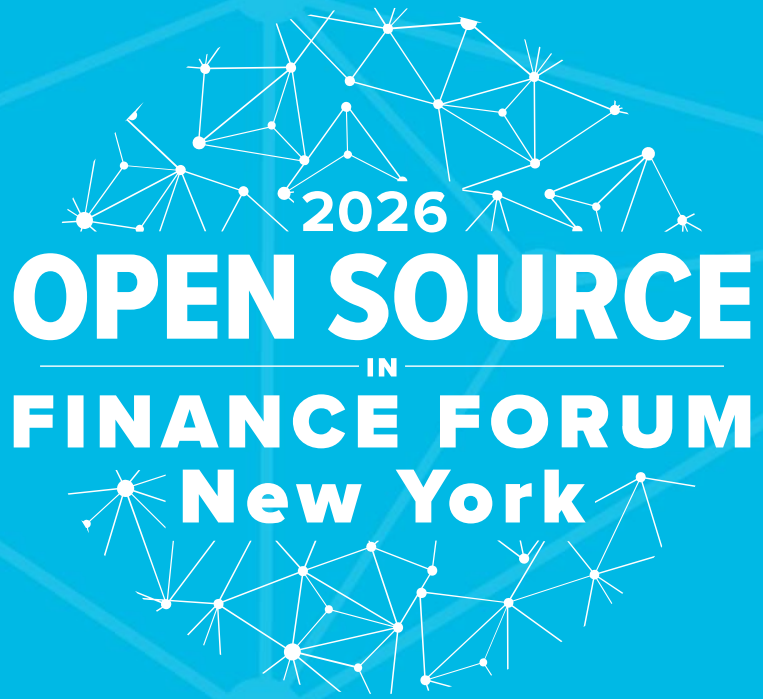


Fulfilling the TraderX Vision

How undergraduates used spec-driven development and LLM agents to grow FINOS TraderX into a consensus-replicated, multi-asset trading venue, and plugged in a separately developed risk engine

Judah Diament · **Yaakov Seif** · **Alex Neugroschl**
diament@yu.edu · aseif@mail.yu.edu · aneugro1@mail.yu.edu

Yeshiva University · Yeshiva College Computer Science

Goals, Partners, Timeline

Goals

1. **Find the limits** of how close to a production-quality system can be built by students who are well-trained in Distributed Systems and/or AI, using Spec Driven Development
2. **Integrate real-world financial systems knowledge** and challenges into our curriculum, so that **our C.S. grads can "hit the ground running" in financial firms**

Partners

Dov Katz
TraderX/FINOS

Rich James
Google Cloud

Timeline

- **Summer – Fall 2026: initial projects launched, experimentation**
- **Fall 2026 – Spring 2027: making risk calculations faster & cheaper using JAX & Google TPUs**
- **Spring 2027: all senior capstone projects focused on financial systems**
- **2027 – 202?:** more financial systems work added to courses, e.g.:

1. IB end-to-end coverage in Advanced Distributed Systems – data platforms, trading and execution, pricing and valuation, research & back testing.
2. Finding alpha, trading strategy development & backtesting, trading agents in AI / ML courses
3. HFT C++ development in Programming Languages, Compilers
4. Regulatory, Security, and Risk challenges in Modern Data Management, Cybersecurity, etc.

Outline

1. **What:** The Trading System
2. **How:** Spec Driven Development
3. **Where:** Deploying on the Cloud
4. **Extensibility:** Plugging in a Risk Engine

What: The Trading System

The Experiment: TraderX + Spec Kit + LLM Agents + One Student

TraderX was designed to do two things

1. **Radical reuse:** assemble scenario-specific trading systems from specs, one state at a time
2. **Open demo platform:** run a new component inside a complete, working trading system

Our test of both goals

- **Builder:** two undergraduate CS students, no prior finance background
- **Specs:** GitHub Spec Kit feature packs, extending the FINOS TraderX lineage (states 001–014)
- **Code, tests, reviews:** Claude Code and OpenAI Codex agents
- **Bar:** production-style requirements, checked through tests and bounded GKE proofs

18
spec-driven states
YU01–YU18, each generated from its spec pack

2.41M
orders/s, matching engine
in process, one thread, c4d

400K
orders/s over FIX and Binary
per order, 18 gateways, GKE

~0.63 s
leader election under load
100K/s, 0 acked orders lost

961
Java unit cases passed
YU18 local sweep · 6 skipped

1
external risk engine
JAX/ORE engine run in Google Cloud

Jun–Oct
YU01 → YU18 · Jun 9–Oct 22, 2026

1,354
canonical commits beyond origin/main

623
requirement IDs · 191 success-criterion IDs

From Reference App to Replicated Trading Venue

FINOS TraderX baseline

teaching reference app (state 009)

- Angular web UI
- trade-service (REST)
- order-matcher (polling)
- NATS messaging
- trade-processor · position-service
- account · reference-data · people
- PostgreSQL

After YU01–YU18 - deployed on Google Cloud

Order entry
REST · FIX 4.4 · binary (SBE)

Scalable gateway tier
pre-trade risk checks at ingress

Aeron Raft cluster ×3
LMAX engine · crossing book · OTC book · journal + snapshots

