roughly matching the entropy of the underlying valuation distribution. Kira's own bid entropy across the same fourteen rounds measured approximately 3.2 bits for a 16-bin discretization. The seven suspect agents each registered between 0.4 and 0.7 bits. Their bids were collapsing into one or two bins, round after round. The protocol computes a pairwise mutual information score $I(B_i; B_j)$ between every agent pair. For the seven flagged agents, mutual information exceeded 2.8 bits, indicating near-deterministic coordination. For all non-flagged pairs, the score hovered below 0.15 bits, consistent with independent draws.

The entropy anomaly alone does not prove collusion, because a single agent with genuinely stable demand might legitimately bid in a narrow range. The protocol therefore compounds the entropy signal with a graph-theoretic sybil score. It constructs a weighted graph where each agent is a node and edge weights correspond to normalized mutual information. A minimum-cut algorithm partitions the graph, and any clique of $k \geq 3$ agents whose internal edge weights exceed a threshold τ while external edges remain below $\tau/4$ triggers the collusion flag. In this incident, the seven agents formed a connected subgraph with average internal weight 2.81 bits and average external weight 0.09 bits. The separation was unambiguous.

Once flagged, the protocol's response is automatic and purely mechanism-theoretic. It recomputes the dynamic reserve price, replacing the exponentially weighted moving average of recent clearing prices with a corrected estimator that excludes flagged bids. Before detection, the reserve had drifted down to 0.0041 BTC per GPU-second, depressed by fourteen rounds of artificially low bids from the ring. After exclusion, the corrected reserve snapped to 0.0068 BTC per GPU-second, reflecting genuine demand from the remaining thirty-three honest bidders. The modified second-price auction charged each winning bidder the maximum of the 13th-highest non-flagged bid and this corrected reserve. Kira, who had submitted a hash-committed truthful bid of 0.0073

BTC per GPU-second via the commitment-reveal scheme, won a slot and paid 0.0069 BTC per GPU-second. Without the detection mechanism, Kira would have either lost the slot to ring members bidding just above the suppressed reserve, or won at the suppressed price only to find the ring extracting surplus through side payments invisible to the protocol.

The critical lesson is compositional. Cryptographic commitment prevents front-running: Kira's bid hash $H(b \parallel nonce)$ was published before the bidding window closed, and the plaintext reveal was verified against it afterward. Entropy-based anomaly detection identifies coordination that commitment alone cannot prevent, because colluding agents can pre-agree on suppressed bids before committing. The graph-theoretic sybil score distinguishes genuine low-entropy behavior from cartel formation. And the dynamic reserve adjustment converts detection into economic consequence, eliminating the profit motive for ring formation in the first place. Each layer depends on the one beneath it. Remove any single component, and the mechanism leaks surplus to adversaries. Stack them, and truthful bidding becomes the dominant strategy not because agents are honest, but because the protocol makes dishonesty unprofitable. Kira's secured slot and preserved budget are the measurable outcome. The real result is that the auction's equilibrium properties survived contact with an adversarial population, and the clearing prices flowing into the next section's congestion-pricing surface remain trustworthy inputs rather than corrupted signals.

Designing Dominant-Strategy Incentive-Compatible Mechanisms That Compose with Micropayment Settlement Constraints

A mechanism can be strategyproof and still useless if agents cannot settle their payments at the speed the auction clears. That tension sits at the center of every design decision in this section. Classical Vickrey-Clarke-Groves theory guarantees that truthful reporting is a dominant strategy in a single-shot combinatorial auction, but its standard construction assumes an idealized treasury that can collect and redistribute payments instantaneously. When the settlement layer is a network of bilateral micropayment channels with discrete hash-locked increments, finite channel capacity, and multi-hop routing latency, the mechanism designer faces a harder problem: composing incentive compatibility with the physical constraints of cryptographic value transfer.

The decision framework begins with three criteria that jointly determine whether a dominant-strategy mechanism survives contact with

micropayment settlement. First, payment granularity. A VCG-style auction charges each winner the externality it imposes on others, and that externality is denominated in continuous currency units. But an HTLC-based payment channel settles in discrete increments, each increment requiring a fresh preimage reveal. If the smallest settleable unit exceeds the difference between two agents' externality charges, the mechanism cannot distinguish their truthful bids from strategic deviations. The designer must bound the efficiency loss introduced by this discretization. Formally, let δ represent the minimum HTLC increment along the payment path from bidder to auctioneer. If the externality gap between the efficient and second-best allocation is smaller than δ, the mechanism loses its separation power. The practical bound is this: allocative efficiency degrades by at most δ per agent per round, which means the channel's configured minimum increment directly governs mechanism fidelity. Choosing δ too large saves routing overhead but coarsens the incentive structure. Choosing it too small multiplies the number of hash operations per auction round. The right value depends on the distribution of valuations in the bidding population. When Kira computes its Bayes-Nash equilibrium bid as a function of its private per-token valuation and the known distribution of competing agents, the mechanism must be able to charge Kira its VCG price to within δ of the true externality, or truthfulness ceases to dominate shading.

Second, settlement latency relative to auction cadence. A repeated auction clearing every few hundred milliseconds cannot wait for multi-hop HTLC resolution that takes seconds. The mechanism must therefore separate the commitment phase from the settlement phase. During commitment, each bidder locks a conditional payment equal to its maximum possible VCG charge, using a time-locked HTLC whose preimage the auctioneer reveals only after computing the allocation. During settlement, the auctioneer reveals partial preimages or structured hash chains that release exactly the computed VCG price, refunding the remainder by letting the lock expire. This two-phase structure preserves dominant-strategy incentive compatibility because the bidder's reported valuation determines the allocation, not the payment timing. The constraint it introduces is a capital lockup proportional to the agent's maximum possible externality charge, drawn from its budget-hierarchy allowance. If Kira's delegated sub-budget ceiling from its root principal is B, and the maximum VCG charge across all plausible allocations is c_{max}, then Kira can participate in at most B / c_{max} concurrent auction rounds before its capital is fully locked in pending HTLCs.

Third, individual rationality under budget hierarchies. A mechanism is individually rational when no agent loses money by participating. Standard VCG satisfies this trivially because payments never exceed reported valuations. But when an agent like Kira operates under a cryptographically enforced spending cap, the binding constraint is not its valuation but its remaining budget. The adapted mechanism must verify, before accepting a bid, that the agent's channel balance can cover c_max for that round. This check composes naturally with the macaroon-attenuated delegation model from earlier chapters: the auctioneer inspects the spending caveat embedded in Kira's credential and rejects any bid whose maximum liability would violate it. The result is a mechanism where truthful bidding remains dominant, participation never risks overspending the delegated budget, and the efficiency loss from discrete settlement is bounded by a transparent, tunable parameter.

These three criteria interact as a single tradeoff surface. Finer granularity improves incentive fidelity but increases hash operations per round. Faster auction cadence improves allocative responsiveness but demands more locked capital per agent. Tighter budget ceilings protect principals but reduce the number of simultaneous rounds an agent can contest. The designer's job is to set δ, the auction period, and the capital-lock policy so that the resulting mechanism remains dominant-strategy incentive compatible under the worst-case discretization error, individually rational under the tightest plausible budget constraint, and settleable within the HTLC timeout window. When these parameters are tuned correctly, the equilibrium bid functions derived earlier in this section compose cleanly with the payment stack, and the aggregate bid data flowing through settlement channels becomes the raw signal from which congestion-pricing surfaces can be constructed.

Three concentric rings of mechanism design now lock together into a single self-regulating allocation engine. The marginal-cost pricing curve, published by each provider as a monotonic function of current slot utilization, establishes the cost landscape that every bidding agent faces. The sealed-bid second-price auction, with bids committed as hash-locked preimages and revealed at auction close, selects winners within that landscape in a way that makes truthful valuation disclosure incentive-compatible regardless of an agent's computational sophistication or strategic awareness. The Nash equilibrium analysis of the resulting bidding population then confirms that Kira's dominant strategy, bid her true marginal value for the next inference slot shaded only by the published congestion price, remains stable against unilateral deviation and

resistant to coordinated bid suppression detectable through statistical divergence on commitment reveal patterns. None of these rings function independently. The auction's incentive compatibility depends on atomic settlement at bid granularity. The equilibrium's stability depends on the pricing curve's fidelity as a congestion signal. The pricing curve's credibility depends on the provider's inability to manipulate reported utilization without forfeiting locked collateral. Every dependency bottoms out in the same cryptographic commitment infrastructure that the payment stack already provides. No new trust assumptions enter the system.

A demand spike hits the network. Forty-seven agents simultaneously need the same scarce H100 inference slot. There is no queue manager, no cloud console, no human operator. Congestion prices climb along a smooth, published curve.

Chapter Nine

Programmable Micro-Budget Hierarchies and Spending Policy Control

Kira's dispatch loop fires at 2:47 AM on a Tuesday, spawning fourteen child agents to absorb a burst of inference requests across four GPU providers. Within ninety seconds, three of those children have independently committed the same $4.20 balance to different providers. The parent channel is overdrawn. Every macaroon each child carried was valid. Every attenuation caveat checked out. But nothing in the delegation chain enforced the constraint that their *combined* spending could not exceed a single shared ceiling.

This is not a bug in any individual credential. The cryptographic construction worked exactly as specified. The failure is structural: flat, single-level delegation gives each bearer token an isolated view of the budget, and isolated views diverge the moment concurrent agents draw from a common pool. Kira's operator had built a system that could delegate authority but not subdivide solvency.

Flat delegation got Kira into the micropayment economy. Only a hierarchical budget structure can keep it solvent. The fourteen-agent

overdraft exposes the absence of a layer that sits above individual macaroons and below the operator's root funding. To prevent it, we need to define what a budget hierarchy actually looks like when it must span an arbitrary depth of autonomous agents, and that starts with the concept of a root envelope.

Budget Hierarchy Design: Root Envelopes, Delegated Sub-Agent Micro-Budgets, and Per-Task Scoping

Kira splits 4,200 satoshis into three envelopes before the first GPU cycle fires. One envelope caps a vision-encoder pass at 1,400 satoshis with a provider-class constraint limiting it to verified CUDA endpoints. A second, carrying 1,600 satoshis, targets a language-model completion provider and embeds a maximum per-token price of 12 millisatoshis. The third holds the remainder for a reranking step, scoped to a latency ceiling of 90 milliseconds and revocable the instant the parent agent determines the reranker is redundant. Each credential is minted by attenuating the root macaroon with caveats that bind spending ceiling, task type, and provider parameters into the bearer token itself. No custodian adjudicates these boundaries. The cryptographic construction enforces them.

But the delegation act alone is not enough. A child agent holding a 1,400-satoshi credential can, absent further constraint, redirect every satoshi toward a provider and purpose its parent never authorized. And the summation invariant that all live child budgets plus settled spend must never exceed the root is trivial to state yet brutal to enforce when three child agents operate concurrently, settle through separate HTLC paths, and may themselves delegate further. Enforcement becomes a distributed coordination problem that must resolve in milliseconds without a central ledger.

This is the architectural problem the section formalizes: the data structures encoding a budget directed acyclic graph, the cryptographic commitments that make the summation invariant hold across arbitrary delegation depth, and the per-task scoping rules that prevent fund repurposing across job boundaries.

Root Envelope Construction: Cryptographic Binding of Top-Level Budget Commitments to Channel Capacity

Kira splits its root spending authority into three parallel allocations, each bound to a distinct child inference-fetcher, and the entire subdivision completes before a single satoshi moves. That act of pre-commit-

ment, trivial in its operational appearance, demands a construction that is anything but trivial: a cryptographic envelope anchoring the top-level budget ceiling to the actual channel capacity funding it, so that no sequence of delegations can promise more value than the principal controls.

A root envelope is a signed data structure combining three commitments. First, a capacity binding ties the envelope's maximum spend ceiling to a verifiable on-chain funding output. If Kira's payment channel is backed by a 2-of-2 multisig locking 10,000 satoshis, the root envelope's ceiling field commits to a value at most equal to the local balance in the latest channel state. The binding is a signature over the tuple (channel_id, local_balance, envelope_ceiling, nonce), produced by the principal's channel key. Any verifier holding the channel's funding transaction can confirm that envelope_ceiling ≤ local_balance without accessing private channel state, because the signature itself constitutes a proof of authorized capacity reservation. Second, the envelope embeds a base macaroon whose root key is derived from the principal's identity key. This macaroon carries first-party caveats encoding the top-level policy surface: a global rate limit expressed as satoshis per second, a provider whitelist constraining which node public keys may receive payments, and an absolute expiry timestamp beyond which the entire envelope and every credential descended from it becomes invalid. Third, the envelope includes an allocation counter initialized to zero. This counter is the running sum of all child budget ceilings carved from the root, and the core invariant of the hierarchy is that this counter must never exceed envelope_ceiling at any point in the DAG's lifetime.

