# Building a High-Performance Trading Execution Platform

## Engineering Case Study using the Salus Research Platform

John Whitton

External Engineering Publication – Version 1.0

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

---

**Abstract.** Salus is a Rust-based research and execution platform for latency-sensitive trading infrastructure. It was built around a simple systems premise: finding an opportunity is not enough. A production execution system must determine whether an opportunity can still be evaluated, simulated, submitted, and settled before the market state changes. That execution problem is the center of the architecture.

This case study describes the engineering problems behind that premise: live market-data ingestion, graph construction, route cataloging, simulation, execution gating, replay, observability, and performance tuning. It also describes how those pieces are becoming shared infrastructure for multiple execution models. It is an external engineering publication, not an implementation guide. Readers should learn how the engineering problems were approached, not how the trading system makes money.

**Narrative flow.** This paper follows one story: the world moves while the system is deciding what to do. The architecture, performance work, replay tooling, and platform direction all follow from that fact. Salus is evolving from an arbitrage-oriented execution engine into a generalized trading execution platform.

## 1 Why This Exists

Modern trading infrastructure is a systems problem before it is a strategy problem. The runtime has to ingest changing market state, decide which parts of the route surface are worth evaluating, simulate candidates against current state, guard execution with explicit safety checks, and leave enough evidence to explain what happened afterward.

Salus was built as a research platform for that class of problem. The system is Rust-based, integrates live Tycho stream data, uses Alloy-based chain interaction through adapter boundaries, stores inspectable read models in PostgreSQL, and treats replay and metrics as first-class engineering tools. The architectural goal is broader than any one trading workload: shared execution infrastructure that can support more than one strategy family.

The goal of this publication is to consolidate the best existing Salus architecture notes, portfolio diagrams, walkthroughs, and design records into a single public explanation. It deliberately does not summarize every subsystem equally. Five well-developed engineering ideas are more useful than twenty shallow descriptions:

- By the time an execution decision has been made, the market state may already have changed.
- Route discovery, evaluation, execution, ordering, and analysis are separate engineering problems.
- Runtime boundaries matter because queues, adapters, and services have different responsibilities.
- Replay and metrics turn failures into diagnosable evidence.
- Performance work is meaningful only when measured against runtime scale.
- The runtime is evolving toward multiple execution models on shared infrastructure.

## 2 The Moving-World Problem

The core problem can be stated plainly:

> Finding an opportunity is not enough. A production execution system must determine whether an opportunity can be evaluated, simulated, submitted, and settled before the market state changes.

That sentence forces an architecture with more states than "found" and "executed." A route can be discovered, made eligible for evaluation, simulated, accepted by a profitability gate, converted into an execution candidate, blocked by stale state, rejected by preflight, broadcast, reverted, confirmed, or replayed later for diagnosis. Collapsing those states into a single boolean makes the system easier to demo and harder to operate.

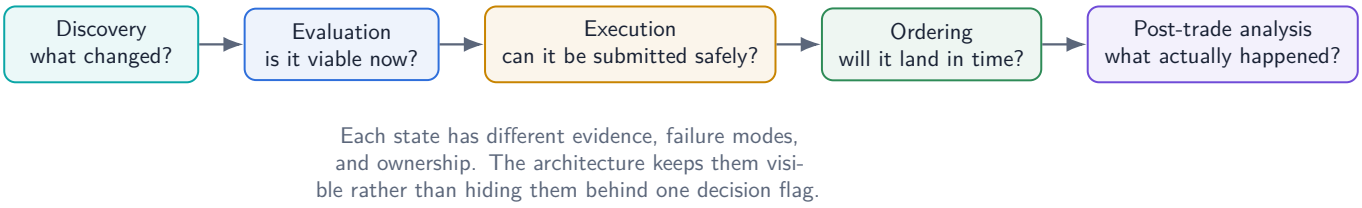


**Figure 1:** The problem is not route discovery alone. Each stage asks a different engineering question.

This distinction also changes what counts as a good engineering result. A missed execution may reveal stale state, insufficient ordering, unsupported protocol coverage, a simulation mismatch, or an observability gap. Those are not the same failure, and they should not produce the same remediation.