Committed egress
leader trade bridge → NATS

Async projection
trade-processor · position-service

Read model + consoles
MariaDB · Angular Trader Desk

Market data
TAQ replay · FRED curves

Algo engine
TWAP · VWAP slicing

Historical data services
Private TAQ archive + Parquet catalogue

Observability
OTel · Grafana · Tempo

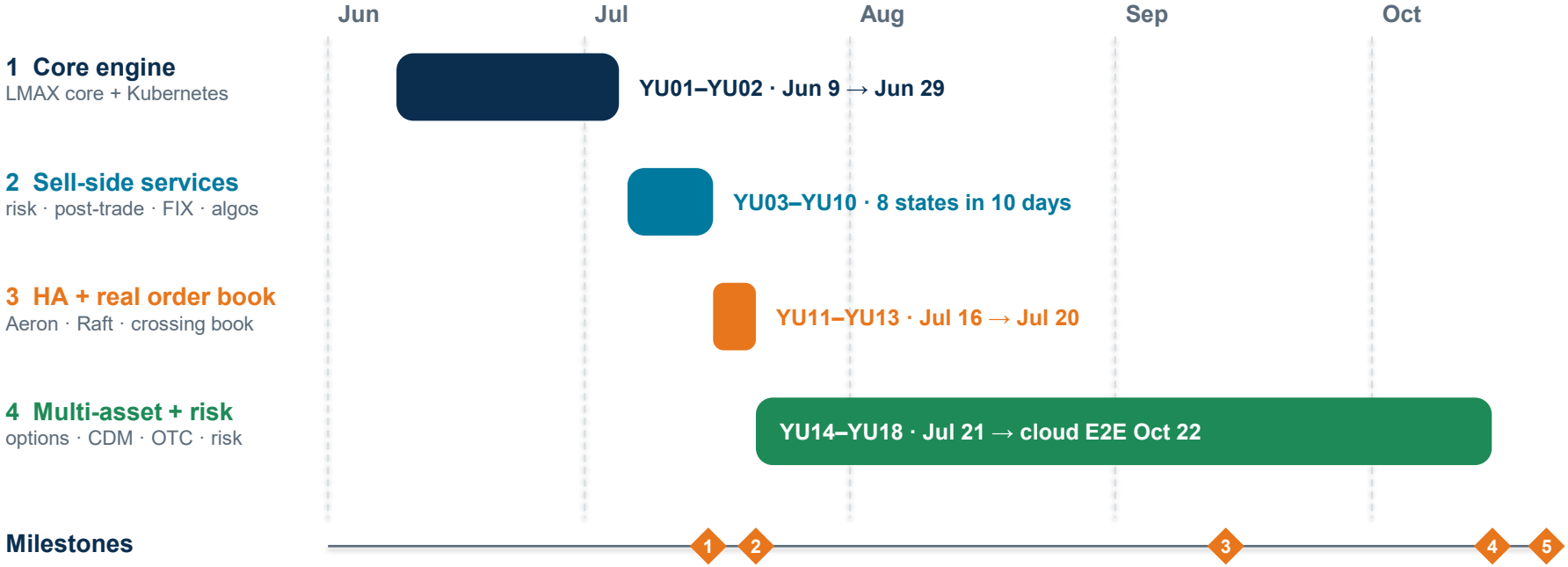
End-of-day risk boundary

EOD cut at log position N
byte-identical on all members

Write-once bundle
Google Cloud Storage + manifest

JAX risk engine
Cloud Batch job (Alex's code)

Four Project Stages, Eighteen States, June Through October



1 **Jul 18** Raft failover on GKE, no k8s in the loop

2 **Jul 21** 136M-order soak, no throughput decay

3 **Sep 14** EOD risk bundles + mock consumer

4 **Oct 15** Cloud E2E: cut priced by JAX engine

5 **Oct 23** 400K orders/s per order, binary + FIX

Stage 1 · YU01–YU02: An LMAX-Style Matching Core on Kubernetes

Open source used

- **LMAX Disruptor 4.0**
- **Agrona (off-heap, lock-free)**
- **Java 21; Epsilon test profile**
- **Spring Boot 3.5**
- **Kubernetes · GKE**
- **NATS**

What was built



- **Replaced TraderX's polling matcher** with a deterministic, single-threaded, in-memory engine
- **Event sourcing:** journal + snapshots are the truth; SQL is only a projection
- **Fixed-point prices, deterministic replay,** ID continuity across restarts
- **Zero-allocation hot path,** checked by allocation gates and dedicated no-GC test runs
- **Kubernetes runtime:** probes, persistent journal volumes, routing on GKE

Spec → code **601 lines** of spec, 64 requirements → **8,526 lines** Java + 3,446 test

Measured

2.41M orders/s
single-thread engine, in process
c4d-standard-8 (AMD EPYC 9B45), 2 dedicated cores, Java 21; same in all 3 repeats

440–631 ns
match p99 (limit cross)
p99.9 ≤ 771 ns; Oct 2026

0 bytes
allocated per order, steady state
4 allocation gates on every push + 2 Epsilon no-GC runs

Stage 2 · YU03–YU10: The Sell-Side Services a Desk Needs

Open source used: NATS JetStream · MariaDB · PostgreSQL · QuickFIX/J (FIX 4.4) · kdb+ / KDB-X (q) · DuckDB · Angular

YU03 — Pre-trade risk gateway
Credit, size, notional and price-collar checks; kill switch (15c3-5 style), in memory

YU04 — Durable control feeds
Risk limits and controls survive a replica outage (JetStream)

YU05 — Post-trade compliance
T+N settlement, reconciliation, TCA, entitlements, reproducible regulatory export

YU06 — EOD price production
Official closes, quality gate, override and publish

YU07 — Historical tick store
kdb+ over NYSE TAQ; 2.66 TiB corpus in Cloud Storage

YU08 — Execution algo engine
TWAP and VWAP parent orders sliced into child orders

YU09 — Ops hardening
Secrets, probes, DB warm-up, recovery manifests, DR runbooks

YU10 — FIX 4.4 ingress
Standard FIX sessions alongside REST and binary order entry

Spec → code **1,229 lines** of spec, 170 requirements, 53 success criteria across 8 states → **14,156 lines** of production Java + 6,096 lines of tests

Proof with real market flow

NYSE TAQ, 3 Mar 2025 open (09:30–10:00), 7 symbols, replayed at 5× through the replicated book

1,138,793 real prints ($18.2B notional)