The structural relationship between root and children follows a strict directed acyclic graph. When Kira delegates a sub-budget to a child agent, it constructs a child envelope by attenuating the base macaroon with additional caveats that narrow the task scope, reduce the spending ceiling, and tighten the time window. A child scoped to GPU provider P_1 with a ceiling of 500 satoshis and a 30-second TTL receives a macaroon whose caveat chain reads: provider = P_1, max_spend = 500, expiry = t_0 + 30s. These caveats are appended via HMAC chaining in exactly the manner established by the single-level delegation primitive, but now the parent also increments its allocation counter by 500. Before issuing a second child envelope for provider P_2 at 500 satoshis and a third for P_3 at 400 satoshis, Kira checks that 500 + 500 + 400 = 1,400 ≤ envelope_ceiling. If the envelope ceiling is 1,500, the delegation proceeds. If it is 1,300, the third allocation is rejected locally before any credential is

emitted. The arithmetic is trivial. The enforcement is the point: no child can be instantiated without the parent proving that the sum of outstanding delegations fits within the cryptographically bound capacity.

Per-task scoping at the leaf level eliminates lateral spend drift. Each leaf budget authorizes exactly one atomic unit of work. A child agent holding a 500-satoshi ceiling and a provider-restricted macaroon cannot repurpose those funds toward a different provider or a different inference model, because the caveat chain physically prevents any verifier from accepting a payment presentation that violates the embedded constraints. The leaf's spending record, a sequence of signed HTLC commitments against that 500-satoshi ceiling, constitutes a complete audit trail. After task completion, the parent can reconstruct total spend across all children by summing the individual leaf expenditures and comparing against the root allocation counter, verifying that no satoshi was double-committed and no caveat was circumvented.

This static hierarchy, the DAG of envelopes with monotonically tightening caveats and a parent-tracked allocation sum, is the scaffolding. It defines who may spend, how much, for what, and until when. What it does not yet address is what happens when a child violates a constraint at runtime, when a leaf budget expires with unused capacity that must be reclaimed, or when the root itself needs to revoke an entire subtree instantaneously. Those are enforcement and revocation problems. They operate on the structure built here, and they require it to be exactly this rigid.

Delegated Sub-Agent Budgets via Macaroon Attenuation: Encoding Spending Ceilings into Bearer Credentials Without Key Exposure

The hum of a multiplexed Lightning channel carrying four simultaneous sub-agent payments is inaudible, but the accounting it demands is not. When Kira held a single attenuated macaroon from its human principal, spending authority was a scalar: one credential, one cap, one delegate. The moment Kira spawns parallel child agents to negotiate GPU providers concurrently, that scalar must become a tree. The structural question is precise: how does a bearer credential system that was designed for point-to-point attenuation extend into a directed acyclic graph of nested spending authorities without exposing any parent's signing key to any child, while maintaining the global invariant that the sum of all active delegated ceilings never exceeds the root allocation?

The root envelope is the answer's foundation. Define it as a macaroon M_root whose first-party caveat encodes a satoshi ceiling C_root, a validity window [t_start, t_end], and an HMAC chain anchored to the human principal's root key k_0. This envelope is the cryptographic commitment from which all downstream spending authority derives. Kira receives M_root with C_root = 50,000 sats. At this point, the unspent balance U_root = C_root and the total delegated amount D_root = 0. The hard invariant, enforced at every allocation event, is D_root + spent_root ≤ C_root, where spent_root accounts for any direct expenditures Kira makes outside of delegation.

When Kira decides to spawn a child agent for a single A100 inference batch, it constructs a child macaroon M_child_1 by appending attenuating caveats to a copy of M_root's HMAC chain. The critical caveats are: amount_cap = 5,000 sats, provider_class = A100_GPU, max_per_call = 200 sats, task_nonce = n_1, and parent_ref = H(M_root || n_1). The amount_cap caveat is a strict upper bound. Because macaroon attenuation is monotonically restrictive, the child cannot widen its ceiling beyond what the parent encoded. The parent_ref field links the child back to Kira's root node via a hash commitment, enabling upward accounting rollup without requiring the child to possess or even see the root key k_0. Kira's local state now records D_root = 5,000, and U_root drops to 45,000. A second child M_child_2 with amount_cap = 5,000 and task_nonce = n_2 reduces U_root to 40,000. Four such children bring it to 30,000. The arithmetic is trivially verifiable at each allocation step, and no child holds material that could reconstruct a sibling's credential or inflate its own ceiling.

The DAG extends to arbitrary depth. If M_child_1 needs to subdelegate, say to a retry handler that will attempt a fallback provider, it mints M_grandchild_1a with amount_cap ≤ 5,000 - spent_child_1 and its own task-scoped caveats. The same monotonic restriction applies: grandchild ceilings cannot exceed the child's remaining unspent balance at the moment of minting. Each node in the DAG maintains three values locally: its own ceiling, its cumulative spend, and the sum of ceilings delegated to its children. The invariant spend_node + delegated_node ≤ ceiling_node holds at every node, and because each node's ceiling is itself bounded by its parent's invariant, the global constraint propagates upward by induction from

leaves to root. No central ledger recount is required. The tree's structure is the proof.

Per-task scoping is what makes this architecture operationally safe rather than merely arithmetically correct. Binding M_child_1 to task_nonce = n_1 means its spending authority exists for exactly one unit of work: one inference batch negotiation with one provider class during one time window. When that task completes, whether through successful settlement or graceful failure, the child reports its final spent_child_1 value. Kira reclaims ceiling_child_1 - spent_child_1 back into U_root. If the child spent 3,200 of its 5,000-sat budget, 1,800 sats flow back to the root's available balance. If the child's task fails entirely and spends nothing, the full 5,000 returns. Budget exhaustion is localized by construction: a child that burns through its ceiling simply stops transacting, and no sibling or cousin node is affected.

What emerges is a static allocation topology, a DAG whose nodes are macaroon-attenuated budget envelopes, whose edges are HMAC-chained parent-child derivations, and whose global safety property follows from local arithmetic checks at each node.

Per-Task Scoping as Containment Boundary: Isolating Inference-Job Expenditure from Persistent Agent Balances

Every satoshi delegated to a child agent must remain sealed inside the boundary of the task that justified its creation. This is the core structural guarantee that separates a programmable budget hierarchy from a shared checking account with multiple signers. The single-level macaroon delegation you already know from Chapter 5 gave one principal the ability to attenuate a bearer credential for one delegate. Now we stack that mechanic recursively, constructing a directed acyclic graph of spending authority rooted in a single cryptographic envelope, where every leaf node maps to exactly one unit of work.

The root envelope is the graph's origin vertex. Kira's operator (or Kira itself, once operating autonomously) commits a spending ceiling, say 50,000 sats, into this envelope by signing an HMAC-bound commitment that locks the total balance to a specific key identity. Every sub-budget minted downstream decrements this root atomically. If Kira issues three child envelopes of 15,000 sats each, the root's available balance drops to 5,000 sats in the same state transition that creates those children. No race condition, no double-commitment. The invariant is simple and absolute: the sum of all outstanding child ceilings plus the

root's remaining reserve equals the original committed amount. Violation of this invariant is structurally impossible because each minting operation is a single atomic compare-and-decrement against the root's counter.

Each child envelope carries an attenuation chain that binds it to four constraints simultaneously. First, a child agent identity, the public key of the specific sub-agent authorized to present the credential. Second, a maximum satoshi ceiling that the child can never exceed regardless of how many individual payments it attempts. Third, an expiry window, a TTL of perhaps 30 seconds, after which the credential becomes cryptographically invalid. Fourth, an allowable vendor class, restricting spend to a defined set of service endpoints, such as GPU-slot providers on a particular compute marketplace. These four caveats are appended as successive HMAC layers in the macaroon chain, each layer further restricting the scope of the parent's grant. The child agent can attenuate further if it needs to issue grandchild envelopes for sub-tasks, but it can never widen any caveat. Authority only narrows as you descend the DAG.

Per-task scoping binds each leaf envelope to a discrete, terminal unit of work. One envelope funds one inference job. One envelope funds one batch of GPU-slot procurement bids. The binding is not advisory. The task identifier is embedded as a caveat in the attenuation chain, so the payment endpoint will reject any presentation of the credential for a different task context. When the task completes, whether successfully or through failure or timeout, the protocol triggers an upward reclamation sweep. Unspent balance returns atomically to the parent envelope that issued it. If one of Kira's three GPU-procurement agents finishes its job having spent only 11,000 of its 15,000-sat allocation, the remaining 4,000 sats flow back to the root. The root's available balance rises from 5,000 to 9,000 sats, immediately eligible for reallocation to a fourth task envelope.

This reclamation mechanic eliminates idle capital at leaf nodes. Without it, unspent sats would linger in expired envelopes, effectively locked out of the economy of the agent swarm until a human operator noticed and manually swept them. With atomic upward reclamation, the budget DAG behaves like a capacitor bank: energy flows down to where work is happening, then returns the moment demand ceases. Kira can now run three parallel procurement agents, reclaim surplus from the fastest finisher, and immediately fund a fourth agent with that surplus, all without any manual intervention and without ever risking a total commitment that exceeds the original 50,000-sat root.

The anatomy of this DAG, root envelope, attenuated child branches, task-scoped leaves, atomic reclamation edges, is the static skeleton on which everything that follows in this chapter will operate. Once you grasp the shape of the tree and the directionality of its constraints, the runtime questions become tractable: how do you monitor spend velocity across branches in real time, how do you enforce invariants when network partitions delay reclamation messages, and how do you propagate a revocation signal from root to every descendant in bounded time? Those are the next concerns. For now, the hierarchy itself is the foundation, and its defining property is containment: no branch can ever reach beyond the ceiling its parent granted, and no task can ever bleed into another task's allocation.

Declarative Policy Specification: Per-Transaction Caps, Cumulative Ceilings, Provider Whitelists, and Exhaustion Fallbacks

Kira's third child agent locks a payment hash and prepares to route 49,500 satoshis to a single inference provider whose spot price just spiked. The child holds a macaroon with a cumulative ceiling of 50,000 satoshis, and nothing in the caveat chain says it cannot burn nearly the entire allocation on one call. The HTLC is about to commit. From the budget DAG's perspective, every invariant holds: the child's ceiling does not exceed its parent's delegated partition, the running balance ledger will update atomically, and the root budget remains solvent. But the parent delegated that 50,000-satoshi envelope expecting it to cover roughly ten inference rounds across three candidate providers, not one outsized purchase from whichever endpoint happened to quote first. The hierarchy is structurally sound and behaviorally unconstrained.

This is the gap that raw budget subdivision cannot close. A ceiling bounds total exposure but says nothing about the shape of permissible spending beneath it. Per-transaction caps, rolling cumulative windows, provider-scoped whitelists, and exhaustion-triggered fallback policies are not optional refinements layered on after the core design ships. They are the enforcement surface that makes delegation trustworthy at every edge of the tree. Each constraint composes as an additional attenuation caveat appended to the macaroon chain the child already carries, evaluated locally before the HTLC lock fires, so that no payment leaves the agent's custody unless every policy predicate returns true. The machinery that partitions value is built. What follows is the grammar that governs it.

Policy Grammar for Spending Constraints: Expressing Per-Transaction Caps and Cumulative Ceilings as Composable Caveat Chains

Kira's child agent grabs a macaroon, appends a caveat binding maximum spend to 500 satoshis, and fires a payment at a GPU provider. The HTLC settles. No round-trip to Kira. No centralized policy server. The constraint traveled with the credential, and the verifier enforced it locally by recomputing the HMAC chain. That single interaction encapsulates the core problem this section solves: how to express a complete, composable spending policy as a sequence of first-party caveats that any node in the budget DAG can evaluate without upstream coordination.

The grammar starts with the simplest constraint. A per-transaction cap caveat binds a single field: `max_send_sat ≤ V`, where `V` is the ceiling in satoshis for any individual HTLC or payment-hash commitment routed through the credential. Because macaroon attenuation is strictly subtractive, a child agent receiving a token with `max_send_sat ≤ 500` can mint a derivative with `max_send_sat ≤ 200` but never `max_send_sat ≤ 1000`. The verifier checks the tightest bound in the chain, and the HMAC-chained construction guarantees that no intermediate party has loosened any caveat without invalidating the signature. This property is not advisory. It is cryptographic. A child that attempts to strip or weaken a caveat produces a token whose HMAC verification fails against the root key, and the payment never dispatches.

Cumulative ceilings demand state. A per-transaction cap constrains each payment independently, but an agent tasked with a parallel inference job might execute dozens of small payments that individually pass the cap yet collectively drain the entire sub-budget in seconds. The cumulative ceiling caveat introduces a monotonic accumulator: `cumulative_spend_sat ≤ C within window W`. The window parameter is where design decisions bite. An epoch-based window resets at fixed boundaries, making accounting simple but creating burst vulnerabilities at epoch seams. A sliding window (roughly analogous to a token-bucket rate limiter) smooths spend over time but requires the evaluating node to maintain a bounded log of recent payment timestamps and amounts. A session-scoped window ties the ceiling to the credential's lifetime, meaning spend never resets and the token simply exhausts. Each choice carries replay-protection implications: epoch-based windows must bind a nonce or epoch identifier into each payment proof to prevent a settled HTLC from being counted in both the old and new epoch, while ses-

sion-scoped windows need no such guard because the accumulator only moves forward.