## 3 From Solver To Salus

The system did not start as a generalized execution platform. It evolved through a series of practical constraints. Solver made the opportunity surface visible. Salus turned that work into a Rust runtime with clearer ownership boundaries, better state handling, and replayable execution evidence. The next step is not "more code around the same strategy." It is a broader research and execution platform where new strategies, simulations, and post-run analysis can share the same engineering foundation.


**Figure 2:** System evolution. Salus is best understood as a step from opportunity detection toward reusable execution and research infrastructure.

That evolution matters because it changes what the platform is optimizing for. The early question was whether a route existed. The later question was whether the system could explain, reproduce, and improve the decision path that led to an execution attempt. Replay, solving, liquidation strategies, market making, and execution research are natural consequences of those boundaries. They are not unrelated future features; they are different consumers of the same execution runtime.

## 4 The Shape Of The System

At a high level, the platform turns live market data into explainable execution decisions:


**Figure 3:** Conceptual Salus pipeline, consolidated from the portfolio runtime walkthrough and the Salus design documentation.

The pipeline is intentionally conceptual. It does not disclose proprietary scoring rules, active execution policy, infrastructure details, or strategy parameters. What matters for this case study is the systems shape: streaming input, inspectable state, graph and catalog ownership, bounded evaluation, explicit execution gates, and replayable evidence. That shape is intentionally strategy-agnostic: the same runtime primitives can support more than arbitrage.

# 5 The Boundaries That Made It Work

The key architectural decision was to separate responsibilities that change for different reasons. Command wiring, chain configuration, stream ingestion, storage, graph topology, evaluation, execution, queues, and metrics all have different failure modes. Treating them as one runtime would make the system shorter, but harder to reason about when something goes wrong. Keeping those boundaries explicit is what lets the platform evolve from a strategy-specific engine into shared execution infrastructure.

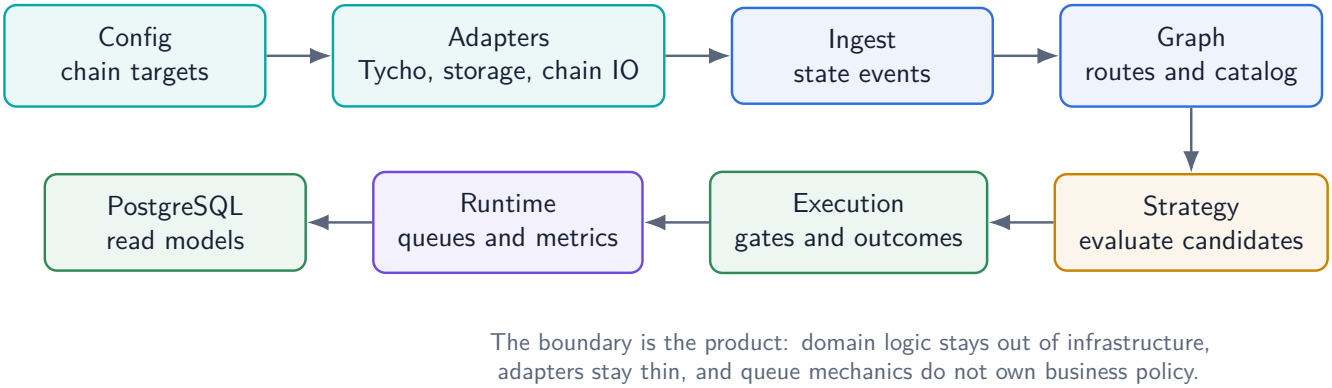


**Figure 4:** High-level ownership map, synthesized from `A2_3_Salus_Architecture`, `salus-architecture-overview`, and `docs/DESIGN.md`.

## 5.1 Making State Useful Without Making It The Runtime

Live state enters through a Tycho simulation stream adapter. The stream carries topology updates, protocol state deltas, block metadata, and provenance. The runtime applies those deltas into a stream-owned simulation frame and evaluates routes only when the current state needed for that route is present.

PostgreSQL is used as an inspectable read-model boundary. It stores tokens, protocol components, component-token relationships, graph metadata, route summaries, and route legs. The database is not the home for worker-local caches, queue internals, transient per-block frames, or execution decisions. That split keeps persistent state reviewable without making the database responsible for the live runtime.