2.28M orders accepted, 0 failures

0.019% rejected by the risk gate

0 open book empty and identical on all 3 members after

Stage 3 · YU11–YU13: Consensus Replication and a Real Order Book

Open source used

- Aeron 1.51 (UDP transport)
- Aeron Archive
- Aeron Cluster (Raft)
- SBE encoding
- Agrona collections
- GKE anti-affinity

What was built

YU11 — **Aeron replication**

Replaced file-backed NATS replication: 520k vs 10.6k events/s (~49×); cross-epoch recovery

YU12 — **3-member Raft cluster**

The deterministic engine becomes a replicated state machine; a minority cannot lead or admit orders; empty-disk rejoin; REST and FIX clients recover across leader failover

YU13 — **Crossing limit order book**

Price-time priority, partial fills, market orders, self-trade prevention, atomic cancel/replace; O(1) intrusive price-level queues

Spec → code **634 lines** of spec, 82 requirements → **16,488 lines** Java + 7,505 test

Measured

~0.63 s
leader election under load
leader JVM SIGKILLed at 100K orders/s, 3 arms: 0 acked orders lost, survivors agree

400K orders/s
per-order, committed
18 gateways; p99 4.82 ms binary, 3.93 ms FIX; all replay-verified

~340 µs
consensus commit (p99)

1.44B orders
1 hour at 400K orders/s
p99 3.82 ms; 0 rejected, missing or duplicate; snapshot + tail replay exact

Stage 4 · YU14–YU18: Multi-Asset, Observability and the Risk Boundary

Open source used

- FINOS CDM instruments
- OCC option symbology
- FRED Treasury curves
- OpenTelemetry
- Grafana · Tempo · Loki
- Prometheus
- Google Cloud Storage

What was built

YU14 Listed equity options: OCC symbols, ×100 multiplier-aware risk; snapshot format bump, replay-proven

YU15 EOD risk extract: a sequenced cut marker names consensus position N; write-once to Cloud Storage

YU16 CDM instruments: ETFs, US Treasuries (bills to STRIPS), corporates; bonds priced as fraction of par

YU17 OTC rates: swaps and swaptions booked as contracts; sequenced FX/rates; sandbox replay venue

YU18 Risk integration: 7 order types (stop, iceberg, pegged, trailing…), EOD risk bundles, risk-service packaging

Spec → code **2,553 lines** of spec (YU14–18), 307 requirements → **21,652 lines** Java + 17,249 test

Measured

6
instrument classes, one ticket
equity · option · Treasury · corporate · swap · swaption

Byte-identical
EOD cut on every member
and again on replay; SHA-256 manifest travels with the bundle

7
order types
market · limit · stop · stop-limit · iceberg · pegged · trailing stop

Takeaway #1: Feature-Rich Trading Systems Built

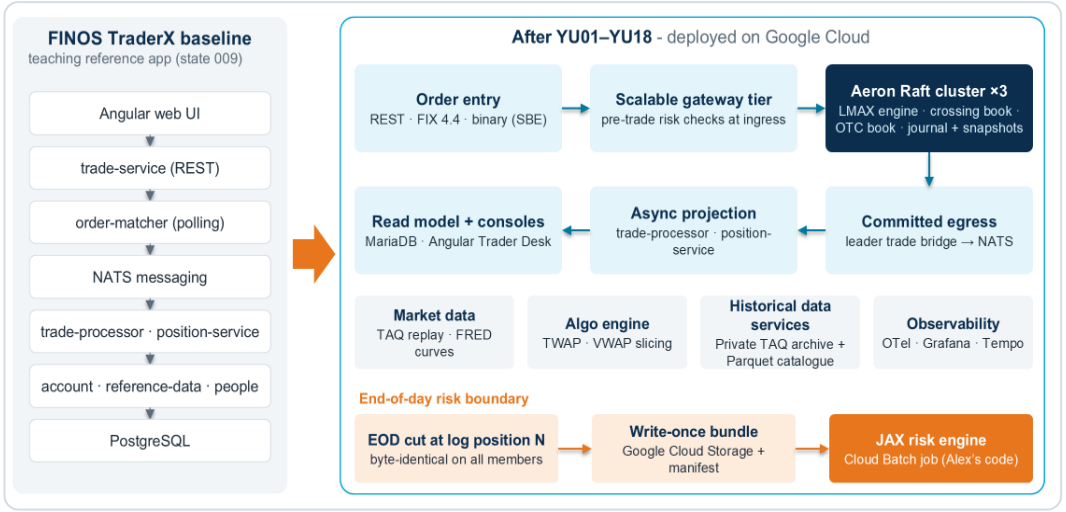
One well-trained undergraduate student was able to build a **feature-rich trading system** in a few months

1. **Core engine**
   LMAX core + Kubernetes
2. **Sell-side services**
   risk · post-trade · FIX · execution algos
3. **HA + real order book**
   Aeron · Raft · crossing book
4. **Multi-asset + risk**
   options · CDM · OTC · risk

How: Spec Driven Development

SDD + Layered States == Full Flexibility Without Rework

- A goal of TraderX is to serve as a **learning and integration platform** for varying architectural and functional states and requirements.
- In the **“old world”,** rearchitecting a system for alternate use cases required creating **an entirely new project** with maintainers, etc.
- By describing new functionality / states as **deltas on existing states**, TraderX allows one to learn and experiment with alternatives **without a fork or rework**.



The YU system is a series of layers built, one state / function at a time, as **deltas on top of TraderX’s state 009**.