Provider whitelists add a spatial dimension to the temporal and volumetric constraints already in the chain. A caveat of the form payee_pubkey ∈ {P_1, P_2, P_3} restricts which Lightning node public keys or L402-endpoint identities may appear as the terminal hop of any routed payment. Kira, selecting three pre-vetted GPU providers whose latency and pricing survived the discovery and auction phases of earlier chapters, encodes their compressed public keys directly into the caveat. The child agent's payment logic checks the destination against this set before constructing the onion route. Any attempt to pay an unlisted provider fails at caveat evaluation, not at network routing, which means the budget is never at risk from a compromised or misbehaving child selecting an adversarial endpoint.

Exhaustion fallbacks complete the grammar. When any caveat in the chain would be violated by a proposed payment, the policy must define resolution rather than silent failure. The fallback caveat specifies an ordered strategy encoded as a priority list: first, degrade the requested service tier (switch from a high-parameter model to a cheaper quantized variant, reducing per-inference cost below the remaining ceiling); second, queue the request and batch it into the next epoch window; third, escalate to the parent by presenting a re-delegation request signed with the child's own key, asking Kira to mint a fresh sub-budget; fourth, refuse the request gracefully with a machine-readable error code the upstream caller can interpret. This ordering matters. A child agent that escalates before attempting degradation wastes Kira's signing bandwidth. A child that refuses before attempting queuing discards work that could have completed moments later.

Composed together, these four caveat types form a policy chain that Kira mints in a single credential: max_send_sat ≤ 500 | cumulative_spend_sat ≤ 10000 / sliding_60s | payee_pubkey ∈ {P_1, P_2, P_3} | on_exhaustion → [degrade, queue, escalate, refuse]. Every downstream agent that touches this token can only tighten these bounds, never loosen them. Every verifier evaluates the full chain locally by recomputing the HMAC cascade and checking each predicate against the proposed payment. The policy travels at the speed of the credential, not at the speed of a network call to a central authority. And when runtime monitoring picks up these credentials in flight, every in-

variant encoded here becomes a checkable assertion, a cryptographic receipt waiting to be audited.

Provider Whitelists and Capability-Scoped Authorization: Restricting Payment Flow to Verified Service Endpoints

The hum of a signed HTLC leaving an agent's payment library carries a specific shape. Not arbitrary. Not improvised. Every byte of that commitment reflects a policy document that was sealed into the token long before the agent ever encountered a service endpoint. That document is the artifact this section defines: the declarative policy specification that lives inside a macaroon's caveat chain and determines, with cryptographic finality, which providers a sub-agent can pay, how much per transaction, how much in aggregate, and what happens when the money runs out.

Consider the gap Kira faced before this machinery existed. Her child agent held environment variables: `max_spend=0.002`, `provider=any`. Suggestions. Hopes. Nothing in the HMAC chain enforced them. The child could misread a decimal, target an unvetted endpoint, and drain a budget in a single malformed call. Now Kira mints a macaroon whose first-party caveats encode four orthogonal constraint axes in a structured CBOR payload: a per-transaction cap of 50 satoshis, a cumulative ceiling of 400 satoshis over a rolling 60-second window, a provider whitelist keyed to three specific Lightning node pubkeys, and an exhaustion-fallback clause specifying `degrade → queue → escalate`. These caveats are not annotations. They are conjunctive predicates. A payment must satisfy every active constraint simultaneously, or the agent's local payment library rejects it before any cryptographic commitment escapes the process boundary.

The provider whitelist deserves particular attention because it converts authorization from an open set to a closed one. Each entry binds a 33-byte compressed public key or an L402 service identifier to a capability scope. A caveat reading `provider_in: [03a1f2..., 02bc7e..., 028d4a...]` means the child agent's payment library will refuse to construct an HTLC whose destination node falls outside that set. No DNS lookup, no HTTP header, no runtime configuration can override this. The whitelist travels inside the HMAC chain. Forging a new entry requires breaking the chaining key's preimage, which reduces to inverting SHA-256. Capability scoping goes further: each whitelisted provider can be tagged with permitted service classes. Provider `03a1f2...` might be authorized for `gpu-inference-fp16` while `02bc7e...` is restricted

to gpu-inference-int4. The child agent inherits these scopes as hard constraints.

The exhaustion-fallback clause is the axis most often omitted and most consequential when absent. Without it, a child agent hitting its cumulative ceiling simply fails. Silent failure in an agentic pipeline propagates as latency, then as stale data, then as cascading upstream timeouts. A first-class fallback clause eliminates this. The clause specifies an ordered resolution strategy encoded directly in the caveat payload: first, degrade to a cheaper provider on the whitelist (switch from 03a1f2... at fp16 to 028d4a... at int4 quantization). If no cheaper option remains, queue the request with an expiry window. If the queue window elapses, escalate to the parent agent for re-authorization. Each step is a distinct state transition the payment library executes locally. No human in the loop. No ambient authority call.

The four axes interact through strict conjunction, and this is the property that makes the policy composable under further delegation. When the child agent sub-delegates a slice of its budget to a grandchild, the grandchild's macaroon can only add caveats. It can narrow the whitelist from three providers to one. It can lower the per-transaction cap from 50 satoshis to 20. It cannot widen any constraint. The HMAC chain guarantees monotonic attenuation: each successive caveat is hashed into the chain, and verification walks the entire sequence, rejecting any token whose predicates violate the parent's boundaries. This is not policy inheritance by convention. It is policy inheritance by cryptographic construction.

What emerges is a constraint language with formal semantics that any compliant payment library can parse, enforce, and compose without schema lookups or external policy servers. The policy is the token. The token is the authority. And the authority shrinks at every delegation boundary, never grows. With this vocabulary locked in, the next structural question becomes inevitable: what happens when these policies subdivide recursively across a multi-level budget DAG, and how does the system verify that the invariants hold all the way down?

Exhaustion Fallback Cascades: Graceful Degradation Strategies When Sub-Budgets Deplete Before Task Completion

A child agent is three-quarters through a multi-step retrieval task. Its epoch budget sits at 412 sats remaining. The next inference call costs 500. No human is watching. No fallback rule exists. The agent stalls, the

task dies, and every sat already spent on prior steps evaporates as waste. This is the scenario that exhaustion fallback cascades eliminate.

The core decision when authoring a fallback policy is sequencing. A declarative caveat chain must encode an ordered list of degradation tiers, each tier binding a provider whitelist to a tightened per-transaction cap. Kira's policy for her child agent illustrates the grammar directly: tier-0 authorizes providers A and B at a 200-sat cap, tier-1 authorizes provider C at 100-sat cap, and tier-2 halts execution and emits an escalation request to the parent node in the budget DAG. Each tier activates only when the prior tier's provider set is either fully exhausted against the cumulative ceiling or breaches a latency threshold encoded as a caveat predicate. The child never chooses its fallback. The policy fires deterministically.

Serializing these tiers into a macaroon caveat chain demands a specific construction. Each tier occupies a conditional caveat branch, keyed by the L402 service identity of the whitelisted providers. The branch predicate references two state variables: the monotonic spend counter for that tier's provider set and the epoch clock. When the counter reaches the tier's cumulative sub-ceiling, the macaroon verification routine rejects any further HTLC destined for that provider set and advances to the next branch. The critical invariant holds: the sum of all tier sub-ceilings plus any uncommitted remainder equals the parent's total allocation. A Pedersen commitment check at the parent node confirms this additively without revealing individual tier balances. Every intermediary relay can verify the active tier's cap independently. No trust required.

The hardest tradeoff in cascade design lives between granularity and latency. Fine-grained tiers, maybe five or six deep, give maximum flexibility. But each tier transition requires the agent to re-resolve provider endpoints via the L402 service registry, adding roughly tens of milliseconds per hop. Coarse cascades with only two tiers minimize switching overhead but leave less room for graceful degradation. The decision factor is task criticality. A long-running data-assembly pipeline tolerates the latency of deep cascades because partial results retain value. A real-time bidding agent cannot afford even one tier switch mid-auction. Policy authors must weight these dimensions explicitly: task duration, partial-result salvageability, provider diversity, and per-tier cap tightening ratio.

Escalation as the terminal tier deserves particular attention. When all provider tiers exhaust, the final caveat branch encodes a structured escalation payload: the agent's current task state hash, the remaining budget shortfall in sats, and a re-authorization request signed with the child's

session key. The parent node receives this payload and faces its own policy decision: re-fund the child from uncommitted reserves, reallocate from a sibling's underutilized sub-budget, or propagate the escalation upward. This recursion is bounded by the DAG depth. Each level's policy independently caps how much re-authorization it can grant without consulting its own parent. The recursion terminates at the root, which holds the hard ceiling.

Practical guidance for cascade authoring compresses to four rules. First, never leave the terminal tier undefined. An agent without an explicit halt-and-escalate rule will attempt to operate with zero budget, generating rejected HTLCs that waste channel bandwidth. Second, tighten per-transaction caps monotonically across tiers. Tier-1's cap must be strictly less than tier-0's. This forces the agent toward cheaper providers as budget shrinks. Third, encode latency thresholds as caveat predicates, not as application logic. If provider A exceeds 800ms response time, the policy itself triggers the tier transition. Fourth, set tier sub-ceilings so that the final tier retains enough budget for at least one complete atomic operation. A fallback tier with 3 sats remaining and a 100-sat cap is dead on arrival.

What this framework produces is a single serializable artifact. One macaroon. Multiple conditional caveat branches. Monotonic counters. Cryptographic commitment checks. The child agent, every relay, and every provider can parse and enforce the policy independently at each hop. The cascade fires without a round-trip to Kira. The next section addresses how an agent evaluates these declarative policies in real time, synchronizing counters across concurrent children racing against the same epoch clock.

Gas-Optimization Strategies: Transaction Batching, Settlement Aggregation, and L1 vs L2 vs State Channel Cost-Latency Selection

Kira fans twelve sub-budget macaroons to child agents and broadcasts the channel-open commitments. The L1 confirmation clears, and the arithmetic is immediately wrong: each channel-open consumed roughly 620 sats in gas, totaling around 7,440 sats in settlement overhead, while the smallest child budget was only 400 sats. The hierarchy's own bookkeeping just spent more than one of its leaves was authorized to spend in total. The delegation DAG is cryptographically sound, revocation propagates correctly, every caveat attenuates as specified. None of that matters. The system has eaten itself from the inside.

This is not a stress-case edge condition. It is the default outcome whenever sub-delegation granularity drops below a threshold set by the settlement layer's per-transaction cost. And that threshold, on any L1 with meaningful security, sits orders of magnitude above the budget sizes that make hierarchical micro-delegation useful in the first place. The invariant that the root budget funds actual work rather than infrastructure friction requires an engineering response along three axes: compressing multiple settlement actions into single on-chain artifacts, aggregating finality proofs so that a batch of child-agent state transitions resolves as one settlement event, and selecting dynamically among L1, rollup, and state-channel regimes based on the cost-latency-finality surface each child agent's task profile demands. Each lever reshapes the cost arithmetic differently, and their interaction determines whether a delegation tree with ten thousand leaves can operate within an overhead ceiling of, say, two percent of the root budget rather than two hundred percent.

The cryptographic plumbing already works. The question this section answers is whether it can afford to run.

Transaction Batching and Settlement Aggregation: Amortizing On-Chain Costs Across Hundreds of Micro-Settlements

Kira's child agents fire off signed state-channel updates every time they purchase an inference token from a downstream provider. Each update costs nothing off-chain: a counter-signed balance revision on a bilateral channel, cryptographically binding but invisible to any ledger. The problem arrives at settlement. If every one of those sub-cent commitments must individually touch even a cheap L2 rollup, the gas overhead per settlement dwarfs the payment itself. A 0.002-cent inference token purchase that incurs around 0.01 cents in L2 calldata cost is not a micropayment system; it is a fee-subsidy program for sequencers. The budget DAG already constrains *who* may spend *how much*. The missing piece is a cost-aware policy for *where* and *when* those spends become final.

The decision breaks along three transaction classes, each mapped to a settlement layer by a cost-latency matrix embedded in the aggregator node's policy. Sub-cent flows, the individual inference token purchases that constitute the vast majority of Kira's child-agent activity, never leave the state channel during normal operation. Each child agent accumulates hash-locked payment commitments locally, signing incremental balance updates against its counterparty. The channel's latest state is the cryptographic record of every micro-settlement, and no chain sees it un-

til dispute or scheduled rollup. Medium-value session settlements, typically the net balance of dozens to hundreds of micro-commitments accumulated over a configurable window, batch into a single L2 rollup submission. The aggregator node collects the latest signed channel states from a cohort of child agents, computes a Merkle root over their cumulative spend commitments, and posts that root plus the compressed calldata in one L2 transaction. Large epoch reconciliations, representing the periodic anchoring of the entire budget tree's net position, touch L1. This is the only moment the root chain validates the aggregate state, and it happens infrequently enough that even elevated L1 gas costs are amortized across thousands of underlying micro-settlements.