## 5.2 Managing Millions Of Routes Without Rebuilding Everything

A long-lived graph allows token and component topology to survive beyond a single block. Route summaries can warm-start the runtime, while component changes can identify which known routes should be considered again. The important public point is not a private scoring method. It is that the system separates topology, candidate scope, simulation, profitability, and execution promotion.

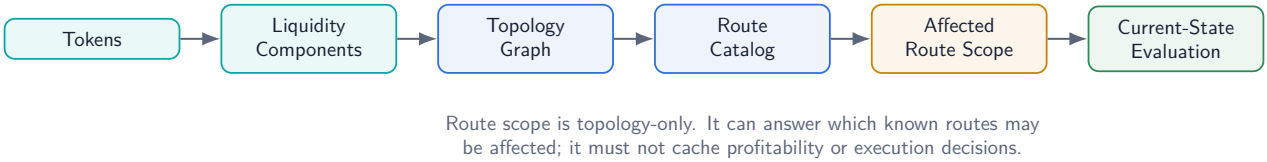


**Figure 5:** Graph and catalog lifecycle. The public-safe explanation focuses on topology and ownership, not proprietary scoring.

## 5.3 Proving A Candidate Is Still Executable

Execution starts only after evaluation and dry-run candidate construction. The runtime keeps the safety states separate: profitability, candidate acceptance, freshness, transaction construction, gas planning, final preflight, broadcast, and outcome classification.


**Figure 6:** Execution gate model. A profitable candidate is evidence, not proof of executable settlement.

# 6 What The Numbers Show

Performance matters because the route surface is large and the market state is moving. Retained runtime statistics show three useful dimensions: graph size, route-universe size, and evaluation throughput.

| 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 1:** Aggregate runtime statistics from retained Salus run artifacts. These are scale indicators, not trading-edge disclosures.

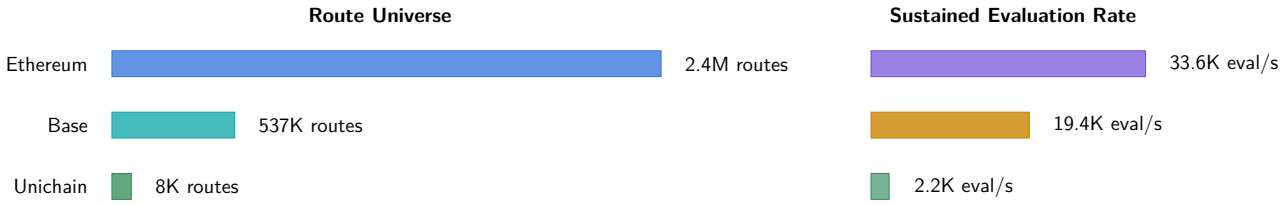


**Figure 7:** Scale and throughput visualized from retained runtime statistics.

These numbers are less interesting as benchmarks than as evidence that the architecture scales. They show that the runtime can evaluate large route surfaces while still retaining enough instrumentation to explain what happened to each execution decision.

The chains also behave differently. Ethereum has a much larger graph and route surface. Base showed why stale work handling matters: a runtime can admit more work than it should finish if a newer block supersedes the evaluation window. Unichain is smaller, but it is still useful as a validation target because it exercises the same pipeline shape at a different scale.

# 7 What Turned Out To Be Hard

## 7.1 Evaluating A Large Route Surface Quickly

Route surfaces grow quickly as graph connectivity increases. The architecture responds by separating route catalog construction, affected-route recall, evaluation request preparation, worker execution, and finalization. That lets the runtime measure where time is spent instead of treating evaluation as one opaque block.

The design also avoids pretending that broad evaluation is always the right operating profile. Wide no-throttle runs are useful diagnostics; production execution needs bounded work, freshness, and predictable latency.

## 7.2 Reasoning About A Moving World

Execution systems cannot assume that market state remains static while decisions are being made. A candidate can be valid against the source state and invalid by the time a final preflight runs. The execution system treats freshness, preflight, submission, and receipt outcome as separate evidence states, which makes stale execution visible instead of silently conflating it with strategy failure.