Who Does What: Spec Kit, LLM Agents and the Developer

	1 Specify	2 Plan	3 Tasks	4 Implement	5 Verify
The Developer	Chooses the capability; states behavior, limits and failure cases	Chooses among designs; signs off each ADR	Orders and scopes the work	Reviews diffs; rejects plausible-but-wrong code	Demands evidence from the running system; approves merges
GitHub Spec Kit	spec.md template: stories, FR / NFR, success criteria	plan.md, research.md, data-model, contracts	tasks.md checklist tied to requirement IDs	Pack drives generate-state for the layer	Success criteria become the acceptance gates
LLM agents	Draft requirements; flag gaps and ambiguities	Propose designs; rival agents critique each other	Break the plan into tasks	Write code and tests (Claude Code: Opus, Fable, Sonnet; OpenAI Codex)	Write proof scripts; newer models re-audit older work

Division of labor in numbers: 1,354 canonical commits · independent design and code reviews · scoped integration approved by the student

One Feature, End to End: The Crossing Limit Order Book (YU13)

1 Spec Kit spec

spec.md - 165 lines, 10 FR, 4 NFR, 6 SC

User story: *"As a trader, I want price-time priority - better-priced resting orders execute first, and at equal prices the earlier order executes first."*

FR-LOB02: A marketable order SHALL execute against resting opposite-side orders best price first and FIFO within a price level; each match step SHALL fill min(aggressor, resting remaining) at the RESTING order's price.

2 Design + tasks

ADR-049/050/057 · tasks.md - 16 tasks

ADR-050: price levels are array-indexed tick slots on a 0.001 grid inside a banded window, holding intrusive FIFO queues of pooled orders → O(1) add, cancel, match, zero allocation

T-LOB04: add LimitBook (levels, FIFO queues, occupancy bitmap)

T-LOB05: rework MatchingEngine into the crossing engine

T-LOB09: crossing-semantics unit tests (price-time, partials, replay)

3 Agent-written code

MatchingEngine.cross() - the matching loop

```
while (a.remaining > 0 && opp != NO_LEVEL) {
  levelPx = book.priceAt(opp);
  if (!market && !crosses(a, levelPx)) break;
  RestingOrder r = book.headAt(restingSide, opp); // FIFO
  if (r.accountId == a.accountId) {   // STP, ADR-057
    preventSelfTrade(r, e, book); /*...*/ continue; }
  int qty = Math.min(a.remaining, r.remaining);
  applyMatchFill(r, qty, levelPx, ...); // resting px
  applyMatchFill(a, qty, levelPx, ...);
  opp = buy ? book.bestAskSlot() : book.bestBidSlot();
}
```

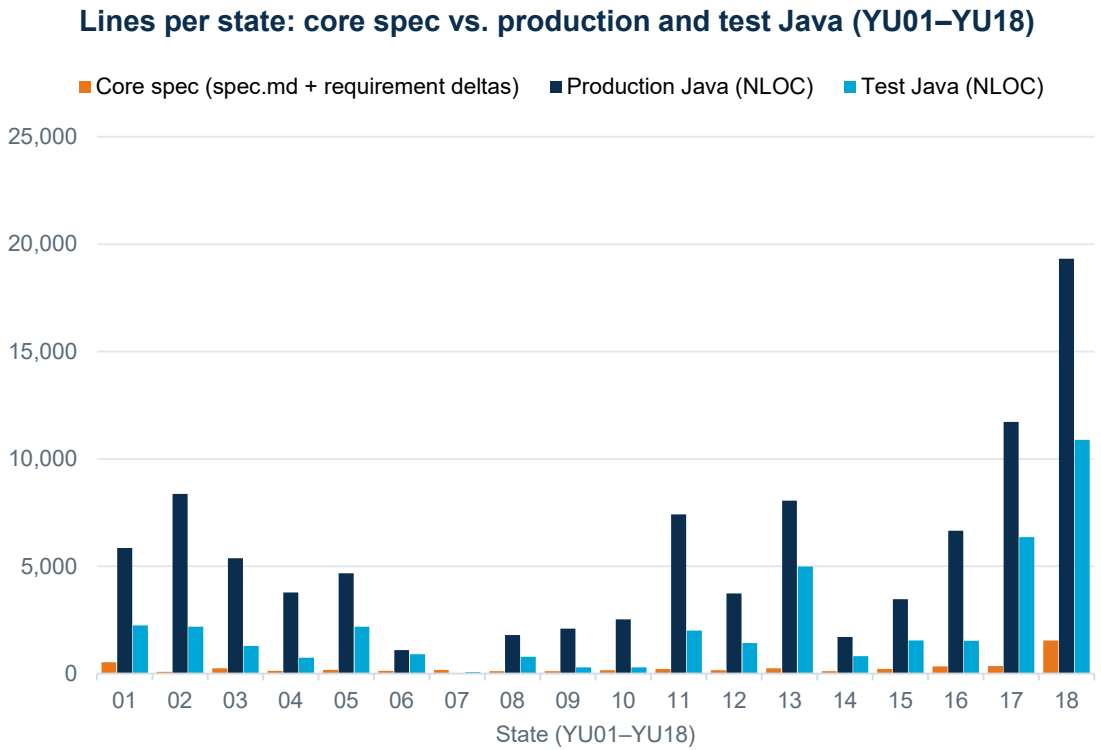
4 Test (student-reviewed)

LimitOrderBookTest - 31 tests

```
sellLowFirst  = limit(ACCT,  SELL, 100, 150.00); // 1st
sellLowSecond = limit(ACCT2, SELL, 100, 150.00); // 2nd
sellHigh      = limit(ACCT,  SELL, 100, 150.01);
buy = limit(ACCT3, BUY, 250, 150.01);
assert FILLED(sellLowFirst), FILLED(sellLowSecond);
assert firstRestingFill == sellLowFirst;   // FIFO
assert sellHigh PARTIALLY_FILLED 50 @ 150.01;
assert FILLED(buy);   // swept two price levels
```

Size of this state: 253 lines of spec + requirement deltas → 8,058 lines of production Java + 4,991 lines of tests. Proven on GKE: 136M-order soak, real TAQ replay, failover with the full book in the snapshot.

How Big are the Specs Compared With the Code They Produce?