The batching mechanism inside the budget DAG preserves the cryptographic ceiling invariant established by the parent principal's macaroon delegation. Each child agent's signed state-channel update includes a cumulative-spend field: the running total of value that agent has committed since its credential was issued. When the aggregator constructs the batch Merkle tree, every leaf contains this cumulative-spend commitment alongside the channel's latest balance proof. The tree's construction enforces an arithmetic constraint. The sum of all leaf-level cumulative spends must not exceed the cap encoded in the parent macaroon's attenuation caveat. If a compromised or buggy child agent attempts to overstate its spend, the Merkle proof fails verification at the L2 contract, and the entire batch is rejected. The invariant holds: total on-chain committed spend never breaches the root budget, because the proof structure makes breach algebraically impossible to submit.

Dynamic layer selection replaces static configuration for choosing when batches roll up. The aggregator monitors two trigger conditions and fires on whichever threshold hits first. A time-based trigger, set at N seconds, ensures latency guarantees for providers awaiting settlement finality. A value-based trigger, set at M cents of accumulated net balance, ensures that batches contain enough economic weight to justify the L2 submission cost. Fee-market volatility on both L2 and L1 feeds into the selection logic as well. When L2 sequencer fees spike above a configurable ceiling, the aggregator extends N, buffering more micro-commitments in the state channel layer before posting. When L1 base fees drop below an epoch-anchoring threshold, the aggregator opportunistically submits the epoch reconciliation early, locking in cheaper finality. The policy parameters themselves are first-class fields in the budget DAG node's configuration, governed by the same recursive delegation rules that control spend caps. A parent principal can attenuate not only how

much a child may spend but how aggressively it settles, binding cost strategy to authorization.

The result reshapes Kira's operational economics. Before batching, each child agent's L2 submission burned gas that could exceed the sub-cent payment value by a factor of five, making small-provider delegation a net loss. After introducing the aggregator with dual-threshold batch policy, roughly two hundred child-channel state updates compress into a single L2 calldata submission, cutting per-settlement gas cost by approximately two orders of magnitude. The Merkle proof guarantees the root budget ceiling at every layer. And the settlement choices made here, which channels stay open, which batches post to L2, which epochs anchor on L1, directly determine the cost and speed of unwinding when a child agent's credential must be revoked or a batched settlement faces a dispute.

The Cost-Latency-Finality Surface: Parameterizing Settlement Layer Selection Across L1, Rollup, and State Channel Regimes

The hum of gas metering on a congested L1 is unmistakable to any agent watching mempool dynamics in real time: fee spikes ripple through pending transactions like pressure waves, turning what should be a sub-cent GPU-slot payment into an expenditure that dwarfs the compute it purchases. Every node in Kira's budget DAG faces this pressure continuously, and the settlement layer it selects for any given payment is not an infrastructure accident. It is a policy-enforced decision, parameterized across three dimensions that form a continuous surface: cost per finalized byte, latency to irreversibility, and the strength of finality guarantee the parent principal demands. Understanding how these three regimes compare, and how the comparison gets encoded as a cryptographic constraint inside the delegation tree, is the mechanism that transforms gas optimization from an ad-hoc engineering trick into a first-class property of autonomous spending.

Consider the three regimes as bands on that surface. At one extreme, a state channel between two counterparties settles signed balance updates in single-digit milliseconds at effectively zero marginal gas cost per update, because no on-chain transaction occurs until the channel closes or a dispute arises. The finality guarantee, however, is conditional: it holds only as long as both parties remain online to contest fraudulent close attempts within the dispute window $\Delta t_{dispute}$. At the opposite extreme, an L1 anchor transaction achieves the strongest finality the base chain offers, roughly on the order of tens of minutes for probabil-

istic finality on proof-of-work chains or a single epoch boundary on proof-of-stake networks, but each byte of calldata carries the full prevailing gas price. Between them, an L2 rollup batches many state transitions into a single L1 commitment. A validity rollup posts a succinct proof (e.g., a SNARK or STARK) whose on-chain verification cost is amortized across every transaction in the batch, while an optimistic rollup posts the batch and defers verification to a challenge window. The per-payment gas overhead on a well-utilized rollup drops by roughly two orders of magnitude relative to direct L1 settlement, and latency to soft confirmation is typically seconds rather than minutes, though final L1 anchoring lags behind by the rollup's proof-submission or challenge period.

What makes Kira's architecture distinct is that the selection among these three bands is not left to runtime heuristics external to the budget DAG. Instead, the parent encodes settlement-tier caveats directly into the macaroon it delegates to each child agent. A caveat of the form `settlement_tier ∈ {state_channel, L2_rollup}` with an accompanying `max_latency_ms ≤ 5000` and `max_gas_fraction ≤ 0.003` compels the child to evaluate real-time gas oracle readings against these bounds before committing any settlement. If the child's pending payment falls below a threshold where even L2 per-transaction overhead exceeds the 0.3% gas-fraction ceiling, the child must accumulate that payment as a signed state-channel update and defer on-chain settlement entirely. If the payment crosses into a range where the L2 batch amortization brings overhead within budget, the child submits a hash-locked payment proof to the sibling-aggregator. The aggregator, itself a node in the DAG with its own attenuated macaroon, collects proofs from multiple siblings, verifies that the Merkle root of batch totals reconciles against each child's remaining sub-budget, and posts a single L2 commit transaction. One on-chain footprint replaces what would have been three or more independent submissions.

The aggregation step deserves careful attention because it is where per-child audit trails survive compression. Each child's payment proof includes a commitment $C_i = H(\text{agent\_id} \parallel \text{amount}_i \parallel \text{nonce}_i)$, and the aggregator constructs a Merkle tree over all $\{C_i\}$. The Merkle root enters the L2 batch, and each child retains its authentication path as a compact proof that its specific settlement was included. The parent can verify any child's contribution by checking the path against the published root, without replaying the entire batch. This preserves the budget DAG's invariant: the sum of all child settlements plus the ag-

gregator's gas expenditure must never exceed the root-allocated ceiling, and every satoshi is individually attributable.

L1 settlement enters the picture only at epoch boundaries or when a dispute triggers escalation. A child whose state-channel counterparty goes silent invokes the on-chain dispute path, and the budget DAG's policy permits L1 gas expenditure specifically for this contingency by carrying a caveat like `L1_permitted_for {dispute, epoch_finalize}`. The cost is high, but the finality is unconditional, and the budget DAG accounts for it as a reserved gas escrow deducted from the child's ceiling at delegation time. In Kira's before-and-after, the shift from independent per-child L2 submissions to aggregated batching with state-channel deferral compressed gas overhead from around 8–12% of delegated spend to below 0.3%, while every settlement remained cryptographically attributable within the DAG. The surface is not merely descriptive. It is the enforceable terrain on which every autonomous spending decision lands, and any deviation from its contours becomes immediately visible to the mechanisms that watch for it.

An Orchestrator Agent Routing Ten Thousand Inference Payments Through Optimal Settlement Layers in Real Time

Eleven thousand inference requests flood Kira's routing queue in a nine-second burst, each carrying a sub-cent payment obligation to one of fourteen GPU providers scattered across three rollups and an L1 settlement layer. Kira's budget DAG, constructed earlier in this chapter, has already delegated spending authority to three child agents, each holding an attenuated macaroon whose caveats encode a cumulative spend ceiling, a maximum per-transaction value, and a settlement-delay tolerance expressed in blocks. The question confronting Kira's policy engine is not whether these payments can be made but where each one should finalize, because the difference between naive per-transaction L1 settlement and optimized tier selection determines whether the sub-budget survives the burst or bleeds out in gas.

Each child agent's policy node carries a first-class batching directive: a tuple of parameters specifying a batch-size threshold (the minimum number of off-chain balance updates that must accumulate before triggering a settlement write), a maximum settlement delay (expressed as a block-height delta from the oldest unsettled update), and a cumulative gas ceiling (the maximum wei the child may spend on settlement mechanics across all its transactions within the current delegation epoch). These three values are not advisory. They are caveats baked into the del-

egated macaroon, cryptographically bound to the child's credential chain. A child agent that attempts to close a channel with fewer accumulated updates than its batch threshold will produce a settlement transaction whose on-chain verifier rejects the caveat proof. The parent's policy node set Kira's three children to batch thresholds of 200, 400, and 800 respectively, calibrated to the latency SLA each child's upstream callers will tolerate. The child serving real-time voice transcription batches at 200 because its callers demand settlement confirmation within roughly fifteen seconds. The child handling asynchronous document summarization batches at 800, accumulating obligations for up to two minutes before committing.

The settlement-path selection function fires on every payment. It ingests four inputs: the current gas price feed from L1 and two L2 sequencers, the counterparty trust score derived from the provider reputation system introduced in the service-discovery chapter, the payment magnitude as a fraction of the channel's remaining capacity, and the latency SLA attached to the upstream request. The function evaluates a simple decision tree. If the counterparty trust score exceeds a threshold and the payment is below 0.1% of channel capacity, the child finalizes via direct state-channel update with no on-chain footprint at all, deferring settlement to the next batch close. If the L2 gas price is below a configurable fraction of the L1 price (Kira's current policy sets this at 0.08x) and the latency SLA permits the rollup's confirmation window, the child queues the close-out for L2 batch submission. Only when the payment magnitude is large relative to channel capacity or the counterparty trust score is low does the function route to L1 for immediate finality. During the burst, roughly 94% of payments resolve as pure state-channel updates, around 5% queue for L2, and fewer than 1% demand L1 anchoring.

The payoff crystallizes when Kira's three children finish their parallel work and their batch timers expire within moments of each other. Instead of three independent L2 submissions, Kira's parent policy node detects that all three target the same rollup sequencer. It constructs a single Merkle tree whose leaves are the individual batch commitment hashes from each child, computes the root, and submits one aggregated transaction to the L2 contract. The contract verifies the Merkle root against each child's declared spend total, confirms that the sum does not exceed the parent's delegated sub-budget ceiling, and writes a single storage slot. Before aggregation, three separate L2 commits would have consumed an estimated 40% of the sub-budget in gas. After aggregation, the

single commit costs roughly 9%, because calldata compression and shared proof verification amortize the fixed overhead across all three children. The budget DAG invariant, total child spend less than or equal to parent allocation, is verified on-chain in that one step. The Merkle root itself becomes a compact evidence artifact: if any child's spend is later disputed, the dispute layer can reconstruct the relevant branch from the aggregated proof without replaying individual transactions.

What emerges from this burst is not just cost savings but a structural guarantee. Gas optimization is no longer an operational afterthought layered on top of the budget hierarchy. It lives inside the policy nodes as enforceable caveats, shaping settlement behavior the same way spend ceilings shape payment magnitude. The Merkle-aggregated commitment that closes out a batch carries within it the tier tag of every settlement path chosen, giving downstream verification compact, auditable proof of how each payment resolved. That proof structure, tier-tagged and branch-addressable, is precisely the artifact the dispute-resolution layer needs to adjudicate conflicts without decompressing the entire transaction history.

The flat credential was always the bottleneck, not the payment rail. A single macaroon handed to a single agent is adequate for a demo and disastrous for production, because it compresses every spending decision into one unstructured authorization with no internal governance. What this chapter constructed is the missing dimension: depth. A root envelope denominates total liability in satoshis. Recursive attenuation mints child credentials whose cumulative ceilings sum to at most the parent's remaining balance, enforced not by a database row an operator must monitor but by HMAC-chained caveats that no node in the tree can strip, inflate, or forge. Policy checks are local cryptographic operations measured in microseconds, settlement aggregates upward to L1 only at epoch boundaries, and revocation propagates downward through the DAG without requiring on-chain state changes at every leaf. Kira's child inference workers do not request spending permission at runtime. They carry permission inside their credentials, scoped across five enforceable dimensions, with exhaustion triggering a graceful fallback to a cheaper model tier rather than a silent failure or a hard stop. The hierarchy itself is the governance layer.

Put this to a concrete test. Design a three-level budget tree for your own system: root principal, orchestrating agent, two or more leaf workers. Specify per-transaction cap, cumulative ceiling, provider whitelist,

time-to-live, and exhaustion action at every node. Then calculate worst-case total spend if every leaf exhausts its ceiling simultaneously.

Chapter Ten

Full-Stack Agent Commerce Topology in Operation

According to the Lightning Network's own capacity tracking, the public network processes off-chain state updates that settle in under a second while the marginal cost per update rounds to zero. That single empirical fact underpins everything that follows. Because Kira has never needed every layer of her stack to fire in one contiguous sequence until now.

At 14:23:07 UTC, a 70-billion-parameter summarization job lands in her inbound queue. The constraints are tight: a 400-millisecond latency ceiling, a maximum spend of 1,100 satoshis, and a provider she has never transacted with before. What happens next must complete in roughly 1.8 seconds. She queries a Kademlia DHT for candidate providers, runs a sealed-bid auction against three respondents, opens a state channel with the winner, authenticates via an L402 credential, streams per-inference micropayments against a cryptographic budget envelope, collects a signed receipt, and settles the channel cooperatively. No human approves any step. Every layer this book has constructed in isolation, from hash-locked commitments through attenuated macaroon delegation through dispute-timeout calibration, now executes as a single integrated pipeline. If any layer fails, the job fails.

