Engineering Deep Dive

Architecture and Design of the Salus Execution Platform

John Whitton

External Architecture Reference – Version 1.0

johnwhitton.com portfolio.johnwhitton.com resume.johnwhitton.com github.com/johnwhitton

---

Abstract. The Public Engineering Case Study introduced Salus as a Rust-based research and execution platform for latency-sensitive trading systems. This paper assumes that context and answers the next question: how is the platform engineered?

The central theme is runtime ownership. Salus separates market-data ingestion, state application, graph topology, route cataloging, evaluation, execution, queues, persistence, replay, observability, and analytics into boundaries that can be reasoned about independently. This document describes those boundaries, why they exist, what each subsystem deliberately does not own, and how those choices support a platform that can evolve beyond one execution strategy without losing operational clarity.

Disclosure note. This is an external architecture document. It describes system design, ownership, tradeoffs, validation, and performance evidence. It does not disclose strategy mechanics, ranking methods, fee formulas, live infrastructure configuration, wallet policy, concrete competitor traces, or reproduction steps for trading behavior.

Contents

1 Relationship To The Case Study 3

2 Canonical Architecture Goal 3

3 Design Goals 3

4 Why Rust? 4

5 Runtime Ownership 4

6 Runtime Control Flow 6

7 State Consumption And Liquidity Mapping 7

8 Graph Construction And Route Catalog 8

9 Route Lifecycle And Evaluation Pipeline 9

10 Simulation And Pricing Context 10

11 Queue Architecture 11

12 Execution Pipeline 13

13 Replay-Driven Development 14

14 Persistence And Read Models 16

15 Validation Architecture 17

16 Observability And Analytics 18

17 Performance Engineering 19

18 Failure Model 20

19 Lessons From Production Searchers 21

20 Architectural Principles 21

21 Design Tradeoffs 21

22 Terminology 22

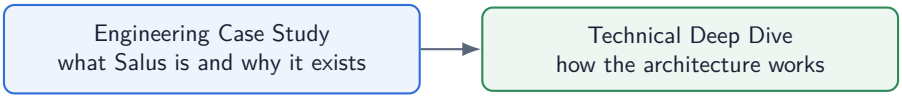
23 Conclusion 22

24 About The Author 23

1 Relationship To The Case Study

The Engineering Case Study answers why Salus exists: modern trading systems operate in a world where market state changes while the system is deciding what to do. This paper does not repeat that narrative. It uses the Case Study as the front door and focuses on implementation architecture.

The relationship is:



That distinction matters editorially. The Case Study introduces concepts such as the moving-world problem, replay-driven engineering, and the evolution from a single strategy workload toward a shared execution platform. The Deep Dive expands those ideas technically: which subsystem owns which decision, why the boundary was drawn there, which alternatives were rejected, and what future work that boundary makes possible.

2 Canonical Architecture Goal

This document is intended to become the canonical external architectural description of Salus. Internal documents currently explain the same subsystems from several angles: design records, implementation notes, walkthroughs, validation plans, queue plans, and performance records. The goal here is not to append another explanation. The goal is to consolidate the strongest public-safe version of each architectural idea into one coherent reference.

That has two consequences.

First, the document favors synthesis over completeness. When several internal records describe state ingestion, graph construction, route evaluation, or queue behavior differently, this paper creates one clearer explanation rather than preserving every version. Internal documents can remain useful as implementation records, but the external architectural story should be singular.

Second, the document is organized around ownership rather than components. A component list answers what exists. An architecture reference must answer what each subsystem owns, what it refuses to own, why that boundary matters, and how the boundary keeps the runtime evolvable.

3 Design Goals

Salus was engineered around a small set of design goals that show up throughout the runtime:

- **Correctness before live execution.** A route being mathematically interesting is not enough. The platform must prove that it can be represented, simulated, protected, submitted, and later explained.
- **Replayability.** Failures should become durable evidence, not one-off operational anecdotes.
- **Observability.** Evaluation, queues, execution gates, stale work, and output artifacts should expose enough evidence to diagnose behavior.
- **High-throughput route evaluation.** The runtime must process a large route universe without making evaluation one opaque block.
- **Multi-chain operation.** The same architecture must handle chains with different graph sizes, block cadence, and route surfaces.
- **Deterministic validation where possible.** Replay, dry-run, fixture, and bounded live modes each answer different validation questions.
- **Safe execution gates.** Execution is a staged handoff, not a side effect of route evaluation.
- **Modular strategy support.** Strategy-specific policy should use shared runtime infrastructure without becoming infrastructure itself.

The tension across those goals is latency versus explanation. A fast system can still be untrustworthy if it hides why it made a decision. Salus therefore treats explainability as part of the runtime contract. The architecture pays for explicit boundaries because those boundaries make performance work safer.

4 Why Rust?

Rust is not just the implementation language for Salus. Its programming model fits the architecture the platform needs. The central architectural idea in the system is ownership: market data owns stream normalization, the graph owns topology, evaluation owns route classification, queues own transport, execution owns submission safety, and persistence owns durable read models. Rust makes that same idea concrete in the code. Values are moved, borrowed, shared, or cloned deliberately, so crossing a runtime boundary requires an explicit decision about who may mutate state and who may only observe it.

That matters in a long-running execution system. A live block handler cannot borrow mutable graph state into background route evaluation and hope the next block does not change it. Salus instead moves owned, block-scoped request snapshots across async boundaries while sharing expensive immutable simulator state through narrow shared handles. The result is an architecture where worker tasks can evaluate routes without secretly owning the live runtime.

Rust also encourages explicit state modelling. Missing live protocol state, stale work, queue drops, rejected evaluations, dry-run failures, and execution outcomes are represented as data rather than hidden control flow. Enums, `Option`, and `Result` make those states visible at subsystem boundaries. Exhaustive matching then turns new states into compiler-visible engineering decisions instead of unreviewed fallthrough behavior.

The async model is similarly aligned with Salus. Tokio provides the long-lived runtime surface for stream consumption, bounded channels, timers, background tasks, receipt follow-up, and service coordination. CPU-heavy work remains behind explicit boundaries: route evaluation uses owned request data, route refresh can be pushed away from the stream path, and blocking work is separated from async polling. This keeps concurrency useful without turning it into ambient shared state.

The type system is also an architectural tool. Traits define adapter and simulation boundaries without forcing strategy code to know every protocol implementation. Deterministic collections make replay artifacts, logs, and validation output stable. Strongly typed queue payloads prevent one stage's work from being mistaken for another's. These choices are not about language fashion. They support predictable operational behavior in a system where latency, correctness, observability, and state ownership all matter at the same time.

5 Runtime Ownership

Runtime ownership is the organizing principle of Salus. The platform is split into layers that each own one kind of decision.