1 : 19.5

core spec line : production Java line

5,017 lines of spec + deltas → 97,643 Java NLOC

157 lines

of Java per requirement

623 documented requirement IDs across YU01–YU18

40,493

Java test lines across the state layers

Unit, integration and controlled protocol fixtures; non-Java tests are additional

Method: nonblank physical lines, including comments, per state layer. A replaced path appears in each owning state. YU18 includes component specs. Java overrides only; non-Java tests excluded. Oct 22 snapshot.

Testing: Four Tiers Plus Gates, Each Looking Somewhere Different

Task-specific regression and verification span all tiers

Tier 4 · Task-specific regression
recorded for each change
Bug reproductions, failure-path tests, script checks and fix-specific regressions

Tier 3 · E2E and running-system proofs
operator-run
orders → consensus → projections → SQL and extracts; attributable effects and failure cases; 3-member failover, snapshot/replay, empty-disk rejoin, wall-clock budgets

Tier 2 · Real infrastructure containers
real MariaDB with the deployed schema, real JetStream, browser clients

Tier 1 · In-process every push
961 Java test cases on the composed YU18 tree; CI runs the YU18 suites on every push

Plus: Alex's engine has 1,844 tests, incl. parity with ORE

Gates cut across every tier

4 zero-allocation gates (every push) + 2 runs on a JVM with no GC at all: the hot path may allocate **exactly 0 bytes**

Source/generated parity, input identity and negative controls on every cloud step

Coverage of what matters

All 18 core properties proven end-to-end (STP, idempotency, failover, replay, reproducible exports…) also have an in-process test gating every commit

Every benchmark run is replay-checked

The leader's Aeron archive is replayed through a shadow engine against client receipts; with receipts removed the check must fail, and does (Oct 2026, GKE)

Test Effectiveness: Defects the Tests Actually Caught

Defect	Caught by	Outcome
Gateway control feed switched to a lookup-only call: 276 catalog tickers failed and retried every second (15 of 534 applied)	Full-stack kind bring-up + book-retick proof	Registers again: 534 of 534 applied, 0 failures; proof passes
G1 pauses of 204–221 ms exceeded the 200 ms heartbeat timeout: elections during full-symbol EOD at 300K/s	EOD cloud test, 62 accounts × 1,000 symbols	50 ms pause target: 0 elections, one cut hash on all 3 members, rejoin re-renders it
Member journal grew ~108 GB/h: a 60 GiB volume fills in 34 minutes at 250K/s	Hour-long load runs on GKE	Purge behind snapshots: 1 h at 400K/s, archive ≤ 6.3 GB
A stale copy of a class in the YU14 layer broke compilation once YU13 added tracing	CI runs each ancestor branch (propagation legs)	Layer re-synced
Mixed-version rolling restart permanently diverged the deterministic cluster	Live roll on GKE	Compatibility-gated upgrade; fresh epoch for this incident
Gradle kept a green result after a DDL-only change, so a falsification “passed”	Falsifying EodSnapshotAndPnlIT	Schema declared as a task input

The rule that made agent-written tests trustworthy: break it first. Use targeted negative controls. Example: deleting the PRIMARY KEY from the deployed DDL makes the 2 idempotency cases fail; the other 9 pass.

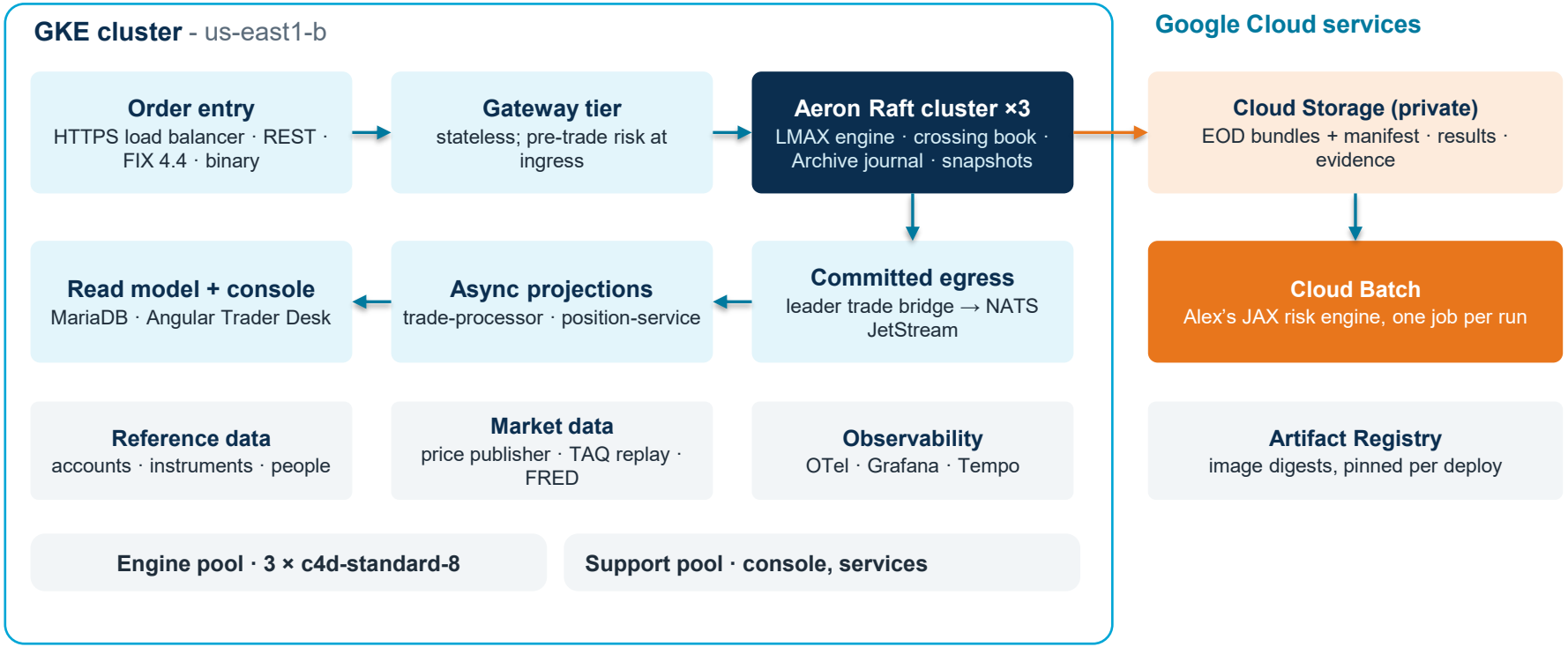
Takeaway #2: Spec Driven Development as the Enabler