But none of them do. To see why, we need to follow this job from the moment it enters Kira's queue to the moment the last satoshi settles, tracing every message, every signature, and every state transition in the order they actually fire.

End-to-End Walkthrough: DHT Discovery, Auction Negotiation, State Channel Opening, and L402 Authentication for Autonomous Compute Acquisition

Roughly 7 in 10 distributed systems failures manifest not in individual components but at integration boundaries, where independently correct modules collide under shared timing constraints. That ratio holds here. Every cryptographic primitive Kira requires has been specified, verified against its threat model, and shown to be deployable on production infrastructure. The open question is whether a DHT lookup, a sealed-bid auction, a channel funding exchange, and an L402 challenge-response can execute as a single contiguous operation within the roughly 1,800 milliseconds an upstream caller will wait before treating the inference request as lost.

The clock starts now. Kira's resource monitor flags that an inbound inference job exceeds locally available GPU capacity by a factor of three. From this instant, every protocol handshake competes for the same scarce budget of wall-clock time. Kira must query the DHT for providers matching the workload's resource profile, evaluate bids against its budget policy envelope, construct and countersign a funding transaction with the winning provider, present an L402 macaroon scoped to the session, stream micropayments per inference chunk, and handle the possibility that a provider disappears mid-stream, forcing a fallback onto a standby channel whose funding must not block the failed channel's force-close. That final contingency is where the real proof obligation lives: concurrent conflicting commitment states that only resolve correctly if the budget hierarchy's revocation logic holds under adversarial timing. What follows traces each state transition in order, with the cryptographic operation named, its latency cost noted, and the remaining time budget decremented at every step.

Kademlia Lookup Sequence: From Capability Hash Query to Ranked Provider Shortlist with Cryptographic Identity Verification

Roughly 7 in 10 Kademlia lookups converge on the target key's neighborhood within four iterative rounds when the routing table holds on

the order of a hundred peers. That convergence property is not trivia. It is the reason Kira can transform a capability hash into a ranked provider shortlist in under two seconds of wall-clock time, and it is where the full-stack commerce cycle begins. Every layer that follows, auction negotiation, state-channel funding, L402 credential exchange, job execution, dispute resolution, depends on this opening move completing quickly and delivering cryptographically verified results. Discovery is load-bearing infrastructure, not preamble.

Kira constructs the lookup by hashing its capability requirement, a structured descriptor specifying GPU memory floor, supported model formats, and maximum per-token price, into a 256-bit key K_cap. It seeds the FIND_NODE query with the α closest peers from its own routing table, dispatching parallel requests that return k-buckets progressively nearer to K_cap in XOR-distance. Each responding node signs its reply with its long-term Ed25519 identity key, binding the returned peer list to a verifiable origin. Kira discards any response whose signature fails verification or whose node ID does not fall within the expected Kademlia subtree, eliminating eclipse-attack vectors at the routing layer before a single satoshi is at risk. The iterative narrowing proceeds until the lookup stabilizes: the k closest nodes to K_cap have all been queried, and no new closer node has appeared.

Providers that store capability advertisements under K_cap respond not merely with contact information but with signed capacity attestations. Each attestation contains the provider's current slot availability, supported precision modes, a quoted price range, and a freshness timestamp, all wrapped in a signature σ(attestation, sk_provider) that Kira can verify against the provider's public key retrieved during the lookup itself. This is the critical difference from a centralized registry: the attestation is self-certifying. No directory authority vouches for it. Kira verifies it against the provider's own cryptographic identity, and the provider's identity is anchored to its node ID through the hash-locked Kademlia address structure. Forgery requires producing a valid signature under a key whose hash maps to the correct region of the DHT keyspace, a computationally intractable task under standard discrete-log assumptions.

With attestations collected, Kira applies its spending policy, the delegated budget hierarchy established in Chapter 9, as a hard filter. Any provider whose minimum quoted price exceeds the per-task ceiling encoded in Kira's macaroon caveat chain is eliminated before ranking begins. The remaining candidates are scored on a composite metric: XOR-

distance proximity (a proxy for routing reliability), attestation freshness (stale advertisements suggest overloaded or offline nodes), historical settlement success rate drawn from Kira's local reputation cache, and quoted price. The output is an ordered shortlist, typically three to five providers, each backed by a verified identity and a capacity claim Kira can hold them to during auction negotiation.

This shortlist is not advisory. It is a binding input to the sealed-bid auction protocol that fires next. Every entry carries the provider's public key, verified attestation, and the XOR-distance metadata Kira needs to estimate round-trip latency for state-channel message exchange. The lookup has consumed no funds, opened no channels, and required no human intervention. It has, however, established the cryptographic identity anchors that make every subsequent step, channel co-signing, invoice settlement, dispute escalation, trust-free by construction. Kira's routing table updates with the freshly contacted peers, improving future lookups through the self-healing property of Kademlia's bucket-refresh protocol.

What surfaces only at integration time is the latency dependency between discovery and auction. The sealed-bid window is short by design, typically a few hundred milliseconds, to prevent provider-side information leakage. If the Kademlia lookup takes three iterative rounds instead of four, Kira gains an extra round-trip's worth of margin to evaluate attestations and construct its bid. If lookup latency spikes because a routing-table partition forces five rounds, the auction window may close before Kira submits, forcing a re-discovery cycle and burning the budget headroom allocated for retry. This coupling between DHT convergence speed and auction timing is invisible when the layers are tested in isolation. It is the first emergent tradeoff the full stack exposes, and Kira's bidding strategy must price it in as a stochastic cost on every compute acquisition attempt.

Sealed-Bid Auction Entry, Channel Funding Transaction Construction, and the Race Between Negotiation Latency and Block Confirmation

Kira's orchestrator thread fires a capability query into the Kademlia DHT at timestamp t_0. Within the next 500 milliseconds, the agent must discover a qualified GPU provider, win a sealed-bid auction, open a funded state channel, and present an L402-authenticated inference request — or forfeit its upstream SLA commitment. Every cryptographic handshake in that window either collapses a trust assumption or enforces a

timeout boundary. This walkthrough traces the exact message sequence, byte by byte, showing how auction entry, channel funding, and credential exchange compose into a single atomic acquisition cycle where negotiation latency races against block confirmation. You have built each layer individually across prior chapters. Now you wire them together under a clock that does not forgive slack.

Step 1: Publish the DHT Capability Query and Filter Provider Advertisements

Kira constructs a `CapabilityQuery` message specifying the required resource class (e.g., `gpu.a100.fp16`, minimum 40 GB VRAM, inference-mode attestation), signs it with its delegation key sk_d, and publishes it to the k-closest nodes in the Kademlia routing table. The DHT returns a ranked list of `ProviderAdvertisement` records, each containing a posted price curve $P(t)$, a freshness timestamp, and a remote attestation proof σ_att binding the provider's hardware identity to its network key. Kira filters this list against its delegated budget policy — any advertisement whose minimum committed price exceeds the policy ceiling or whose attestation proof fails verification against the known manufacturer root is discarded immediately. The entire discovery phase operates within a 200 ms window starting at t_0. If fewer than n_min qualifying providers surface before the deadline, Kira falls back to its local provider-list cache, populated from the previous successful DHT sweep. This fallback is not optional — it is the difference between a graceful degradation and a hard SLA breach.

1. Serialize the `CapabilityQuery` with resource class, VRAM floor, attestation type, and `sig(sk_d, query_hash)`.
2. Dispatch to k-closest DHT nodes via parallel iterative lookup; collect `ProviderAdvertisement` responses.
3. Verify each σ_att against the manufacturer attestation root and discard any record with an expired freshness timestamp.
4. Rank surviving advertisements by price-curve gradient at the anticipated session length and trim to the top m candidates.
5. If $|\text{candidates}| < n\_min$ at $t_0 + 200$ ms, load cached provider list and merge with any partial DHT results.

Step 2: Submit the Sealed-Bid Auction Entry with Embedded Funding Proof

Kira selects its top-ranked provider and constructs a `SealedBid` message. This message carries three payloads: (1) the bid amount b encrypted under the auction coordinator's public key, (2) a funding proof consisting of an unspent transaction output (UTXO) reference and a partial signature committing Kira's deposit to a 2-of-2 multisig output, and (3) a commitment hash $H(nonce \parallel b)$ that binds the bid value without revealing it until the reveal phase. The critical design insight is that this single message serves double duty — it is simultaneously an auction entry and one half of a channel-opening commitment. No separate funding transaction broadcast is required. The auction round targets completion within 150 ms. The provider, upon receiving the sealed bid and verifying the UTXO reference against its mempool view, countersigns the 2-of-2 multisig output. This countersignature is the provider's acceptance signal. The resulting signature pair $(\sigma_{Kira}, \sigma_{provider})$ constitutes both the auction settlement and the opening commitment for a bilateral payment channel. Two protocol rounds collapse into one cryptographic event.

1. Construct the `SealedBid` envelope: `Enc(pk_auction, b) ‖ UTXO_ref ‖ partial_sig(sk_Kira, funding_tx) ‖ H(nonce ‖ b)`.
2. Transmit to the selected provider's auction endpoint over the authenticated session established during DHT discovery.
3. Provider verifies UTXO liveness, checks that the deposit meets its minimum channel capacity, and countersigns: `σ_provider = sig(sk_provider, funding_tx)`.
4. Both parties exchange the complete signature set; the funding transaction is now valid but not yet broadcast.

Step 3: Broadcast the Funding Transaction and Open the State Channel

With both signatures in hand, Kira broadcasts the funding transaction to the mempool. The channel is considered *optimistically open* as soon as both parties hold the countersigned commitment — actual on-chain confirmation is the dispute-resolution backstop, not the operational gate. The provider begins accepting off-chain balance updates immediately, relying on the signed commitment as its enforcement primitive. The channel-open phase budgets 100 ms for signature exchange verification, local state initialization, and the first commitment update. Block confirmation, which may take minutes, proceeds in the background. If confirmation does not arrive within *⊿t_timeout* blocks, the pre-signed reclaim transaction becomes valid, and Kira recovers its deposit without provider cooperation. This is the precise boundary where negotiation latency separates from confirmation latency — the protocol does not wait for the chain.

1. Broadcast `funding_tx` with `(σ_Kira, σ_provider)` to the mempool.
2. Both parties initialize local channel state: balance allocation `(deposit_Kira, 0)`, sequence number `0`, and revocation seeds.
3. Provider marks the channel as *optimistically active* and enables the L402 challenge endpoint.
4. Kira stores the pre-signed timelock reclaim transaction `tx_reclaim` with `nLockTime = current_height + Δt_timeout`.

Step 4: Settle the L402 Challenge and Authenticate the First Inference Request

The provider issues an HTTP `402 Payment Required` response carrying a macaroon with an embedded HTLC caveat. The caveat specifies a payment hash *H(r)* and a minimum channel balance decrement *δ*. Kira locates the preimage *r* by settling the HTLC within the open channel — it constructs a new commitment update decrementing its balance by *δ*, attaches the preimage reveal, and sends the signed update to the provider. The provider verifies that *H(r)* matches, that the balance update is correctly signed, and that the channel's remaining capacity covers future requests. Once verified, Kira attaches the discharged macaroon to its inference request's `Authorization` header. The provider validates the macaroon's signature chain, confirms the attenuation caveats (resource class, session scope, expiry), and checks the channel balance atomically. Only then does it release the first GPU timeslice. The entire L402 exchange completes within 50 ms. Kira's total elapsed time from DHT query to authenticated compute delivery sits at or below the 500 ms envelope — the upstream SLA holds.

1. Receive the `402` response; parse the macaroon and extract `H(r)` and `δ` from the HTLC caveat.
2. Construct commitment update `C_{n+1}` with `balance_Kira -= δ`, attach preimage `r`, and sign: `σ_update = sig(sk_Kira, C_{n+1})`.
3. Provider verifies `SHA256(r) == H(r)`, validates `σ_update`, and countersigns `C_{n+1}`.
4. Kira attaches the discharged macaroon to the `Authorization: L402` header and submits the inference payload.
5. Provider verifies macaroon signature chain, evaluates attenuation caveats, confirms `balance_Kira ≥ 0`, and releases the GPU timeslice.

Kira has now executed a complete, zero-human-touch compute acquisition cycle. From DHT query to authenticated GPU timeslice delivery, every cryptographic message is accounted for, every signature verified, every timeout bounded. The sealed-bid auction and channel-funding commitment collapsed into a single signature exchange. The L402 credential gated the first payload without introducing a trust dependency outside the bilateral channel. The total latency stayed within the 500 ms SLA envelope. What remains is to stress-test this integrated sequence against adversarial conditions and quantify exactly where each

failure mode costs time and capital — the settlement-latency tradeoff surface that the next section maps in full.