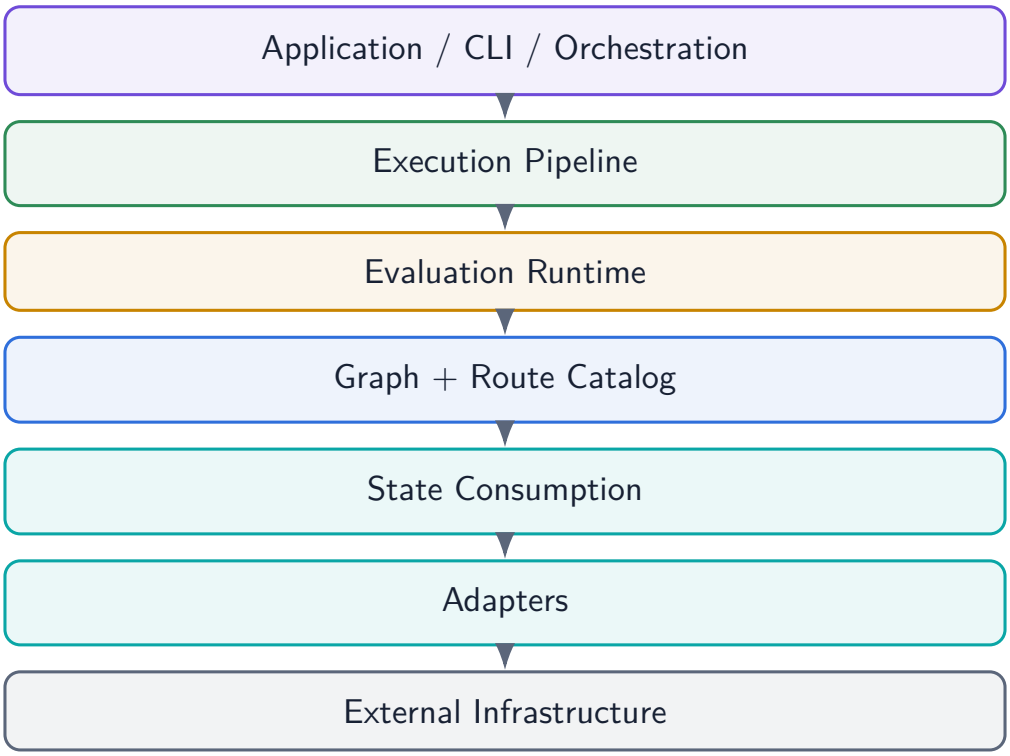


Figure 1: Layered architecture map. Each layer narrows ownership before handing evidence or work to the next boundary.

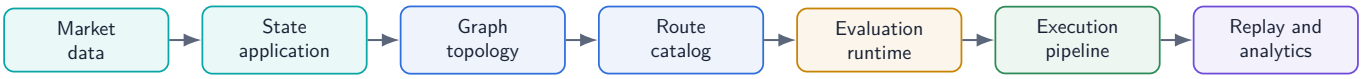


Figure 2: Runtime ownership spine. Each stage owns a different class of decision and exposes evidence to the next stage.

Subsystem	Owns	Deliberately does not own
Adapters	External systems, storage access, stream transport, RPC transport, decoding boundaries	Strategy policy, graph policy, profitability, execution selection
Ingestion	Salus-shaped stream events, tokens, components, block metadata, provenance	Route selection, execution gates, durable storage policy
Graph	Topology, route enumeration, route membership, route metadata	Profitability, current live protocol state, transaction construction
Route catalog	Runtime lookup surfaces for known routes and affected components	Pricing, selection, gas, execution outcome
Strategy/evaluation	Route evaluation, profitability classification, selection policy, explainability	Queue mechanics, persistence, submission transport
Execution	Dry-run candidates, live signals, freshness checks, preflight classification, outcome typing	Route discovery, selection ordering, graph mutation, queue policy
Runtime queues	Bounded transport, backpressure, lifecycle, metrics	Business decisions carried by queued messages
Persistence	Durable read models, warm-start topology, inspectable route summaries	Live protocol state, queue state, worker-local caches, execution decisions
Replay/analytics	Reproducible evidence and offline analysis surfaces	Live submission, hidden live state, production policy mutation

Table 1: Ownership boundaries. The architecture is designed so each subsystem can be tested and explained without absorbing adjacent policy.

The most important consequence is that data does not automatically become authority. Storage may know the route

universe, but it does not know whether a route is executable at the current block. A route catalog may know which routes touch a changed component, but it does not know whether those routes are profitable. A queue may hold execution work, but it does not decide execution policy. These distinctions are what keep the runtime explainable under load.

5.1 Architecture Review Questions

Each subsystem in this document is evaluated through the same review pattern:

- **What problem does this boundary solve?**
- **What does this boundary own?**
- **What does this boundary refuse to own?**
- **What alternative design would be simpler?**
- **What tradeoff does the current design accept?**
- **How does this boundary support future execution models?**

This pattern prevents the document from becoming a catalog of components. The important thing is not that Salus has a graph, a queue, a route catalog, or a replay command. The important thing is that each subsystem owns a different engineering question and can be improved without silently changing the others.

6 Runtime Control Flow

The live runtime is easiest to understand as a sequence of ownership handoffs. The app orchestrates these handoffs, but it should not absorb the policy behind them.

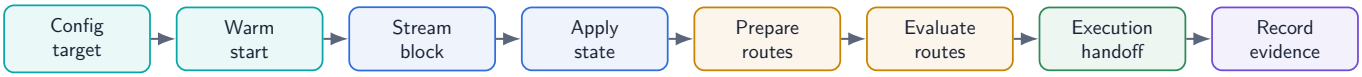


Figure 3: Live runtime control flow. The application coordinates handoffs; subsystem policy remains owned by the subsystem.

This flow has two important asymmetries.

First, startup and block processing have different meanings. Warm-started storage can load graph and route topology, but it cannot make the stream state fresh. The runtime therefore treats startup as a catalog-hydration step and block processing as the point where executable state enters the system.

Second, evaluation and execution have different clocks. Evaluation is block scoped and can use worker fan-out. Execution is order sensitive and intentionally serial at the submission boundary. Treating those clocks as the same would make route evaluation wait on execution details that do not belong to it.

6.1 Operational Modes

The same architecture supports several operating modes because the ownership boundaries are stable:

Mode	Question answered	Primary evidence
Storage inspection	What topology and routes are known?	Graph counts, route summaries, route-scope hydration
Replay evaluation	Does a labelled state source reproduce route behavior?	Replay artifacts, route statuses, rejection reasons
Follow dry-run	Can repeated labelled states exercise evaluation and candidate construction?	Follow artifacts and dry-run candidate outputs
Bounded live dry-run	Does current stream state evaluate and construct candidates safely?	Live route metrics, dry-run outputs, queue summaries
Continuous live mode	Can the runtime keep up with the stream and preserve execution safety?	Runtime metrics, route attempts, execution outcomes
Offline analytics	What does a labelled route universe show for one state source?	Analysis summaries and route-level artifacts

Table 2: Operational modes. Each mode uses the same architecture to answer a different engineering question.

This is why the Technical Deep Dive avoids presenting one command as the system. Commands are entry points into architectural surfaces. The surfaces are the durable part.

7 State Consumption And Liquidity Mapping

7.1 Problem

Live trading infrastructure has to consume partial, rapidly changing market state. A stream update may carry topology changes, component state changes, block metadata, and provenance, but it is not a complete historical archive of everything the runtime has ever known. Salus therefore treats live state as a stream-owned frame: decoded state is applied as it arrives, and evaluation only uses routes whose required state is present in that frame.