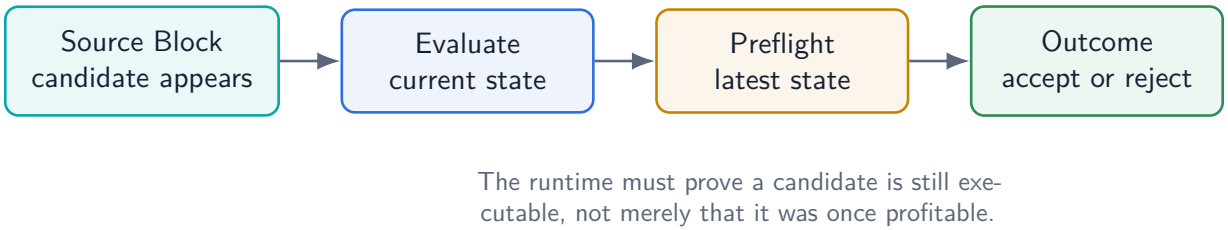


**Figure 8:** Freshness model. The source state and execution state are not assumed to be identical.

### 7.3 When Many Candidates Are Really One Event

Many apparently distinct candidates can represent the same underlying market movement. A route catalog can expose several paths through related components even though the opportunity family is effectively one timing problem. That drives two design choices: keep candidate generation explainable, and make post-run artifacts good enough to identify repeated opportunity families without revealing the selection logic.

### 7.4 Using Queues Without Hiding Policy

Queues are necessary because route preparation, evaluation, persistence, and execution have different latency and ownership profiles. But queues are not a place to bury business logic. The queue wrapper is generic, the queue adapter owns mechanics, and the business service owns policy.

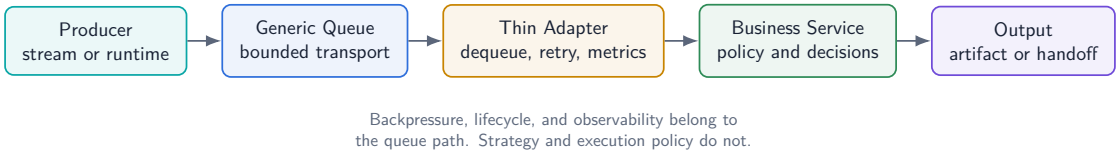


**Figure 9:** Queue ownership pattern, consolidated from the queue documentation and runtime hardening work.

### 7.5 Turning Replay Into Evidence

The biggest change in the platform was not another optimization. It was treating replay as a first-class architectural primitive. Once execution attempts could be replayed against retained state, debugging shifted from speculation to evidence. Replay now supports more than defect diagnosis: it gives the platform a way to understand execution behavior, analyze outcomes, tune performance, validate strategies, and support future research. It helps distinguish model error from timing error, source-state validity from later-state invalidity, and simulation mismatch from ordering failure.

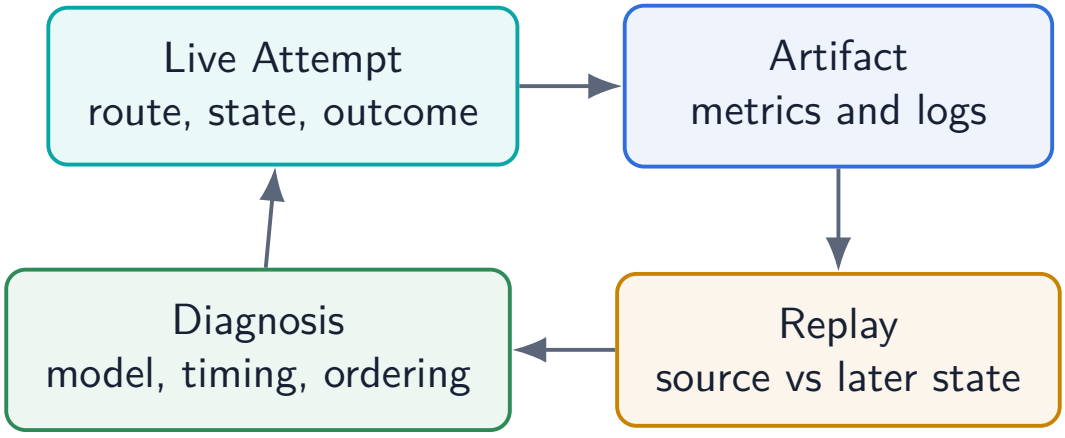


**Figure 10:** Replay feedback loop. Replay is not only a debugging tool; it is one of the primitives that lets the platform learn from execution.