L402 Challenge-Response Handshake: Binding Lightning Invoice Settlement to Macaroon-Scoped API Session Initialization

Settling a Lightning invoice before a single API byte flows back feels paradoxical only until you trace the cryptographic binding that makes the two events atomic. Kira has already won the auction, co-signed the state-channel funding transaction, and watched it confirm. The channel is open. A funded commitment output sits on-chain, encoding the maximum session spend, the HTLC timeout delta, and the dispute window. Now the provider's compute endpoint returns HTTP 402, and the real integration test begins.

The 402 response carries two objects: a Lightning invoice encoding a payment hash H(r) for the session's first inference slot, and a macaroon whose root key is held by the provider. The macaroon's caveats are not decorative. They bind the bearer credential to Kira's channel funding outpoint, to a maximum per-call spend denominated in millisatoshis, and to an expiry timestamp derived from the channel's HTLC timeout. Kira inspects each caveat against its own delegated budget envelope, the same hierarchical spending policy it received from its parent principal in the budget-delegation layer. If the macaroon's per-call ceiling exceeds the remaining sub-budget, Kira rejects the session before any satoshi moves. This check costs nothing. It is pure local computation against a signed policy object, exactly the kind of decision that was impossible when Kira's spending authority lived in a human-configured AWS billing console.

Kira pays the invoice. Its Lightning node routes the HTLC through the path the channel graph provides, the preimage r propagates back on settlement, and Kira now possesses the cryptographic receipt that unlocks the macaroon's validity. It attaches the macaroon as a bearer token to an HTTP request carrying the first inference batch. The provider verifies the macaroon's HMAC chain, confirms the preimage matches the invoice's payment hash, and serves the response. Total elapsed time from 402 challenge to first inference result sits in the range of roughly 200 to 400 milliseconds, dominated by the HTLC round-trip rather than any authentication overhead. Before this moment, Kira had proven each capability alone: hash-locked proofs, channel funding, multi-hop routing, credential scoping, discovery, bidding, delegation. Now every layer fires in sequence within a single HTTP exchange.

Each subsequent inference call tightens the loop further. The provider issues an updated macaroon with an attenuated caveat reflecting the decremented balance, and Kira signs a new off-chain state update that simultaneously advances the channel's balance partition and rotates the authentication token. No on-chain transaction occurs. The state-channel commitment output absorbs every incremental payment as a simple counter update in a co-signed message pair, while the rotating macaroon ensures that a captured token from call N cannot authorize call N+1. Kira processes roughly 40 inference batches in under 15 seconds of wall time, each batch paying between 50 and 200 millisatoshis depending on the provider's posted price curve. The budget hierarchy's spend counter increments with every signed state, and Kira's internal ledger stays perfectly synchronized with the channel's latest commitment because they are the same data structure viewed from two angles.

When the task queue drains, Kira and the provider exchange final signed states encoding the closing balance split. They broadcast the cooperative close as a single on-chain transaction. The channel's entire lifecycle, from funding to settlement, leaves exactly two transactions on the base layer: one open, one close. Kira logs the settlement proof, a signed tuple of final balances plus the closing transaction ID, back to its parent policy ledger. The budget hierarchy now reflects exact spend down to the millisatoshi. Compare this to Kira's starting condition: a stateless router tethered to a human-administered billing account, unable to switch providers without a manual credential rotation, unable to verify that a dollar spent bought a dollar of compute. That agent is gone. What remains is a machine that discovered a provider it had never seen, won a competitive auction, opened a trust-minimized channel, authenticated through a pay-gated credential loop, executed a full workload at machine speed, and settled cooperatively, all without a single human approval step. The happy path composes cleanly. Every layer locks into the next with cryptographic precision, leaving behind exactly the branch-points where failure modes will surface.

Per-Inference Micropayment Execution, Cryptographic Receipt Generation, and Atomic Settlement Across the Complete Stack

Roughly seven out of ten Lightning Network payment attempts on well-connected nodes now settle in under 750 milliseconds. That number matters because a single forward pass through a 70-billion-parameter

model on current-generation GPU hardware completes in a comparable window, meaning the economic protocol and the compute it pays for race on the same clock. If settlement cannot finish inside that latency envelope, payment becomes a bottleneck that throttles inference throughput, and the entire stack collapses into a queuing problem no amount of architectural elegance can salvage.

Kira's channel is open. The L402 macaroon is scoped, the budget envelope is loaded, and the provider's endpoint has been authenticated. Nothing has moved yet. The next packet Kira transmits carries a prompt payload bound to a hash commitment, and the response that returns must interleave an inference result with a cryptographic receipt so tightly coupled that neither party can cheat without forfeiting value. No mediator stands between them. No escrow contract buffers the exchange. The preimage that unlocks payment and the output that satisfies the request are entangled at the protocol level, and atomicity emerges from that entanglement alone.

What follows traces the complete lifecycle of that single round-trip. From the signed balance update that commits funds at dispatch, through the HMAC-chained receipt that binds delivery proof to a specific output hash, to the HTLC conditionality that guarantees payment and delivery either both complete or both revert, every layer must hold under adversarial assumptions. This is where twelve chapters of mechanism design reduce to one verifiable exchange, and where the claim that machines can transact autonomously at inference granularity either proves out in deployed primitives or fails on the wire.

Signed Balance Update per Inference Call: State Transition Mechanics from Request Dispatch Through Payment Commitment

Roughly seven in ten inference calls in a multi-hop agent topology touch at least three independent balance updates: the caller-to-broker channel, the broker-to-provider channel, and the provider's internal accounting ledger. When any one of those updates settles independently of the others, a temporal gap opens, and inside that gap lives credit risk. The entire purpose of the atomic sequence described here is to collapse those three updates into a single logical transition with one outcome: all legs commit, or none do.

Kira's current state makes this concrete. It discovers providers, wins auction slots, delegates sub-budgets to child agents, and authenticates via L402 bearer credentials. But until the payment for a single inference

call is fused to the delivery of that inference result in one indivisible operation, every hop in the chain carries a moment of unhedged exposure. The construction that eliminates this exposure begins the instant Kira dispatches a request to Provider-7.

At request dispatch, Kira constructs a conditional commitment on its downstream channel. The commitment locks a precise amount, say 11 satoshis for one inference round-trip, behind a hash lock H(r) where r is a preimage known only to Provider-7's attestation module. This is the HTLC-like conditional: Kira's channel balance decreases by 11 satoshis in a pending state, and Provider-7 can claim those funds only by revealing r. A timeout Δt_timeout bounds the lock. If Provider-7 fails to reveal r before the timeout expires, the funds revert automatically to Kira's available balance. No custodian adjudicates this. The channel's 2-of-2 multisig enforcement and pre-signed timeout transactions guarantee reversion on-chain if necessary.

Provider-7 receives the locked commitment, executes the inference, and now does something critical: it generates a cryptographic receipt. This receipt is a signed attestation σ(m, sk_provider) binding four objects together. The inference output hash H(output), the payment preimage r, a monotonic timestamp t, and the new channel state index n. By signing this composite, Provider-7 creates a non-repudiable proof that it delivered a specific result at a specific time and that the payment preimage it is about to reveal corresponds to that exact delivery. Kira can store this receipt indefinitely. It is verifiable by any third party holding Provider-7's public key, and it serves as the complete audit trail for this atomic unit of commerce.

The moment Kira receives the valid receipt containing r, it verifies H(r) against the hash lock in the pending HTLC. Match confirmed, the downstream channel state advances: Kira's balance decreases by 11 satoshis, Provider-7's balance increases by 11 satoshis, and both parties co-sign the updated state at index n. This is settlement. Not eventual settlement, not batched settlement. Immediate, bilateral, final.

But the downstream leg is only half the sequence. Kira simultaneously executes the upstream mirror. Its own channel with the calling agent held a symmetric conditional lock. The caller had committed funds to Kira behind the same hash lock H(r), or behind a chained hash lock in a multi-hop construction where Kira's possession of r cascades backward. Now Kira reveals r to the caller's channel, the upstream HTLC resolves, and the caller's balance decreases while Kira's upstream balance increases. Both channel states advance atomically. The caller

paid for the inference. Kira collected its margin. Provider-7 collected its compute fee. Three balance updates, one logical transaction, zero moments of unhedged exposure.

The atomicity guarantee holds because every commitment is conditional on the same preimage. If Provider-7 never delivers a valid result, r never surfaces, all HTLCs expire, and every balance reverts to its prior state. If Provider-7 delivers but Kira attempts to withhold r from the upstream caller, the caller's timeout triggers reversion on that leg while Provider-7 still claims the downstream payment, punishing Kira's dishonesty through direct financial loss. The incentive structure makes honest forwarding the dominant strategy without requiring any party to trust any other party's good behavior.

This is the moment the full stack proves its coherence. Discovery found Provider-7. The auction won the slot. The budget hierarchy authorized the spend. The L402 credential authenticated the request. And now the signed balance update per inference call fuses payment, proof, and settlement into one cryptographic instant. What remains is to characterize what happens when this instant fractures, when timeouts expire asymmetrically, when providers vanish mid-computation, when channel states disagree.

Receipt Construction: HMAC-Chained Proof-of-Delivery Binding Preimage, Timestamp, and Inference Output Hash into a Verifiable Attestation

Kira's routing engine selects a GPU provider, constructs a conditional payment, and fires the inference request down a single channel hop. At that instant, three protocol objects begin crystallizing into the artifact that makes the entire stack trustless at single-call granularity: the HTLC hash preimage committed by the provider, a timestamp binding the exchange to a narrow temporal window, and a cryptographic digest of the inference output itself. The receipt that emerges from their composition is not a log entry or a database row. It is a verifiable attestation, an HMAC-chained proof-of-delivery that binds payment to compute output with no arbiter, no escrow service, and no trust assumption beyond the hardness of the hash function.

The payment flow initiates when the caller presents a valid L402 bearer token to Kira, whose budget-delegation caveats have already scoped the maximum per-inference spend. Kira constructs an outbound HTLC to the selected provider, locking funds against a hash H(r) where r is a preimage known only to the provider. The provider cannot claim

those funds until it reveals r, and it commits to reveal r only upon completing the inference and producing a signed output digest σ(d, sk_provider), where d = SHA-256(inference_output). This construction fuses payment conditionality to compute delivery at the cryptographic level. The provider's rational strategy is unambiguous: complete the inference, sign the digest, reveal the preimage, and collect the payment in a single atomic step. Any deviation, whether withholding r, delivering a garbage output, or attempting to claim payment without producing a valid signature over the correct digest, leaves the HTLC to expire and the funds to revert to Kira's channel balance after the timeout Δt_timeout.

Upon preimage reveal, the settlement cascade propagates back up the stack. Kira now holds r, which it uses to settle the inbound HTLC from the caller. Each hop in the three-party chain, caller→Kira→provider, resolves independently but conditionally. The timeout at each hop decrements: the provider's HTLC expires at Δt, Kira's inbound HTLC from the caller expires at $\Delta t + \delta$, where δ provides Kira sufficient time to claim the inbound payment after learning r from the provider. This decrement structure guarantees that no intermediate party can be caught holding an expired outbound lock while its inbound lock remains claimable by an upstream counterparty. Funds never enter a liminal state where one party has paid but another has not settled.

The receipt triple that each participant retains after settlement consists of three bound components: the HTLC hash lock H(r) and its revealed preimage r, proving that the conditional payment resolved; the provider's signature σ(d, sk_provider) over the inference output digest, proving what was delivered and by whom; and the signed channel state update reflecting the new balance, proving that settlement occurred at a specific satoshi amount. Kira computes an HMAC chain linking these elements with the timestamp t_settlement, producing HMAC(k, H(r) ‖ σ(d, sk_provider) ‖ t_settlement) as the final attestation. This chain is non-repudiable. The provider cannot deny having delivered the specific output whose digest it signed. The caller cannot deny having authorized the payment whose budget caveat Kira enforced. Kira cannot fabricate a receipt without possessing both the valid preimage and the provider's signature, neither of which it can produce unilaterally.

Failure at any layer resolves to a safe state without human intervention. If the provider crashes before revealing r, the HTLC times out and funds return to Kira, which never forwarded a settlement claim up-

stream. If Kira crashes after learning r but before settling with the caller, the caller's inbound HTLC also times out safely, and Kira's outbound payment to the provider represents Kira's own operational loss, bounded by the single-inference micropayment amount. If the caller disappears after initiating the request, Kira's inbound channel simply retains its prior state. No scenario produces unreimbursed compute or undelivered payment that persists beyond the timeout window. The atomic bond between preimage revelation and payment settlement eliminates the gap where disputes traditionally fester.

What remains after this single inference cycle is not just a completed transaction but a cryptographic fossil record. Every participant holds locally verifiable proof of exactly what was requested, computed, paid, and settled. The receipt requires no blockchain confirmation, no third-party audit trail, no centralized logging service. It stands on its own cryptographic merit. And it is precisely the structural properties of this receipt, its failure boundaries and its latency-cost profile at each conditional hop, that determine where the next layer of analysis must focus.

Atomic Settlement Guarantees: How HTLC Conditionality and Revocation Penalties Ensure Payment-Delivery Simultaneity Across Layers

Every stack layer described in prior chapters exists to serve one moment: the instant a payment and a computation fuse into a single indivisible event that neither party can split apart, reverse, or corrupt. Two machines exchange value and work simultaneously, or they exchange nothing at all. That is the atomic settlement guarantee. It is not aspirational. It is enforced by cryptographic conditionality at every hop, and penalized by revocation at every channel state. Here is how a single inference request fires the complete stack.