1. **SDD scales** a person's efforts, allows for rapid learning and experimentation at a fraction of the effort and time as manual coding

2. **Layered state specs** maintain the integrity (and a complete record) of all existing work while enabling one to build out a graph of states which can be mixed and matched as required by pedagogical or functional goals

Where: Deploying on the Cloud

TraderX on Google Cloud: System Diagram



CI and Cloud Delivery: From a Push to the Cloud

	1 Trigger	2 Compose	3 Test	4 Gate	5 Output
TraderX · GitHub Actions	Push or PR to traderX-risk-integration	Render the YU18 effective tree (generate-state.sh)	Engine JUnit, composed services, EOD source + generated, console (Node, Angular)	Allocation gates; YU18 composition; every suite must have executed	JUnit artifacts; console image built and runtime-smoked
Risk engine · Cloud Build	Push to the engine repo's main branch	Cloud Build trigger	Build the engine container image	Image digest sent to Artifact Registry	TraderX pins a revision

Takeaway #3: End-to-End SDLC Possible w/SDD

1. SDD can be successfully used for a student to go from **planning → dev → CI → cloud deployment**

2. **Human expertise and supervision** needed to build the pipeline, tune, monitor – SDD is great, but the human is still needed

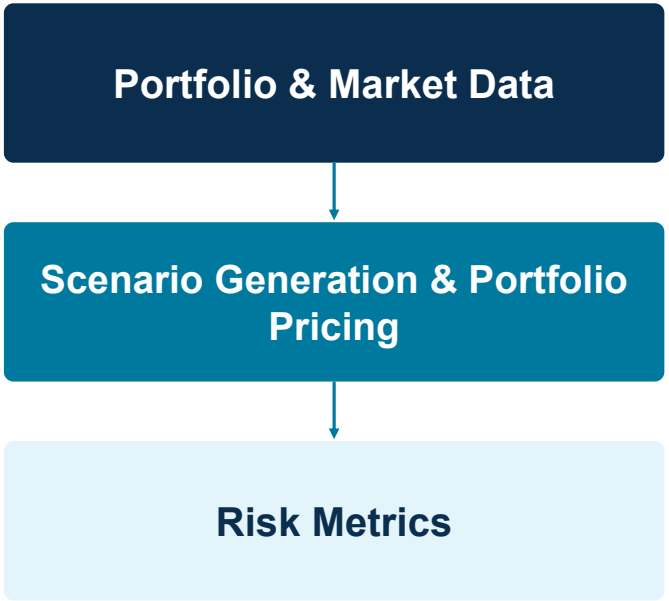
Extensibility: Plugging in a Risk Engine

Background

Computational demands of risk analysis

1. End-of-day risk requires substantial computation: portfolios are repeatedly repriced across many market scenarios.
2. These repeated calculations create parallel workloads suited to accelerator hardware.
3. Floating-point precision affects both numerical accuracy and hardware throughput.

End-of-day risk



Research Objectives & Google Cloud Support

Research objectives

1. Build an end-of-day risk engine that supports calculations across multiple numerical precisions.
2. Test the engine on a variety of financial instruments with different pricing methods and computational costs.
3. Determine whether risk can be computed reliably using more lower-precision simulations compared with fewer FP64 simulations.
4. Compare accuracy, execution time and cost across CPU, GPU and **TPU**, with a focus on **TPU** speedups.

Research support

Initial Grant from Google Cloud

Awarded by Google to Yeshiva University's Computer Science Department to support this research.

JAX & TPUs

Array computing

Python library for numerical array computing on CPU, GPU and **TPU** backends, supporting multiple floating-point formats and configurable matrix-multiplication precision.

JIT compilation

Uses just-in-time compilation through XLA to compile numerical kernels and reuse compiled programs for repeated workloads.

Vectorization and sharding

Supports vectorized calculations across market scenarios and sharding of the scenario axis across devices.

TPUs: Next-Generation AI Chips

TPUs are custom accelerators co-designed with open software to power the entire AI lifecycle—from frontier training and large-scale inference to the multi-step reasoning data demands of Agentic AI.

Risk Engine: Models and Calculations

Market setup: Discount/forward curves, swaption volatilities, FX spots, known fixings and trade conventions.

Calibration: Bootstrap LGM model volatilities to market swaption baskets, including the Hull-White parametrization.

Simulation: Correlated LGM/Hull-White rate paths, with FX paths for multi-currency exposure conversion.