## 8 Engineering Principles

**Build for explanation, not only performance.** A fast execution system that cannot explain itself is difficult to operate. The platform therefore treats every execution decision as something that should leave evidence: what state was observed, what was evaluated, what was rejected, what was submitted, and what happened afterward. Performance matters, but explainability is what makes performance trustworthy.

**Replay before optimization.** Optimization is only useful when the team can reproduce the behavior being improved. Replay turns an execution attempt into a durable object of study rather than a transient log event. That changes the engineering loop: first make the behavior observable and repeatable, then decide whether the right fix is faster evaluation, better freshness handling, improved ordering, or clearer model boundaries.

**Evidence before intuition.** Trading systems invite plausible stories. A missed execution can look like a strategy problem when it is really stale state, a simulation mismatch, a queue delay, or an unsupported protocol edge. The runtime is designed to make those differences visible. Measurements, artifacts, and replayable outcomes should settle engineering arguments before intuition hardens into policy.

**Architecture before strategy.** The architecture is valuable because it separates the reusable execution problem from the particular strategy that happens to use it first. State ingestion, graph ownership, evaluation, execution gates, queues, observability, and replay are platform concerns. Strategy logic should consume those capabilities without owning the infrastructure that makes them reliable.

**Separate ownership from execution.** Clear boundaries keep the system understandable under pressure. Adapters translate external systems, queues move work, services own policy, and the runtime coordinates lifecycle and evidence. That separation is not ceremony. It prevents a local implementation shortcut from becoming an implicit rule about how the platform behaves.

**Shared infrastructure before duplication.** Long-term platform evolution depends on resisting one-off execution paths. When new strategies need replay, analytics, simulation, or execution gates, the best answer should be to improve the shared runtime rather than create parallel machinery. That principle keeps the platform coherent as the strategy surface evolves.

## 9 What Changed My Thinking

**I expected route discovery to be the hard part.** It was only one part of the system. Discovery creates candidates; it does not prove current profitability, safe submission, early ordering, or realized settlement.

**Execution timing mattered more than the first version assumed.** A route that is valid at the source state may be invalid by final preflight or settlement. Systems need explicit freshness and outcome evidence.

**Many candidates can represent one underlying event.** Summing candidate-level observations naively can overstate what the runtime really found. Artifacts need enough structure to group related behavior without exposing proprietary selection rules.

**Submission path is not the same thing as execution quality.** Submission path is only one part of execution quality. The runtime still needs freshness, pricing discipline, preflight, outcome tracking, and replay.

**Replay changed debugging.** Without replay, teams are left arguing from logs. With replay, a failure can be classified as model mismatch, stale state, ordering loss, unsupported protocol coverage, or an operational gap. The same evidence loop also supports execution research and strategy validation.

**Metrics changed optimization.** Once evaluation service time, worker utilization, queue depth, stale work, and per-chain route surfaces are visible, performance work becomes empirical.

## 10 What I Would Build Differently Today

The main architectural direction still holds: separate state, graph ownership, evaluation, execution gates, queues, and replay. I would make the strategy-agnostic parts of the platform explicit even earlier, because the same execution runtime can serve more than one strategy family. The changes I would make today are mostly about making the evidence loop earlier and stronger.

- I would make replay artifacts a primary output from the beginning, not a hardening feature added after execution behavior became interesting.
- I would design the analytics model earlier, because every serious performance discussion eventually depends on retained evidence.
- I would represent opportunity families more explicitly in public-safe artifacts, so repeated candidates can be analyzed without exposing private selection rules.
- I would treat chain-to-chain differences as a first-class design input, not a portability detail discovered during validation.
- I would describe the shared execution boundary earlier, so future strategy work has a clearer architectural home.

## 11 What Comes Next

The current architecture is a base for broader execution infrastructure, not a claim that every future trading capability already exists. The next step is not a list of disconnected features. It is the continued separation of reusable platform capabilities from the specific strategies that consume them.