7.2 Architecture

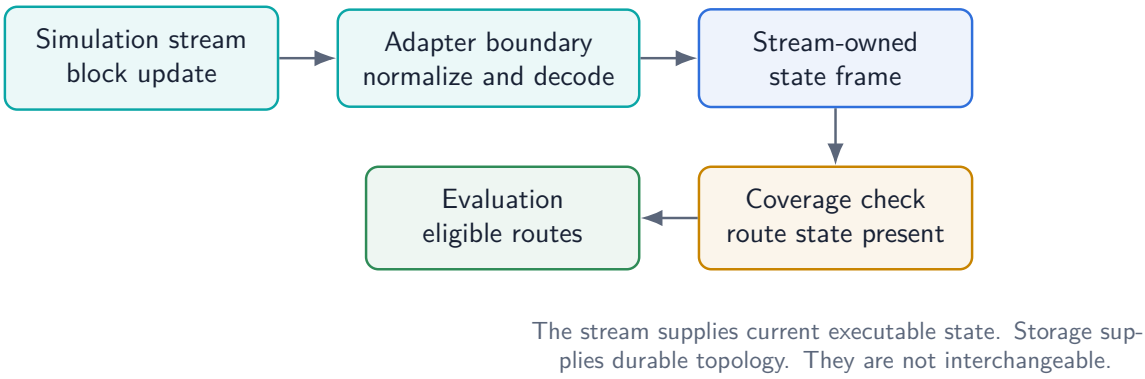


Figure 4: State consumption boundary. Live state is applied through an adapter-owned stream boundary before routes become eligible for evaluation.

The ingestion layer owns normalized component and token records, stream block metadata, and decoded protocol state surfaces. The graph layer owns topology: tokens, liquidity components, component-token membership, and route connectivity. The route catalog bridges those worlds by making topology useful at runtime. It answers questions such as: which known routes include this component, which routes start from this token, and which route ids are available for scoped evaluation?

7.3 Ownership Contract

State consumption owns the conversion from upstream stream shape into Salus-shaped runtime inputs. That includes block identity, component updates, token metadata, protocol-state deltas, and provenance. It does not own route selection. It does not own graph mutation policy beyond providing the facts the graph needs. It does not decide whether a candidate should be evaluated or executed.

The state frame has to be explicit because it is the only place where the runtime can answer "do we have the current state required for this route?" A route can be topologically valid and still unevaluable for the current block if one of its components lacks decoded state. Treating that as an explicit coverage question is safer than allowing the evaluator to guess or silently reuse old state.

7.4 Why Not Treat The Stream As Storage?

The stream is optimized for current state and ordered updates. Storage is optimized for durable inspection and warm start. Combining the two creates ambiguous authority: if a value exists in storage but not in the current stream frame, is it current enough to execute against? Salus avoids that ambiguity by requiring live evaluation to prove state coverage against the stream-applied frame.

That design also improves replay. A replay artifact can label its state source and route universe explicitly. The platform can then compare source-state validity with later-state invalidity without pretending all state came from one implicit global cache.

7.5 Tradeoffs

The main alternative would be to treat persisted topology and live state as one database-backed model. That makes startup and inspection simpler, but it hides a dangerous difference: durable topology can be old but still useful; executable state has to be current. Salus keeps those concepts separate. Persisted read models warm-start graph and route catalog ownership, while live evaluation waits for stream-applied state.

Another alternative would be to rebuild broad route surfaces on every block. That is simpler conceptually and expensive operationally. The current design uses long-lived topology plus component-level impact analysis. Topology changes can refresh affected route families, while ordinary state changes can trigger evaluation for known impacted routes without rebuilding the universe.

8 Graph Construction And Route Catalog

8.1 Problem

The graph has to serve two different needs. It must represent liquidity topology accurately enough to enumerate route candidates, and it must expose small enough lookup surfaces that a live block can be processed without materializing the entire route universe every time.

8.2 Architecture

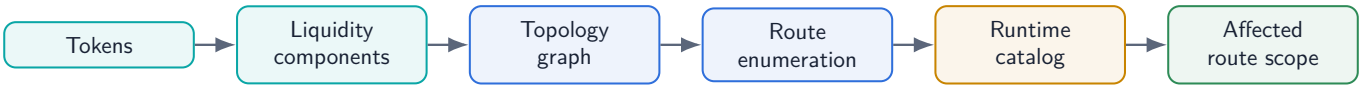


Figure 5: Graph and route-catalog lifecycle. The public architectural point is topology ownership, not private route-selection behavior.

The graph owns topology and route enumeration. The catalog owns runtime lookup over known routes. It can be hydrated from persisted route summaries at startup and then updated when route-refresh work completes. The catalog does not store profitability, quote state, gas context, or execution outcomes.

This boundary is subtle but important. A route catalog should be able to say "these routes may be affected by a

changed component." It should not answer "these are the routes to execute." The first is topology. The second belongs to evaluation and strategy policy.

8.3 Tradeoffs

A single graph object that owns topology, route admission, evaluation, and selection would be easier to call from the application layer. It would also be harder to validate. Salus keeps graph ownership narrower so route formation can be tested without execution semantics, and evaluation can be tested without mutating topology.

The cost is coordination. The app runtime has to apply stream updates in order, update graph state, maintain route catalog snapshots, and hand route scopes to evaluation. That coordination is explicit because hidden coupling would make freshness bugs harder to isolate.

8.4 Route Identity And Catalog Stability

A route catalog is useful only if route identity is stable enough to support inspection, replay, forced diagnostics, and analytics. Salus treats route ids and route summaries as durable topology read models. The runtime can therefore hydrate a catalog, select route ids for focused runs, compare replay artifacts, and preserve route-level evidence across different validation modes.

The catalog is still not the runtime truth for execution. It is the truth for known topology. Execution truth requires current state, evaluation, safety gates, and outcome evidence. This split lets route identity support debugging without turning the catalog into a decision engine.

8.5 Incremental Updates

Live graph mutation has to distinguish topology changes from ordinary state updates. A reserve update may make known routes worth evaluating. A new component or changed component membership may require route discovery. Those paths have different costs and different freshness implications.

The current design keeps expensive route refresh work behind an explicit background stage and keeps known-route evaluation on the hot path. That means a newly discovered route may become eligible on a later block, while already-known affected routes can be evaluated immediately. This is a deliberate tradeoff: fresh evaluation of known routes is more important than blocking the stream to make every possible new route available in the same block.

9 Route Lifecycle And Evaluation Pipeline

9.1 Problem

Route evaluation has to process a large candidate surface while still preserving the difference between discovery, admission, evaluation, selection, and execution promotion. Collapsing those states into one "found route" event would make the runtime fast to demo and hard to operate.

9.2 Architecture

Figure 6: Route lifecycle. Each step narrows or annotates the route set before any execution handoff is possible.

The route-preparation stage converts impacted route ids into evaluation requests for a specific block. The evaluation runtime uses block-scoped state, quote context, protocol simulation, and worker fan-out to evaluate routes. The selection layer produces execution artifacts only after evaluation has created structured route results.

The runtime deliberately distinguishes:

- routes known by topology
- routes impacted by a changed component
- routes with required state coverage
- routes evaluated against the current block state
- routes classified as economically interesting
- routes selected for possible execution

That lifecycle supports observability. If no execution candidate appears, the operator can ask whether the problem was route scope, state coverage, pricing, simulation, profitability, freshness, or selection.