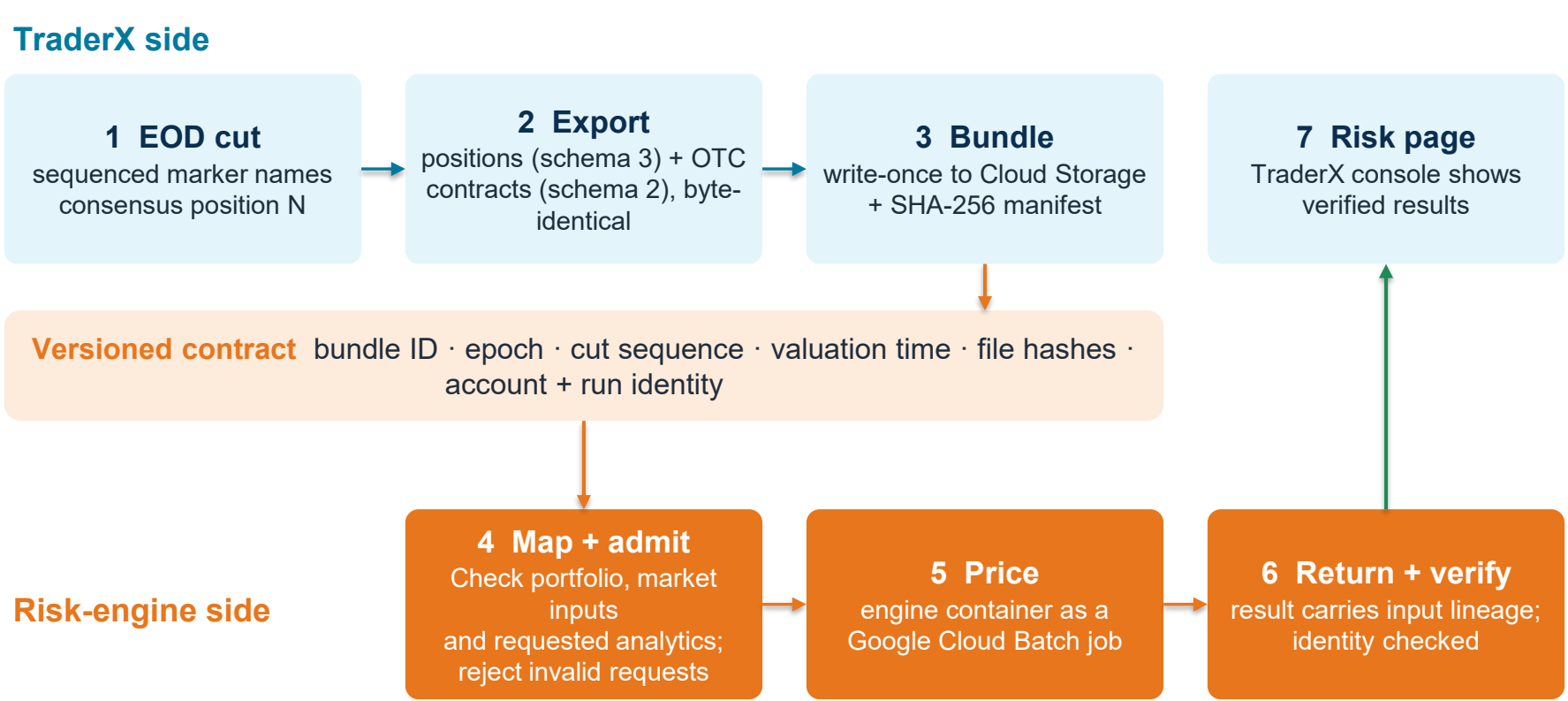
Scenario generation: Exposure uses Sobol sampling with a Brownian bridge; market risk uses Sobol-based Monte Carlo or historical rate-curve shocks.

Pricing: Discounted cash flows for swaps/Treasuries; European swaptions: Bachelier, or Jamshidian using a Hull-White model; LGM grid rollback for Bermudan/American swaptions.

Risk: NPV, exposure profiles (EPE/ENE/PFE), rate-curve VaR/ES; Delta/Gamma/Vega by bump and revalue or AD, and Theta by date roll.

Precision: Configurable numerical precision by calculation stage, with per-product controls for pricing.

Plugging in an Outside Component: The JAX Risk Engine



Takeaway #4: Extensibility

1. The TraderX approach to software architecture **proved extensible** enough to enable plugging in a component which was developed entirely separately from the main system

2. Extension required **coordination and communication at the API level** for handoffs to and from the extension

Key Takeaways

1 SDD rapidly increases the speed at which we move from spec to working code

One student extended 18 versioned states, reviewing requirements, designs, code and running-system evidence.

2 Layering is a powerful approach for learning and experimentation

Both learning and exploring alternatives for meeting functional requirements require trying out many options. The layered SDD/state approach enabled this.

3 TraderX can serve as the baseline for building realistic cloud-deployable systems

One student built TraderX up into a low-latency, fault-tolerant, feature-rich multi-asset trading venue which was successfully containerized and deployed on GKE

4 TraderX works as an extensible demo platform

An external risk engine ran on real TraderX positions with no pricing code copied in.

Thank you - questions?

Student work (branch traderX-risk-integration)
github.com/YU-CS-Summer-2026-Distributed-TraderX/traderX

TraderX
traderx.finos.org

JAX risk engine
github.com/AlexNeugroschl/JAX_Risk_Engine

Judah Diament · Yaakov Seif · Alex Neugroschl
diament@yu.edu · aseif@mail.yu.edu · aneugro1@mail.yu.edu

Yeshiva University · Yeshiva College Computer Science



Scan to give feedback

More Information

How TraderX Does SDD: The Spec is the Source, Code is Generated

State lineage

FINOS 001–014 | YU01 | YU02 | YU03 | YU04 | YU05 | YU06 | YU07 | YU08 | YU09 | YU10 | YU11 | YU12 | YU13 | YU14 | YU15 | YU16 | YU17 | YU18

Each state = its parent + a delta. **Core engine** · **Sell-side services** · **HA + order book** · **Multi-asset + risk**