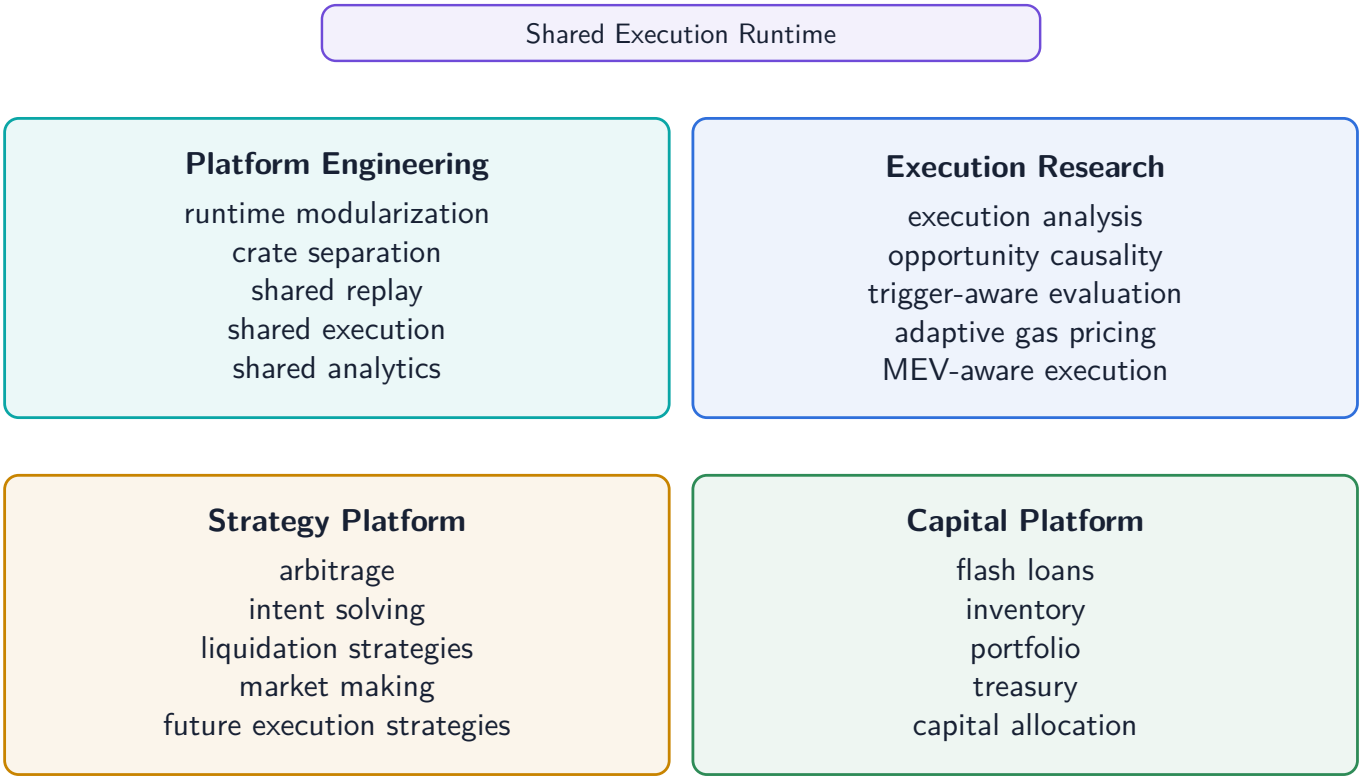


**Figure 11:** Future direction as platform capability groups. Each theme builds on the same execution runtime rather than becoming a separate system.

Several themes matter more than any one feature. Platform engineering keeps the runtime modular enough that replay, execution, and analytics can be shared. Execution research turns retained evidence into better analysis of what caused an opportunity and how an execution path behaved. The strategy platform leaves room for solving, liquidation strategies, market making, and future execution strategies to reuse the same runtime. The capital platform adds the accounting and allocation layer needed for strategies that depend on inventory, financing, portfolio exposure, or treasury constraints.

The important point is architectural: these capabilities should not become four separate systems. They should compound on the same execution runtime. Modern trading infrastructure is ultimately an execution problem: durable advantage comes from reusable infrastructure, evidence-driven engineering, and a platform that can support multiple execution strategies as the market and the research agenda evolve.

## 12 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 portfolio material, architecture assets, and internal design records into a public-safe engineering narrative. It intentionally describes engineering problems, architecture, tradeoffs, and lessons rather than proprietary strategy mechanics.