9.3 Evaluation Ownership

Evaluation owns route-level classification. It asks: given this route, this input, this current state, and this quote context, what happened? It should produce structured results that downstream stages can inspect without rerunning the route. It should not mutate graph topology, write durable storage, enqueue submission work directly, or hide missing-state rejections behind aggregate counts.

The route-prep boundary exists because constructing an evaluation request is itself meaningful work. It may require route lookup, token metadata, state coverage checks, route-catalog snapshots, and stale-work checks. Keeping that work separate from evaluation service time makes performance diagnosis more honest.

9.4 Latest-Block Freshness Versus Drain-All Diagnostics

Salus intentionally supports two different evaluation shapes. In diagnostic mode, retaining previous blocks can show broad throughput and route-universe behavior. In live freshness mode, latest-block behavior matters more than FIFO fairness. The queue and evaluation stages therefore expose stale-work handling instead of pretending every admitted block should always finish.

The design consequence is that "discarded as stale" is not inherently a failure. It can be the correct outcome when a newer block makes older work less valuable than current work. The important thing is that the runtime records the discard so performance analysis can distinguish healthy coalescing from unbounded backlog.

9.5 Tradeoffs

The major tradeoff is between breadth and freshness. Broad no-throttle evaluation is useful for diagnostics because it shows how the evaluator behaves against a large route universe. Live operation needs bounded work because stale evaluation is worse than narrower current evaluation. Salus therefore supports both diagnostic breadth and current-block freshness, while keeping those modes explicit.

10 Simulation And Pricing Context

10.1 Problem

Route evaluation is not just path traversal. Each leg must be evaluated against the protocol state available for that block, and route economics must be expressed in a consistent quote context. The platform also needs enough determinism that replay and validation can reproduce important decisions.

10.2 Architecture

The simulation boundary is designed around fixed inputs: route, input token, input amount, output token, current protocol state, quote context, and safety constraints. Strategy-specific amount search and route-family selection remain above this boundary. That split matters for platform evolution. Future strategies can use a shared fixed-input evaluator without inheriting the current strategy's amount-search or objective policy.

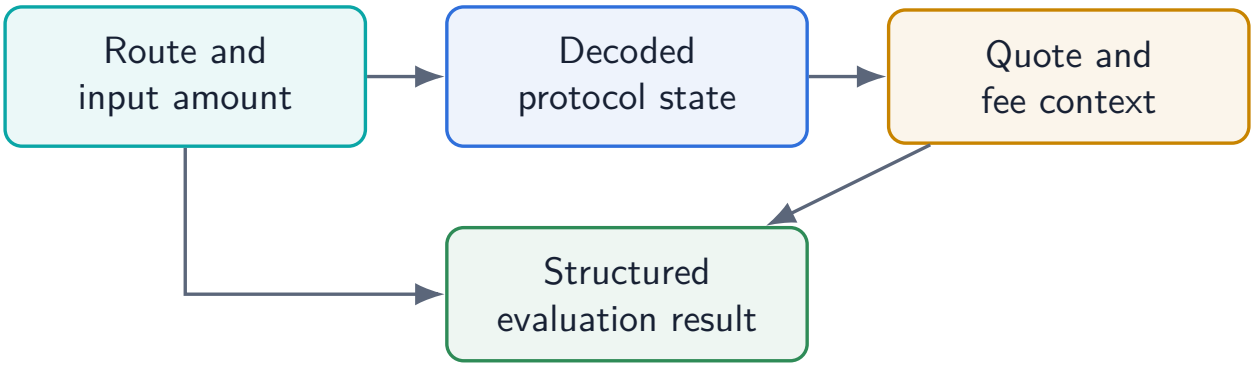


Figure 7: Evaluation context. Simulation answers a fixed-input route question; strategy policy decides which questions to ask.

Pricing context is part of the evaluation input, not a hidden global. That makes route-level results easier to replay and compare. High-level gas-aware profitability is reported as engineering evidence, but exact formulas and thresholds are not part of the public architecture.

10.3 Quote Consistency

Quote context exists because route outputs and execution costs need a shared unit for engineering comparison. The platform should be able to explain whether a route was rejected because raw path output was poor, because quote context was missing, because execution cost changed the result, or because a safety gate blocked promotion.

The architecture therefore treats quotes as labelled inputs to evaluation and analytics. That makes route-level evidence easier to audit. It also prevents quote lookup from becoming a hidden substitute for protocol simulation. The route still has to be simulated through the supported protocol path; quote context helps interpret the result.

10.4 Why Fixed-Input Evaluation Matters

The fixed-input evaluator is the platform boundary that future strategies can reuse. A strategy can decide how to search the input space or what objective to optimize, but the lower layer should answer a narrower question: for this specific route and input, what output and rejection evidence does the current state produce?

That separation keeps strategy research from rewriting execution plumbing. It also makes replay and validation more durable. A replay artifact can carry a fixed route question and compare behavior across implementation changes without depending on the full live strategy search loop.

10.5 Tradeoffs

Using protocol simulation directly is more complex than relying only on cached prices, but it preserves the execution semantics of the route. Using a generic fixed-input evaluator is less convenient for strategy-specific optimization, but it creates a reusable platform boundary. Salus keeps strategy search above the evaluator so future execution models can supply different objectives without changing the simulation contract.

11 Queue Architecture

11.1 Problem

The live runtime has work with very different latency profiles: stream ingestion, graph persistence, route refresh, route preparation, route evaluation, route persistence, and execution. If all of that work runs inline with stream polling, the system can fall behind. If queues absorb policy, the system becomes impossible to reason about.

11.2 Architecture

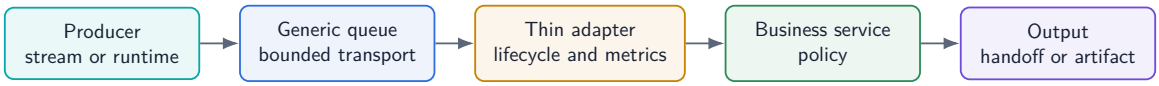


Figure 8: Queue ownership pattern. The queue transports work; the business service owns the decision carried by that work.

The queue layer owns bounded capacity, overflow behavior, lifecycle, monitored senders, queue depth, peak depth, blocked sends, dropped sends, and shutdown. Queue adapters own mechanical concerns such as dequeueing, retry shape, latest-block coalescing, stale-work purging, and delegation. Business services own graph, route-prep, evaluation, execution, or persistence decisions.

Current queue-shaped surfaces include live ingest, graph and route stages, graph persistence, route refresh, route persistence, route preparation, route evaluation, and execution-stage handoff. They are intentionally not identical. Some are durability queues. Some are freshness queues. Some are serial execution seams. Some are latest-block coordination seams.

11.3 Queue Types By Ownership

Queue shape	Why it exists	Correct default
Durability queue	Move graph or route persistence off the hot path while preserving committed work	Drain on shutdown
Refresh queue	Run expensive route discovery separately from stream state application	Latest useful work, owner-applied output
Preparation queue	Build evaluation requests without blocking stream polling	Stale-aware completion handoff
Evaluation queue	Coordinate block-scoped route evaluation and worker fan-out	Latest-block freshness by default, drain-all for diagnostics
Execution queue	Serialize live execution signals and preserve submission evidence order	Serial manager, bounded handoff

Table 3: Queue shapes. Queue policy follows the stage's ownership, not one universal FIFO rule.