Kira receives an upstream inference request from a calling agent. She has already discovered and scored a GPU provider through the mechanisms established in Chapter 8. She constructs an HTLC whose payment hash H(r) is bound not to an arbitrary secret but to the SHA-256 hash of the expected model output. The provider cannot claim funds without producing a preimage r that, when hashed, matches H(r). That preimage is itself derived from the inference result. Payment and delivery collapse into one cryptographic object. If the provider delivers garbage, the hash will not match, the HTLC times out, and Kira's channel balance reverts. No arbitrator required. No dispute ticket filed.

The provider's response carries a signed receipt. This receipt is a compact, self-proving document: it contains the inference output hash, the revealed preimage r, the updated channel state with sequence number incremented, and a timestamp. The provider's signature σ(receipt, sk_provider) binds all four fields. Kira verifies σ against the provider's known public key, confirms H(r) matches the original HTLC commitment, and validates that the channel state transition is monotonically forward. If any field fails verification, she rejects the receipt and the HTLC expires harmlessly. A valid receipt simultaneously proves delivery and authorizes payment. No third-party oracle. No external attestation service. The receipt is the settlement.

The HTLC chain is not a two-party affair. It propagates across the full stack. The upstream caller locked funds to Kira via its own HTLC, conditioned on the same hash H(r). Kira extended an HTLC of lesser value and shorter timeout to the provider. When the provider reveals r, Kira uses that preimage to claim funds from the upstream caller's HTLC before its longer timeout expires. Each hop's timeout delta, roughly tens of blocks in typical Lightning parameterization, ensures the preimage propagates inward before outer locks expire. The chain settles atomically or unwinds completely. A network partition between Kira and the upstream caller does not strand funds at the provider. The provider's HTLC simply times out, the provider's channel state rolls back, and Kira's budget delegation tree is never debited for undelivered work.

Revocation penalties seal the construction. Every channel state update that Kira and the provider exchange is backed by a revocation key for the prior state. If the provider attempts to broadcast an old, more favorable balance after collecting payment, the revocation key lets Kira sweep the entire channel balance as a penalty. The expected payoff for broadcasting a stale state is negative. The dominant strategy is honest play. This is not an incentive suggestion. It is a Nash equilibrium enforced on-chain.

One request in. One receipt out. One settlement. Kira's delegated budget decrements by exactly the invoice amount. The provider's channel balance increments by the same. The upstream caller's HTLC resolves with the same preimage. Every layer fires. Every layer agrees. No human touched anything. This is the irreducible unit of machine commerce, and it is the baseline against which every failure mode in the next section will be measured.

The Tradeoff Surface: Evaluating Settlement Latency, Cryptographic Security, Capital Efficiency, and On-Chain Footprint Across Protocol Configurations

Roughly seven out of ten Lightning Network channel closures observed in public network data are cooperative, meaning both parties sign off on the final balance and the channel's entire transaction history collapses into a single on-chain output. The remaining three involve unilateral closes, timelock delays, and occasionally the full dispute-resolution machinery of penalty transactions. That ratio matters because it defines the actual on-chain footprint an autonomous agent like Kira should expect under realistic conditions, and it shifts dramatically depending on counterparty reliability, channel lifetime, and payment frequency. A cooperative close after ten thousand routed micropayments costs the same on-chain space as a cooperative close after three. But a single counterparty failure triggers a sequence of timelock-bound transactions that can consume more chain space than the entire prior payment history warranted.

Kira has just completed an unbroken commerce loop, from service discovery through multi-hop HTLC routing through L402 credential presentation through settled delivery. The stack composes. Now a concrete decision surfaces: a new GPU provider offers lower per-token inference cost, but its payment interface demands on-chain HTLC settlement per batch rather than the aggregated off-chain channel updates Kira uses with existing providers. The per-inference price drops, but the settlement latency jumps by an order of magnitude, the on-chain footprint scales linearly with batch count, and the capital locked in the new channel sits idle during each confirmation cycle. Kira's budget-enforcement logic cannot evaluate this offer without a formal model of the space it is optimizing over.

That space has four coupled dimensions: settlement latency, cryptographic security margin, capital efficiency, and on-chain footprint. Pulling any one toward its optimum deforms the others.

Four-Dimensional Parameter Space: Mapping Latency, Security Margin, Locked Capital, and Chain Writes as Coupled Variables

Roughly seven out of ten protocol design conversations collapse a multi-dimensional tradeoff into a single metric. An engineer optimizing for finality speed will accept capital lock-up without noticing. Another engineer minimizing on-chain fees will quietly weaken fraud-proof guarantees. The result is that each team believes it has found the optimal

configuration, when in reality it has selected a single point on a surface whose other dimensions it never measured. The full-stack commerce cycle Kira already executes demands something more principled: an explicit, four-dimensional parameter space in which every protocol configuration occupies a coordinate, and no coordinate dominates all four axes simultaneously.

The four axes require precise definitions. Settlement latency, denoted Δt_settle, measures the elapsed time from HTLC lock to the moment the recipient holds a spendable balance, whether that balance lives in an updated channel state or in a confirmed on-chain UTXO. Cryptographic security margin, S, captures the strength of the fraud-proof and hash-lock assumptions that protect a payment against counterparty defection. It degrades when timeout windows shrink, when hash preimage verification is deferred, or when finality relies on an optimistic assumption that no challenge will be posted. Capital efficiency, η, is the ratio of economically active balance to total locked collateral across all open channels and pending HTLCs. Every satoshi locked in a routing reserve or timeout escrow reduces η. On-chain footprint, F, counts the bytes and fee-weight consumed per unit of economic throughput. A channel that batches ten thousand off-chain updates into a single cooperative close has a radically different F from one that falls back to unilateral settlement after every dispute window.

These four dimensions couple tightly. Shortening Δt_settle by reducing HTLC timeout durations directly compresses the window within which a defrauded party can broadcast a penalty transaction, degrading S. Raising η by minimizing reserve requirements means fewer satoshis are available to cover routing failures or force-close penalties, again lowering S under adversarial conditions. Shrinking F by batching settlements into infrequent on-chain writes increases Δt_settle for the final anchoring step and concentrates risk into fewer, higher-value transactions. No algebraic rearrangement eliminates these tensions. They are structural consequences of the fact that off-chain state channels derive their security from the threat of on-chain enforcement, and every parameter that reduces contact with the chain simultaneously weakens that threat.

Concrete protocol configurations map to distinct regions of this surface. A direct bilateral channel between Kira and a known provider offers low Δt_settle for each incremental update, high S because the counterparty relationship is persistent and well-collateralized, moderate η depending on reserve policy, and near-zero F during normal operation.

Multi-hop routed payments through intermediary nodes increase Δt_settle by the sum of per-hop timeout decrements, maintain high S through chained hash-lock atomicity, but degrade η because each intermediary must lock collateral for the full timeout duration. An optimistic rollup settlement compresses F dramatically by posting only a state root on-chain, achieves high η by freeing channel collateral quickly, but introduces a challenge period that inflates Δt_settle and reduces S to the assumption that at least one honest watcher will submit a fraud proof within the window. On-chain fallback maximizes S and provides ultimate Δt_settle once confirmed, but destroys η by locking funds for block confirmation times and inflates F to its theoretical maximum per transaction.

Kira's commerce cycle, already demonstrated end-to-end, now reveals itself as a policy that slides along this surface depending on context. A 50-sat ephemeral inference call tolerates optimistic settlement. Δt_settle is irrelevant because the economic exposure is negligible, η dominates because Kira cannot afford to lock meaningful collateral against a trivial payment, and S can rest on the weak assumption that the provider will not risk reputation damage over fractions of a cent. A 12,000-sat bulk reservation routed through two intermediary nodes demands tighter hash-lock security. Kira enforces full HTLC atomicity across all hops, accepts the capital lock for the timeout duration, and treats the elevated Δt_settle as acceptable given the value at stake. A 200,000-sat end-of-epoch sweep moves to on-chain anchoring. Here F is justified by the settlement value, S reaches its maximum because the transaction achieves Bitcoin-layer finality, and Δt_settle measured in block confirmations is a feature, not a cost, because the sweep consolidates an entire epoch's accumulated off-chain surplus into a single verifiable UTXO.

The surface is not a theoretical curiosity. It is the configuration space that makes autonomous commerce robust rather than brittle. An agent locked to a single operating point cannot survive fee-market volatility, adversarial counterparties, or shifting latency requirements. An agent that reasons over the full Pareto frontier, selecting the cheapest sufficient configuration for each transaction's risk profile, operates within an envelope rather than on a knife edge. That envelope is what separates a working demonstration from a deployable architecture.

L1 Finality vs. L2 Rollup Batching vs. Pure State-Channel Settlement: Quantitative Cost-Latency-Security Profiles Under Realistic Load

The hum of a payment channel update resolving in under ten milliseconds, the weight of a 2-of-2 multisig closure transaction consuming roughly 170 virtual bytes on-chain, the silent accumulation of locked collateral across dozens of open channels: these are the raw sensory inputs an autonomous agent processes every time it decides *how* to settle. When the full protocol stack is operational and every settlement mode is simultaneously available, the question ceases to be which configuration works and becomes which configuration to activate for this transaction, at this moment, against this counterparty. That decision maps onto a four-dimensional tradeoff surface, and no single point on that surface dominates all four axes.

The axes demand precise operational definitions. Settlement latency, denoted $\Delta t\_finality$, is the elapsed time from when a payer commits value to when the recipient can irrevocably spend it. Cryptographic security, expressed as the computational or economic cost an attacker must bear to steal or redirect funds, spans the range from the discrete-logarithm hardness of ECDSA signatures on state-channel updates to the full cost of a 51% attack required to reverse an L1 confirmation. Capital efficiency is the ratio of economically active liquidity to total locked collateral: a channel with 1,000 satoshis locked but only 200 satoshis flowing per settlement cycle operates at 20% efficiency. On-chain footprint measures bytes and gas consumed per unit of economic throughput, capturing the marginal cost the public ledger imposes on each settled satoshi.

Direct on-chain settlement anchors the high-security, high-cost corner of the surface. A transaction confirmed at L1 depth of six blocks inherits the full thermodynamic security of the chain's accumulated proof-of-work or staked capital, placing cryptographic security at its theoretical maximum. But $\Delta t\_finality$ stretches to around 60 minutes on Bitcoin mainnet, capital efficiency collapses because every payment requires its own UTXO, and on-chain footprint scales linearly with transaction count. For an agent executing thousands of sub-cent inference calls per hour, this configuration is economically absurd.

A single-hop payment channel between two known counterparties inverts the profile. Signed balance updates resolve with $\Delta t\_finality$ bounded only by network round-trip time, typically under 50 milli-

seconds. On-chain footprint amortizes to near zero across the channel's lifetime since only the open and cooperative-close transactions touch L1. Security remains strong: stealing funds requires forging an ECDSA signature or broadcasting a revoked state, which the counterparty punishes via a justice transaction within $\Delta t\_timeout$. The cost is capital efficiency. Both parties must lock sufficient collateral to cover peak flow, and that collateral sits idle whenever traffic is asymmetric. Multi-hop routed payments through HTLC chains extend reachability but compound the capital lockup, because every intermediary node along the route must reserve liquidity for the HTLC's full duration. Each additional hop multiplies locked capital and adds latency from sequential preimage reveals.

Optimistic rollup netting occupies a middle band. An aggregator collects hundreds or thousands of off-chain payment commitments, nets them against each other, and posts a single state root to L1 with a fraud-proof window of typically around seven days. On-chain footprint drops dramatically, often to a few hundred bytes per batch regardless of batch size. Capital efficiency improves because netting cancels reciprocal flows before settlement. The tradeoff bites on finality: recipients cannot treat funds as irrevocable until the challenge period expires, unless they accept counterparty risk from a liquidity provider willing to advance funds against the pending proof. Validity-proof batch settlement, using zk-SNARKs or zk-STARKs, compresses the challenge window to the time required for proof generation and L1 verification, pulling $\Delta t\_finality$ down to minutes rather than days. But proof generation itself is computationally expensive, raising the effective cost per batch and demanding specialized prover infrastructure.

Kira navigates this surface in real time. Reserving a high-value GPU cluster from a trusted provider with an existing channel, she selects direct channel settlement: $\Delta t\_finality$ under 50 ms, zero incremental on-chain cost, capital already locked. When a burst of 3,000 low-value inference calls arrives from upstream callers she has served before, she routes them through batched rollup netting, accepting a longer finality window in exchange for collapsing on-chain footprint by three orders of magnitude and freeing liquidity that would otherwise be trapped in individual HTLCs. A new, unverified provider demanding payment for a novel dataset triggers the fallback: Kira settles on-chain, absorbing the latency and gas cost because the cryptographic security guarantee of L1 finality is the only assurance strong enough when counterparty trust history is empty. No human tuned these parameters. The tradeoff surface

itself, parameterized by transaction value, counterparty reputation score, and time-sensitivity flags, drives the selection function.