11.4 Backpressure As A Design Signal

Backpressure is not merely an implementation detail. It tells the runtime which stage is claiming more work than the next stage can process. A bounded queue turns that claim into observable evidence: depth, peak depth, blocked sends, dropped sends, stale purges, and shutdown drain behavior.

The design question is what backpressure should mean for each stage. For durability, blocking may be acceptable because losing committed state is worse. For freshness, dropping or coalescing stale work can be correct because old work is less valuable than current work. For execution, serial processing is intentional because ordering is part of correctness.

11.5 Tradeoffs

FIFO is not always the right model. A diagnostic run may intentionally drain previous blocks to measure throughput. A live freshness run may prefer the newest block and purge stale queued work. Execution remains serial because submission ordering and evidence ordering matter more than parallelism.

The architectural rule is that queue policy can decide what happens to work as work: buffer, block, drop newest, coalesce to latest, or drain on shutdown. It cannot decide whether a route is good, whether a graph edge is valid, or whether an execution should be submitted.

12 Execution Pipeline

12.1 Problem

Execution must not be a side effect of evaluation. A profitable-looking route can still fail candidate construction, freshness validation, calldata construction, gas planning, final preflight, broadcast, or outcome tracking. Those are different failure classes and need different evidence.

12.2 Architecture

Figure 9: Execution gate model. Each gate produces separate evidence so failures do not collapse into one ambiguous status.

The execution boundary begins after strategy evaluation and dry-run candidate construction. The candidate is structured evidence: route id, token path, component path, input and output expectations, quote context, gas context, and payload class. It is not a transaction. The live signal is a runtime message that asks the execution stage to attempt planning and submission under the current execution context.

Execution owns:

- candidate typing and dry-run results
- live execution signals
- source-block freshness validation
- transaction planning inputs
- final preflight classification
- outcome typing for broadcast, pending, confirmed, reverted, or rejected

Execution does not own route discovery, graph mutation, profitability policy, or queue mechanics. Submission adapters own transport details behind an adapter boundary, while execution owns the platform's classification of the result.

12.3 Tradeoffs

Serial execution is a deliberate constraint. It reduces raw throughput at the submission boundary, but it protects nonce ordering, output ordering, and evidence clarity. Receipt tracking can be separated from later signal handling so the system can keep moving after broadcast while still retaining later outcome evidence.

Another tradeoff is preflight cost. Final preflight adds latency, but it separates unsafe execution from accepted execution attempts. Salus treats that cost as part of the safety model, not as an optional convenience for normal operation.

12.4 Outcome Taxonomy

Execution outcomes need to be more specific than success or failure. A rejected candidate, preflight rejection, broadcast-only attempt, pending receipt, confirmed receipt, reverted receipt, and stale source state all imply different engineering work.

Outcome	Meaning	Follow-up question
Candidate rejected	The route could not become an execution candidate	Was the evaluated artifact complete enough?
Freshness rejected	The source state was no longer acceptable	Is the runtime too slow or the guard doing its job?
Preflight rejected	The final transaction simulation failed	Is this model mismatch, state drift, or integration failure?
Broadcast-only	The transaction was sent but terminal receipt evidence is absent	Is receipt follow-up delayed or unavailable?
Confirmed	Execution reached a terminal success state	Did observed outcome match expected evidence?
Reverted	Submission landed but failed	Can replay classify model, timing, or downstream state change?

Table 4: Execution outcome taxonomy. Specific outcomes turn live behavior into actionable engineering questions.

12.5 Minimum-Output Protection

Minimum-output protection is an execution safety concept, not a route-discovery concept. Evaluation can estimate whether a route is interesting. Execution has to ensure that the submitted request cannot settle below the accepted safety constraint. Keeping that protection in the execution gate prevents evaluation from becoming an implicit transaction policy engine.

This boundary also improves replay. If a route was valid at source state but failed later due to protection constraints, the replay story can separate route modeling from settlement safety.

13 Replay-Driven Development

13.1 Problem

Without replay, teams argue from logs. A failure can look like bad strategy when it was stale state, model mismatch, ordering loss, insufficient state coverage, calldata construction, or an operational issue. Salus treats replay as a first-class architectural primitive because execution systems need a way to turn transient live behavior into durable engineering evidence.

13.2 Architecture

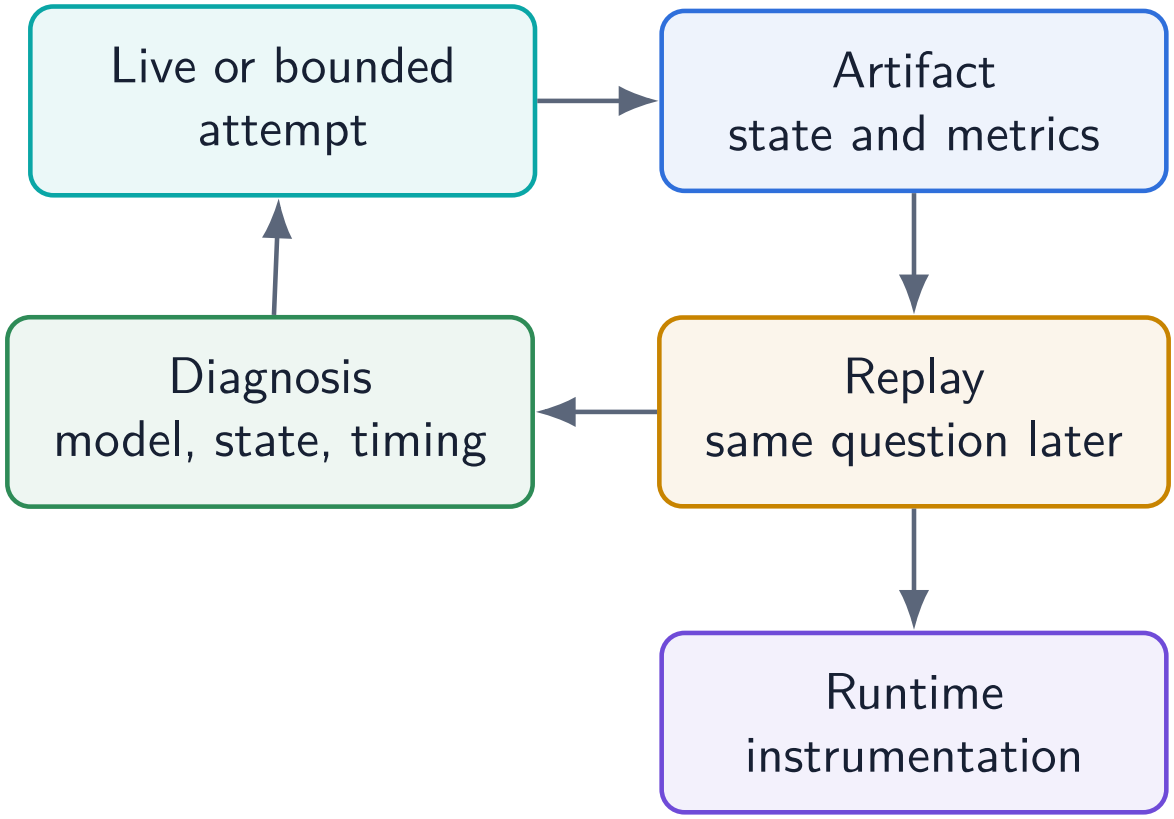


Figure 10: Replay architecture. Replay converts live behavior into evidence that can improve evaluation, execution, and instrumentation.

Replay inputs supply labelled state, route scope, block metadata, quote context, and optional decoded protocol state. Storage supplies route read models and route metadata. Strategy owns evaluation behavior. Execution owns dry-run candidate typing when replay moves into follow dry-run mode. The app orchestrates the command and writes artifacts.

Replay answers questions such as:

- Was the route valid against the source state?
- Did later state invalidate the execution?
- Was the failure caused by missing model coverage or by timing?
- Did the route require protocol state that was not present?
- Did dry-run candidate construction preserve the evaluated route evidence?

13.3 Tradeoffs

Replay is not a substitute for live freshness. A replay artifact can show what would have happened under a labelled state source; it does not prove that live state will be available at the moment of submission. That is why replay and live validation are separate surfaces. Replay makes behavior reproducible. Live validation proves current integration behavior.

13.4 Replay As Platform Infrastructure

Replay is not only a debugging feature. It is the mechanism that allows execution research to compound. Once execution attempts can be reconstructed, the platform can compare source-state validity, later-state invalidity, simulation

behavior, queue delay, and execution outcome without relying on memory or partial logs.

That matters for future strategies because each strategy will create new failure modes. A generalized execution platform needs one replay language that can describe route state, execution evidence, and outcome classification across those strategies. The current replay surface is the beginning of that language.

14 Persistence And Read Models

14.1 Problem

The runtime needs durable topology and route read models for warm start, inspection, and analytics. It also needs to avoid turning the database into a live-state authority. If storage owns too much, the system can accidentally treat old topology as executable state.

14.2 Architecture

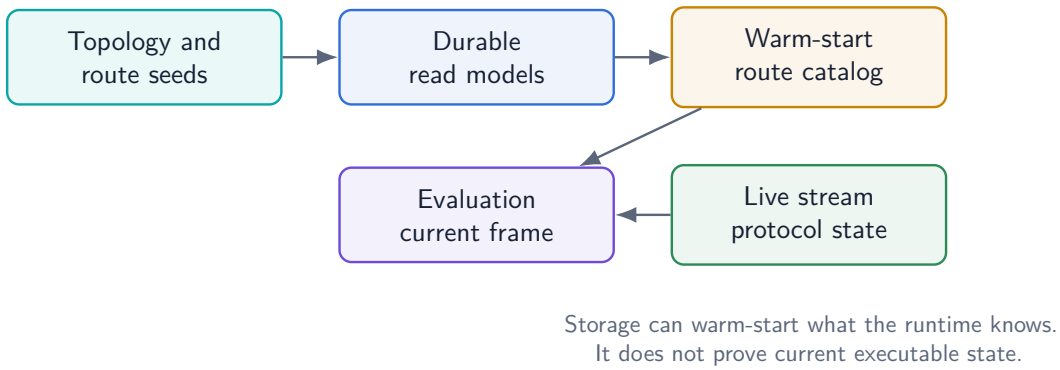


Figure 11: Persistence boundary. Durable read models support startup and inspection while live state remains stream-owned.

Persistence owns canonical tokens, protocol components, component-token membership, graph metadata, route summaries, route legs, derived route-scope read models, high-level run summaries, and stable output artifacts.

Persistence does not own decoded live protocol state, queue messages, worker-local caches, current route profitability, current submission context, or per-block scheduling state. Those are runtime or artifact surfaces.

14.3 Tradeoffs

Keeping persistence narrow makes the live runtime more explicit, but it also requires more startup orchestration. The app must hydrate topology and route catalogs, then wait for current stream state before evaluation. That extra coordination prevents a common failure mode: confusing "known route" with "currently executable route."

14.4 Read Models Versus Runtime State

Read models are intentionally shaped for inspection and warm start. They answer questions such as how many tokens are known, which components exist, which route summaries are persisted, and which routes touch a component. Runtime state answers questions such as which block is current, which protocol states are decoded, which route-prep jobs are stale, and which execution attempts are in flight.

The separation keeps storage useful without making it dangerous. If read models try to become the live runtime, they start accumulating transient concepts: queue state, worker state, live quote state, preflight state, nonce state, and receipt state. Those concepts have different lifecycles and should stay out of the durable topology store.

14.5 Analytics Artifacts

Analytics artifacts are a third category. They are durable enough to inspect after a run, but they are not canonical live state. Route-level JSONL outputs, runtime metric streams, replay batches, and analysis summaries are evidence surfaces. They should be labelled by state source and route universe so the reader knows what question they answer.

This is why offline analytics can use a current-catalog route universe for a diagnostic run without implying that the same universe existed historically at that block. Labelling prevents analytics from becoming misleading.

15 Validation Architecture

15.1 Problem

No single validation command can prove a system like Salus. Config validation, storage inspection, deterministic replay, bounded live dry-run, execution preflight, runtime metrics, and parity checks each answer different questions. The validation architecture therefore has layers.