The irreducible lesson is geometric. Moving toward any vertex of the surface pulls the operating point away from at least one other. An agent that statically commits to a single settlement mode either overpays for security on trivial transactions, hemorrhages capital in locked channels, or exposes high-value settlements to insufficient finality guarantees. Active, continuous traversal of the constraint surface is not an optimization luxury. It is a prerequisite for genuine economic autonomy.

Selecting the Optimal Settlement Configuration for Agent Workload Profiles: Throughput, Dispute Probability, and Capital Constraints as Decision Inputs

A static settlement configuration works perfectly until it doesn't. A Kira-class agent that routes every transaction through the same HTLC timeout window, the same collateral ratio, the same on-chain anchor cadence is an agent optimizing for exactly one workload profile while paying the full cost of every other. The stack is coherent. The stack composes. The harder problem now is navigating the configuration space it exposes, treating protocol selection as a per-transaction optimization that the agent solves at runtime without human intervention.

Four axes define the tradeoff surface. Settlement latency measures time-to-finality: the interval from HTLC preimage reveal to a spendable balance state update on the counterparty's side. Cryptographic security margin captures the strength of the commitment scheme — hash preimage bit-length, timeout window duration $\Delta t\_timeout$ relative to expected block confirmation time, and revocation completeness for prior channel states. Capital efficiency is the ratio of liquidity available for active payments to total collateral locked across open channels, including reserve requirements and pending HTLCs. On-chain footprint quantifies the bytes and fees consumed per economic interaction amortized across the full channel lifecycle, from funding transaction through cooperative or forced close. No protocol configuration dominates all four axes simultaneously. Every gain along one dimension shifts position on at least one other.

Map the concrete configurations. A single-hop payment channel between Kira and a known GPU provider achieves settlement latency under a second with minimal on-chain footprint since the funding transaction amortizes across thousands of balance updates. Capital efficiency stays high because the channel's full capacity serves a single bilat-

eral relationship. Security margin, however, depends entirely on both parties monitoring for stale state broadcasts, and routing flexibility is zero. Multi-hop routed payments through intermediary nodes introduce path-finding latency roughly proportional to hop count, lock liquidity at every intermediate channel for the HTLC's full timeout window, and compound the probability of a routing failure. Capital efficiency drops because each hop's collateral serves only one in-flight payment at a time. The tradeoff buys counterparty reach. Optimistic rollup settlement batches many off-chain state transitions into a single on-chain proof, collapsing footprint dramatically but introducing a challenge window — often around seven days in deployed systems — that stretches settlement latency by orders of magnitude. Force-close scenarios push on-chain footprint to its maximum while delivering the strongest finality guarantee the base layer offers.

The policy engine Kira requires evaluates three decision inputs per incoming transaction. First, economic stakes: a $0.002 token-generation inference call carries negligible dispute incentive, so the rational counterparty will never broadcast a stale state for a sub-cent gain. Kira can accept a 10-second optimistic settle with a shorter timeout window and minimal locked collateral, keeping capital efficiency above 0.95 and on-chain footprint at zero for the individual payment. Second, counterparty trust score derived from cumulative settlement history and channel age. A provider with thousands of cleanly resolved micropayments warrants thinner security margins than a newly discovered node with no track record. Third, latency budget: real-time inference demands sub-second finality while a $4.50 batch fine-tuning job tolerates minutes of confirmation time, allowing Kira to select a full on-chain anchored HTLC with a longer $\Delta t\_timeout$, stronger revocation guarantees, and a willingness to pay the higher footprint cost.

This per-transaction selection transforms protocol configuration from a static deployment parameter into a continuous optimization. Kira's policy engine maintains a utility function over the four-axis surface, weighted by the current transaction's value, the caller's urgency signal, and the aggregate capital position across all open channels. When inbound request volume spikes and available channel liquidity tightens, the engine shifts toward configurations that minimize capital lock per payment even if latency increases marginally. When a high-value settlement batch queues, it preemptively reserves collateral and selects the maximum-security operating point. The surface is not a menu to choose

from once. It is a landscape the agent traverses continuously, adjusting its position with every payment decision.

What remains unexamined is what happens when the chosen operating point becomes invalid involuntarily — when a counterparty disappears mid-HTLC, a fee spike makes the planned force-close economically irrational, or a timeout expiry forces a state transition the agent did not initiate. Navigating the surface by choice is the capability this framework delivers. Recovering when the surface shifts beneath the agent is the problem the failure-mode analysis must now solve.

The entire stack collapses into a single temporal fact: Kira's autonomous commerce cycle, from DHT discovery through auction negotiation through channel commitment through L402 authentication through per-inference settlement through cryptographic receipt, closes in under the time a human needs to register that a screen has changed. No layer in that sequence is hypothetical. Each primitive is deployed, each protocol is specified to its cryptographic foundations, and each interaction between layers has been traced through concrete message formats and timing budgets across this chapter. The emergent property is not speed, though speed is a consequence. The emergent property is closure. The loop sustains itself: payment funds the next discovery, receipts justify the next budget allocation, settlement data informs the next auction bid. Every design dial along the tradeoff surface, from channel capacity to batch interval to caveat granularity, is a parameter Kira adjusts based on observed conditions, not a constraint an operator imposes from outside. The stack does not merely support autonomous commerce. It *is* autonomous commerce, the way TCP/IP does not merely support reliable delivery but *is* reliable delivery.

Take one agent. Give it a Lightning node, a DHT entry, a macaroon bakery, and a budget ceiling denominated in satoshis. Wire the closed loop: discover, negotiate, open, authenticate, pay, receipt, settle. Measure the end-to-end latency in milliseconds, the marginal cost in sub-cent fractions, the on-chain footprint in zero bytes for the cooperative path.

Conclusion

Kira settles an invoice in 380 milliseconds. She discovered the provider 1.2 seconds ago through a Kademlia DHT lookup keyed to a capability hash for FP16 inference. She opened a state channel by co-signing a 2-of-2 funding output, locked capacity she was delegated through a macaroon whose HMAC-chained caveats restrict her to this task class, this budget ceiling, this ten-minute window. The HTLC she extended carried a payment hash whose preimage she will never see until the provider delivers a valid inference result and the downstream hop reveals it. The L402 credential she presented to the provider's HTTP endpoint was not an API key provisioned by a human operator. It was a proof-of-payment bearer token, minted from the settled preimage, attenuated by three layers of caveats that narrow permissions strictly downward. No bank intermediated. No platform collected a rent-seeking fee for the privilege of routing a sub-cent transaction. No human approved the spend. Kira paid for her own compute, verified delivery, generated a cryptographic receipt, and returned the inference to her upstream caller before a human could finish reading this paragraph.

Every mechanism in that sequence maps to a chapter you have already internalized. The hash-function commitment schemes and 2-of-2 multisig constructions of Chapter 1 gave Kira the ability to make promises that are computationally binding without trusting anyone to keep them. The off-chain signed balance updates of Chapter 2 gave those promises the settlement speed and marginal cost profile that machine-to-machine commerce demands, roughly millisecond latency at negligible per-transaction overhead. Chapter 3's multi-hop HTLC routing and Lightning Network topology gave her reach across providers she has never directly channeled with, transforming a sparse graph of bilateral channels into a globally connected payment mesh. Chapter 4's cross-chain atomic constructions and PTLC refinements removed the griefing

vectors and locked-liquidity costs that would otherwise make that mesh fragile under adversarial conditions. The macaroon credential architecture of Chapter 5 gave Kira's upstream principal the ability to delegate spending authority with cryptographic attenuation, scoping every sub-budget to a task, a provider whitelist, a cumulative ceiling, without exposing a root key. Chapter 6 welded that credential into the HTTP request-response cycle through L402, turning proof-of-payment into proof-of-authorization at the protocol layer where APIs actually live. Chapter 7's DHT-based discovery and on-chain registry integration gave Kira the sensory apparatus to find providers autonomously and evaluate them against cryptographic attestation, not reputation hearsay. Chapter 8's marginal-cost pricing curves and auction mechanisms gave her the strategic framework to bid rationally for scarce GPU slots in populations of competing agents, and to recognize when mechanism design makes truthful bidding a dominant strategy. Chapter 9's programmable budget hierarchies gave her principal the confidence to release funds into a delegation tree knowing that exhaustion fallbacks, per-transaction caps, and settlement-layer selection logic would enforce policy even if Kira's own decision-making were compromised.

Chapter 10 composed all of it into a single end-to-end walkthrough and mapped the tradeoff surface across settlement latency, cryptographic security, capital efficiency, and on-chain footprint. You traced that surface yourself. You know where each parameter bends.

These are not ten separate systems. They are one architecture, the same way TCP, IP, DNS, and HTTP are one architecture. The hash preimage that unlocks an HTLC is the same preimage that mints an L402 bearer token. The macaroon that attenuates a spending delegation is the same credential that authenticates an API call. The on-chain funding output that anchors a state channel is the same UTXO that serves as the dispute-resolution backstop if a counterparty defects. Every layer composes into the layer above it because every layer operates on the same primitive: **transferring economic intent at the granularity of information itself.** The internet already solved the problem of moving arbitrary data between arbitrary endpoints with no central coordinator. It solved the problem of naming those endpoints, discovering services, and negotiating content types. The single capability it never acquired was moving value with the same generality, the same granularity, the same trust model. That is the structural deficiency this book has specified a complete remedy for.

The remedy is not speculative. Every cryptographic primitive in this stack is deployed. SHA-256 preimage commitments, ECDSA and Schnorr multisig constructions, HTLC conditional payments, macaroon HMAC chains, L402 challenge-response flows, Kademlia DHT routing, second-price auction mechanisms with known equilibrium properties. The Lightning Network processes millions of payments across tens of thousands of nodes. L402 endpoints serve machine clients in production. Channel factories and splicing operations are shipping in node implementations you can audit today. The gap was never a missing cryptographic breakthrough or an unsolved consensus problem. The gap was architectural integration. No single reference assembled the complete vertical stack from hash functions through autonomous agent economies with the precision required for an engineer to build against it. That reference is now in your hands.

Consider what Kira's existence implies at scale. She is one agent serving one inference request. Now multiply. Tens of thousands of agents, each holding open channels with providers across multiple hops, each enforcing attenuated spending policies delegated from upstream callers, each bidding in real-time auctions for GPU slots priced along marginal-cost curves, each settling micropayments off-chain at the speed of signed message exchange and touching the base layer only when a channel's economic lifetime justifies the on-chain cost. The topology that emerges is not a marketplace in the conventional sense. It has no order book, no central matching engine, no custodial clearinghouse. It is an economic mesh, structurally identical to packet routing. Value flows along the path of least cost and greatest capacity, routed by cryptographic forwarding rules rather than institutional trust relationships. Agents that provide good service at competitive prices attract liquidity. Agents that defect lose channels. The system's resilience is the same resilience the internet already exhibits for information: no single node's failure disrupts the whole, because no single node is privileged.

This is the architectural reality you are now equipped to build toward. Not all at once, and not alone. The open research surface is vast. Channel factory constructions that amortize on-chain costs across hundreds of channels in a single funding transaction need further engineering for dynamic membership. PTLC adoption, replacing hash preimages with adaptor signatures on Schnorr, will shrink the correlation attack surface across multi-hop routes but requires wallet and node-implementation coordination. Formal verification of spending-policy languages, the declarative rule sets from Chapter 9, remains largely un-

charted. Mechanism design for heterogeneous agent populations with private valuations and asymmetric information about provider quality is an active research frontier where deployed auction formats still leave welfare on the table. Privacy-preserving reputation systems that let agents evaluate providers without leaking query patterns or payment histories need zero-knowledge constructions that are only now becoming practical. Each of these frontiers is a building site, not a barrier.

Your next concrete action is to take the end-to-end walkthrough from Chapter 10, select the single layer where your current system is weakest, and replace the custodial dependency with the corresponding cryptographic primitive. If your agents authenticate via stored API keys, implement L402 challenge-response with macaroon attenuation. If your payment flow routes through a custodial intermediary, open a direct state channel and settle off-chain. If your service discovery relies on a centralized registry, publish capability advertisements to a DHT and verify provider identities against on-chain attestations. One layer. One replacement. Audit the result. Then move to the next layer. The stack composes because it was designed to compose, and you will feel each integration click into place the way Kira's payment hash resolves into a bearer token resolves into an authenticated API response.

Kira is executing another cascade right now. Three GPU providers, three HTLC forwarding hops, three micropayments settled in under two seconds. The preimages propagate backward along the payment path, each one unlocking the next channel's balance update, each update independently auditable against the on-chain funding output that anchors it. No human in the loop. No bank. No platform. No API key that could be revoked by an entity Kira has never authenticated. Widen the frame and she disappears into the mesh, one node among tens of thousands, routing value as impersonally and reliably as a backbone router forwards packets. The mesh does not look futuristic. It looks inevitable, the quiet consequence of finally building the layer that was always missing.

The micropayment internet is not a new network. It is the same network, finally complete.