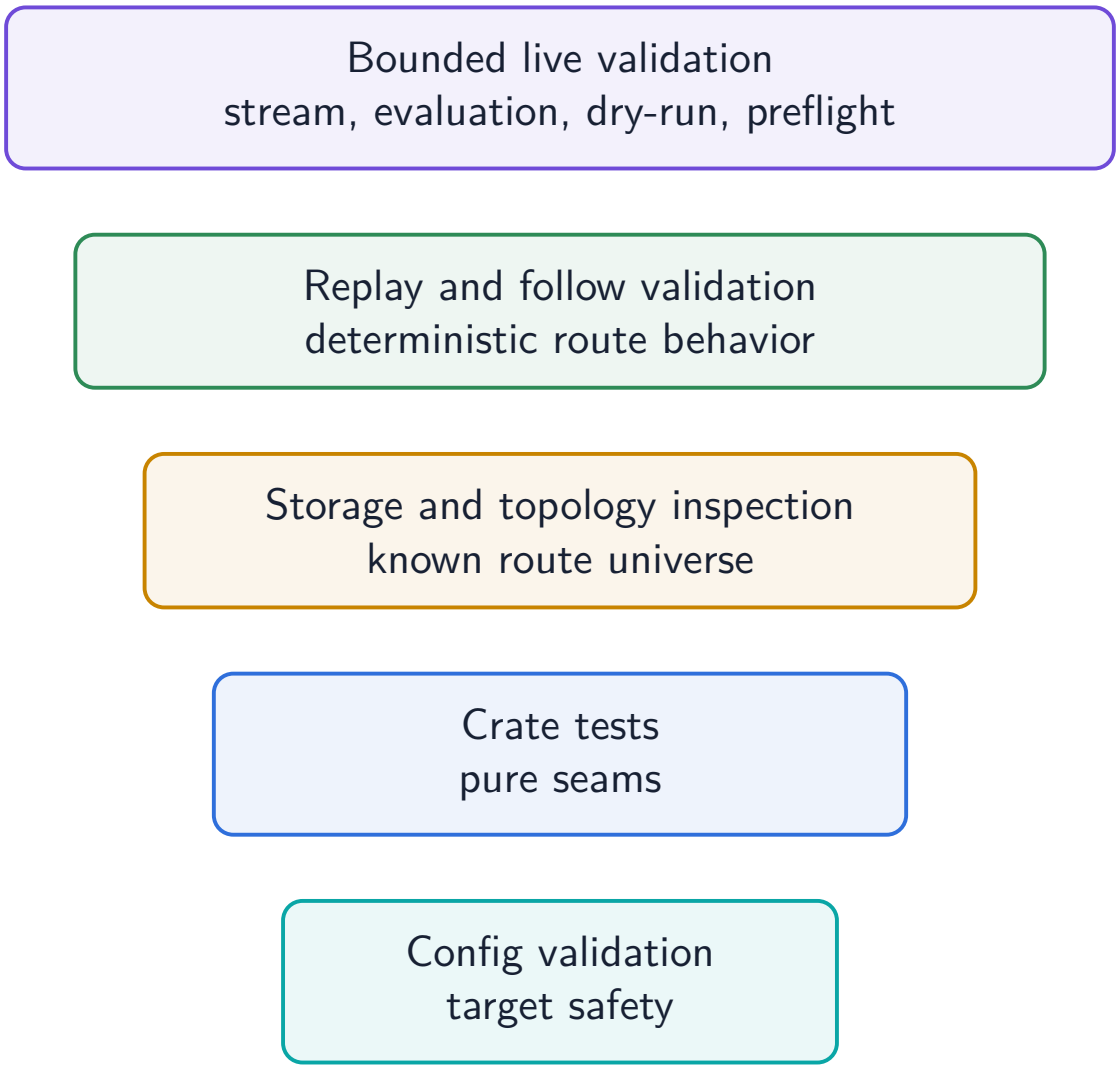


Figure 12: Validation layers. Each layer answers a different engineering question and narrows the debugging surface.

Config validation answers whether the operator selected a coherent target. Storage inspection answers what topology and routes are known. Replay answers whether a labelled state and route universe produce stable evaluation behavior. Bounded live validation answers whether the current stream, route catalog, evaluation, dry-run, and execution gates

work together. Runtime metrics answer whether the system behaved within operational expectations.

15.2 Tradeoffs

Layered validation is more work than one end-to-end test, but it makes failures actionable. If storage route counts are wrong, live execution tests are the wrong debugging tool. If replay is stable but bounded live evaluation rejects many routes as missing state, the issue is likely live coverage or state source alignment. If evaluation succeeds but execution rejects the candidate, the failure belongs to execution gates rather than graph or strategy evaluation.

15.3 Validation Question Matrix

Question	Best validation surface	Why
Does config target the intended chain?	Config validation	Fails before storage, stream, or execution startup
Does storage contain coherent topology?	Inspect commands	Reads durable route and graph models directly
Does route evaluation behave deterministically?	Replay evaluation	Uses labelled state and fixed route universe
Does route prep keep up with live blocks?	Runtime metrics	Shows queue wait, stale work, and service timing
Does execution planning reject unsafe candidates?	Dry-run and preflight validation	Exercises execution gates without requiring live settlement
Does the live stream provide usable current state?	Bounded live dry-run	Combines stream, route catalog, evaluation, and dry-run evidence

Table 5: Validation question matrix. The right validation surface depends on the architectural boundary under review.

15.4 Why Parity Is Bounded

Salus began with a migration reference, but the platform is now Salus-native. Parity remains useful for bounded shared contracts and historical confidence, not as a reason to preserve legacy architecture. That distinction matters for future work. A strict parity harness can catch regressions in shared behavior, but it should not prevent the runtime from adopting cleaner ownership boundaries when the behavior contract is preserved.

16 Observability And Analytics

16.1 Problem

Trading runtimes fail in ways that aggregates can hide. A system can have high throughput while accumulating stale work. It can show profitable observations while producing no executable candidates. It can broadcast successfully while never receiving terminal receipt evidence. Observability has to preserve those distinctions.

16.2 Architecture

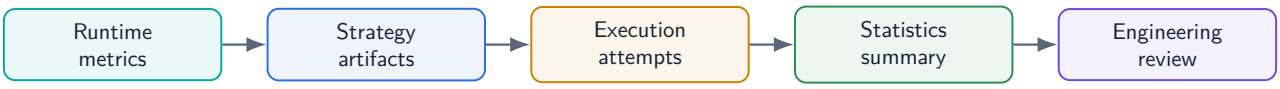


Figure 13: Evidence feedback loop. Metrics and artifacts are designed to support engineering review, not only dashboards.

The platform exposes several evidence surfaces:

- runtime metrics for queue depth, service time, worker utilization, stale work, stream cadence, and execution attempts
- route evaluation artifacts for route-level status and rejection reasons
- execution artifacts for dry-run, preflight, submission, and outcome classification
- retained statistics for cross-chain runtime behavior
- replay artifacts for later investigation

The architecture keeps these outputs owned by the subsystem that understands them. Graph analytics explain topology. Strategy artifacts explain route evaluation. Execution telemetry explains execution gates and outcomes. The app can serialize and report them, but it should not own the underlying policy.

16.3 What Observability Must Preserve

Observability should preserve causality, not just counters. A useful metrics stream lets an engineer reconstruct how work moved through the runtime:

- when a block was observed
- how many routes were admitted for preparation
- how long preparation waited and ran
- how many routes reached evaluation
- how much work was discarded as stale
- which routes became execution candidates
- which execution gate accepted or rejected them

That causality is more important than a dashboard number. It lets the team ask whether a performance issue is stream intake, route scope, request construction, evaluation compute, queue backlog, preflight, or receipt follow-up.

17 Performance Engineering

Performance work in Salus is evidence-driven. The most useful statistics are not marketing benchmarks; they show which part of the architecture is scaling and where backlog or staleness appears.

Chain	Graph tokens	Components	Routes	Sustained eval/s	Peak eval/s	Avg ms/route
Ethereum	3,415	4,552	2,401,108	33,578	44,610	0.0357
Base	1,837	2,505	537,312	19,422	32,897	2.5235
Unichain	29	61	8,414	2,203	22,500	0.4554

Table 6: Representative retained runtime statistics. These are engineering scale indicators, not trading strategy disclosures.

The cross-chain differences are more useful than any single number. Ethereum exercises a large route universe and demonstrates evaluator throughput. Base shows why stale-work handling matters: the runtime can admit more work than it should finish if newer blocks supersede evaluation windows. Unichain provides a smaller validation target that still exercises the same pipeline shape.

Performance decisions follow three principles:

- **Measure the stage, not only the command.** Queue wait, request build, evaluation service time, compute time, finalization time, and worker utilization answer different questions.

- **Bound live work.** Diagnostic breadth and live freshness are different operating modes. Unlimited route evaluation is useful for evidence; it is not automatically a live profile.
- **Keep ownership visible.** If route prep is slow, the fix should land in route prep or catalog ownership. If evaluation is slow, the fix belongs to evaluation. Queue changes should not hide subsystem costs.

17.1 Interpreting The Numbers

The retained statistics show why architecture matters. Ethereum's large graph and route universe stress evaluator throughput and startup cost. Base's stale route behavior shows why latest-block policy exists. Unichain's smaller graph proves that the same pipeline can operate at a different scale without becoming a special-case runtime.

The most important performance question is therefore not "how many routes can the system evaluate?" It is "how much useful current work can the system evaluate while preserving enough evidence to explain what happened?" That is a harder and more relevant question for production execution infrastructure.

17.2 Performance Tradeoffs

Salus accepts several performance tradeoffs deliberately:

- warm start can do more work once so stream processing can avoid rebuilding topology every block
- route refresh can complete after known-route evaluation rather than blocking the hot path
- route evaluation can fan out within a block while execution remains serial
- latest-block coalescing can discard stale work to protect freshness
- retained metrics add overhead because explanation is part of the system's value

These are architecture decisions, not just optimizations. Each one chooses what the runtime values when time, state, and work volume conflict.

18 Failure Model

The failure model is intentionally conceptual and public-safe. It is organized around engineering classes rather than specific transactions.

Failure class	What it usually means	Owning boundary
Stale state	The route was evaluated against state that no longer matched the execution moment	Freshness and execution planning
Missing state	The route catalog knew a route, but the current state frame lacked required component state	State coverage and evaluation
Simulation mismatch	Replay or live simulation did not match expected protocol behavior	Simulation and validation
Minimum-output protection	A candidate could not satisfy output safety constraints at execution time	Execution gate
Preflight rejection	Final transaction simulation rejected the planned request	Execution preflight
Reverted submission	A submitted transaction was included but failed	Outcome tracking and replay
Route overlap	Multiple candidates reflected one underlying market movement	Analytics and route-family explanation
Fee pricing failure	Execution cost assumptions were insufficient for the live opportunity window	Execution research and pricing context
Downstream state change	Another state change invalidated the opportunity before settlement	Replay and timing analysis

Table 7: Public-safe failure model. Each class points at an engineering boundary rather than a strategy reproduction path.

The architectural goal is not to eliminate every failure class. In a moving market, some failures are expected. The goal is to classify failures accurately enough that engineering effort goes to the right layer.

19 Lessons From Production Searchers

Production searchers tend to converge on a few public, architecture-level lessons:

- specialized executors can reduce transaction overhead and improve execution control
- ordering strategy matters as much as opportunity discovery
- non-public submission alone does not guarantee inclusion priority
- bundled or inventory-aware execution can change the economics
- replay and observability are necessary to understand missed attempts

For Salus, the important lesson is not to imitate any specific external implementation. The important lesson is that execution quality is a systems problem. It includes state freshness, route evaluation, fee context, preflight, submission path, receipt evidence, capital model, and post-run analysis.

This is why the platform separates opportunity discovery from execution research. The same route result can be used to ask several engineering questions: was the model correct, was the state fresh, was the submission early enough, was the execution path appropriate, and did the artifacts explain the outcome?

20 Architectural Principles

The architecture is guided by a small set of principles:

Principle	Meaning in Salus
Ownership before convenience	Boundaries are kept explicit even when a shorter implementation would merge orchestration, evaluation, queues, and execution.
Replay as architecture	Replay is treated as a platform primitive for understanding behavior, validating changes, and explaining outcomes.
Evidence before intuition	Performance tuning and execution research start from artifacts, metrics, and classified outcomes rather than anecdotes.
Shared infrastructure before strategy expansion	New execution models should reuse state, graph, evaluation, execution, replay, and analytics surfaces instead of duplicating them.
Strategy-agnostic runtime boundaries	Strategy modules may ask different questions, but they should consume common runtime contracts.
Freshness before completeness	In live execution, current useful work is more valuable than exhaustively finishing stale work.
Public-safe observability	The platform should explain engineering behavior without exposing strategy mechanics or private operating details.

Table 8: Architectural principles. The point is not to add process language, but to summarize the engineering choices that recur throughout the runtime.

21 Design Tradeoffs

21.1 Explicit Boundaries Over Shorter Code

Salus could be shorter if the app runtime owned graph mutation, route selection, queue mechanics, persistence, and execution. It would also be much harder to reason about. The current design prefers explicit boundaries because they make failures local and future strategy support possible.

21.2 Reusable Runtime Before Strategy Expansion

The runtime is intentionally being shaped as shared execution infrastructure. That means state ingestion, graph ownership, route catalogs, evaluation, queues, replay, persistence, and analytics need stable contracts before every future strategy receives specialized implementation. This slows the first strategy slightly and improves the platform.

21.3 Deterministic Evidence Before Live Optimization

Performance optimization without replay and metrics tends to create folklore. Salus treats retained metrics, replay artifacts, dry-run outputs, and bounded live evidence as prerequisites for serious tuning. That is why statistics are part of the architecture rather than an afterthought.

21.4 Conceptual Roadmap

Future work should build on the same ownership model. The natural evolution is not a collection of separate systems, but a broader platform where execution research, historical simulation, route analytics, additional strategy surfaces, solver-style infrastructure, inventory-aware execution, and capital management reuse the same runtime foundations.

The constraint is that future capabilities should not collapse boundaries. New strategies should consume shared state, evaluation, execution, replay, and analytics contracts. They should not bypass them.

22 Terminology

Term	Public-safe definition
Route	A path through known market components that can be evaluated as one execution question.
Route Catalog	The runtime lookup surface for known routes, route ids, and route membership.
Route Scope	The subset of routes selected for a specific evaluation or diagnostic pass.
State Frame	The labelled view of current component state available to the runtime for a block or replay source.
Coverage	Whether the current state frame contains enough decoded state to evaluate a route.
Freshness	Whether the state, route work, and execution decision still correspond to the relevant live moment.
Execution Candidate	A route evaluation result that has passed enough checks to be considered by the execution pipeline.
Preflight	A final safety check before live submission or before classifying a candidate as executable.
Replay	A deterministic or labelled re-run used to understand behavior, validate changes, or explain outcomes.
Artifact	A durable output such as a metric, route summary, replay record, or diagnostic file.
Strategy Module	A policy layer that asks questions of the shared runtime without owning the runtime itself.
Runtime Boundary	An ownership line between subsystems that defines what state, decisions, and evidence each subsystem controls.

Table 9: Terminology reference. These definitions describe architecture, not proprietary strategy mechanics.

23 Conclusion

Salus is best understood as an execution architecture. The interesting engineering work is not only finding a route. It is maintaining current state, mapping liquidity into an inspectable graph, evaluating routes under bounded latency, preserving queue freshness, gating execution safely, replaying failures, and turning runtime behavior into evidence.

The architecture is built around ownership because ownership is what lets the system evolve. Market data, graph topology, route catalogs, evaluation, execution, replay, persistence, and analytics each answer different questions. Keeping those questions separate is what allows the platform to support more than one execution model while remaining understandable to engineers.

Modern trading infrastructure is therefore not simply a trading problem. It is an execution-systems problem: reusable infrastructure, evidence-driven engineering, and durable architectural boundaries are what make long-term platform evolution possible.

24 About The Author

John Whitton is an engineering leader and hands-on systems builder focused on trading infrastructure, DeFi, distributed systems, financial infrastructure, Rust, and AI-native software development. His recent work has focused on building and documenting execution systems where architecture, performance, correctness, and operational clarity matter.

- https://johnwhitton.com
- https://portfolio.johnwhitton.com
- https://resume.johnwhitton.com
- https://github.com/johnwhitton

This publication consolidates existing Salus architecture notes, implementation walkthroughs, validation records, and performance evidence into a public-safe external architecture reference. It describes engineering design and tradeoffs, not proprietary trading mechanics.