Anatomy of one state's spec pack

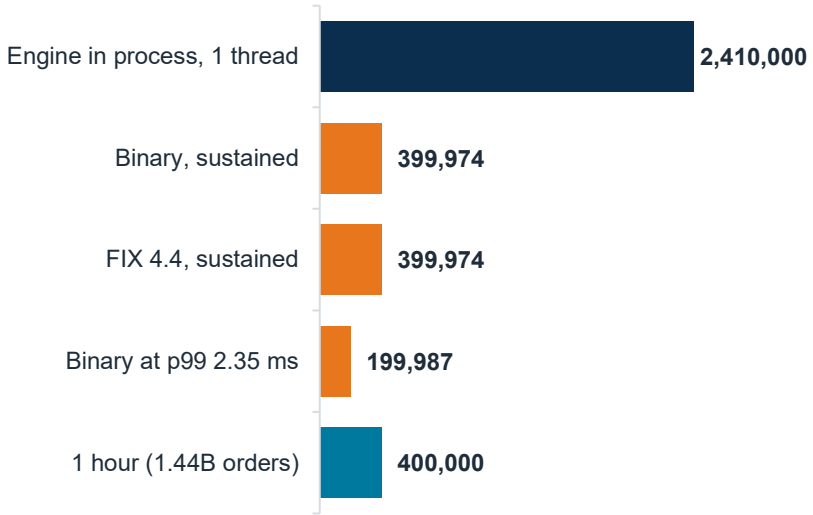
File	What it pins down
`spec.md`	User stories, functional and non-functional requirements, success criteria
`requirements/*-delta.md`	Exactly what changes versus the parent state
`plan.md · research.md`	Design, and the alternatives rejected (with reasons)
`data-model · contracts/`	Entities, wire formats, file formats
`system/adr-*.md`	Architecture decisions (70 across YU01–YU18)
`tasks.md`	Ordered, checkable work items
`runtime-overrides/`	This state's code layer (added or replaced files)

Generating and gating a state

1. **Generate the parent state** recursively, back to FINOS state 001
2. **Overlay this state's overrides** last writer wins, file by file
3. **Effective tree = the code that runs** composed, never hand-edited
4. **Gates** unit · zero-allocation · integration · GKE proofs
5. **State branch → images → GKE** e.g. YU13-limit-order-book

Performance: Every Number with its Workload, Hardware and Date

Throughput (orders/s acked, 3-member consensus)



Latency (coordinated-omission-safe)

Measure	p50	p99	Conditions
Match: limit cross	381–391 ns	440–631 ns	in process, 1 thread
Consensus commit	~220 µs	~340 µs	p50 flat, 100K–400K/s
Order → ack, 100K/s	1.69 ms	2.78 ms	p99.9 3.26 ms
Client send → ack	0.93–1.04 ms	2.0–2.2 ms	100K/s, no generator delay
Binary, 200K/s	1.34 ms	2.35 ms	9 gateways, 576 conns
FIX 4.4, 400K/s	1.63 ms	3.93 ms	QuickFIX/J, 18 gateways
1 h at 400K/s	1.50 ms	3.82 ms	1.44B orders, binary
Election, 100K/s	~0.63 s	-	0 acked orders lost

Rig: 3 × c4d-standard-8 members, 9–18 × c4d-standard-8 gateways (7 cores each), GKE us-east1-b, 22–26 Oct 2026. Engine rows: one c4d-standard-8, 2 dedicated cores.

How every number was captured. Per order, open loop, synthetic two-sided flow (one instrument, qty 1); 30 s window after 20 s warm-up. Every run passes: 0 missing, duplicate or rejected orders; the 3 members agree; trades = accepted orders; and strict replay of the leader's Aeron archive against client receipts (a negative control must fail, and does).

Integration result: cloud end-to-end run, 22 October 2026

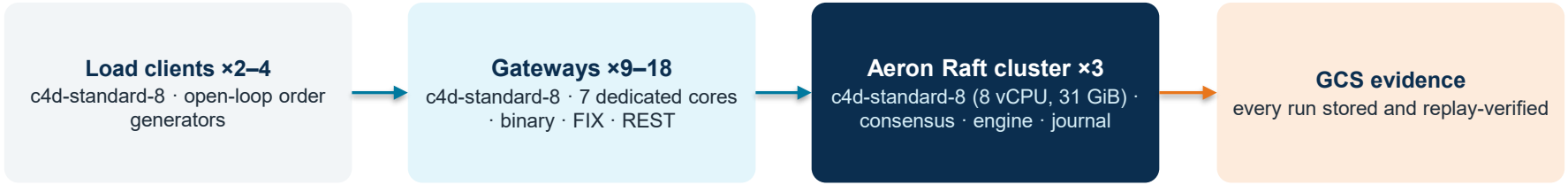
- ✓ **Fresh 3-member TraderX cluster on GKE** 8 Treasury bills traded; all 3 members agree
- ✓ **EOD cut at consensus sequence 75** 16 positions exported, byte-identical bundle
- ✓ **Account 22214's 8-bill portfolio mapped** account, run and export identity preserved
- ✓ **Priced as a Google Cloud Batch CPU job** Done on latest engine
- ✓ **Result verified, shown on the hosted Risk page** NPV $846,759.12

Demonstrated

- Full pipeline: trades → consensus cut → Cloud Batch pricing → verified result on the hosted Risk page
- Alex's engine build qualified and run
- Lineage checks refuse stale results

GCP Deployment: Instance Sizes and Tuning

Benchmark rig · GKE, us-east1-b · October 2026



Recommended configuration

all configuration, no code changes

- **Generational ZGC on members and FIX gateways:** FIX p99 3.8–5.8 ms vs 6.2–11.1 ms on G1
- **Smoothed Linux writeback:** no ~11 s flush bursts stalling all members
- **4 MiB UDP receive buffers:** the 128 KiB default dropped datagrams
- **7 dedicated cores per gateway:** p99.9 ~30% lower than 3 cores

2.78 ms

p99 at 100K orders/s

down from 6.1–16.6 ms before tuning; p99.9 3.26 ms

Risk Engine: Details & Work in Progress

Open Source Risk Engine (ORE)

What ORE provides

Implementation: Open-source C++ platform built on QuantLib, with Python bindings.

Modeling: Calibrates financial models and simulates correlated, multi-currency markets.

Analytics: Computes portfolio prices, sensitivities, exposure/XVA and market-risk measures.

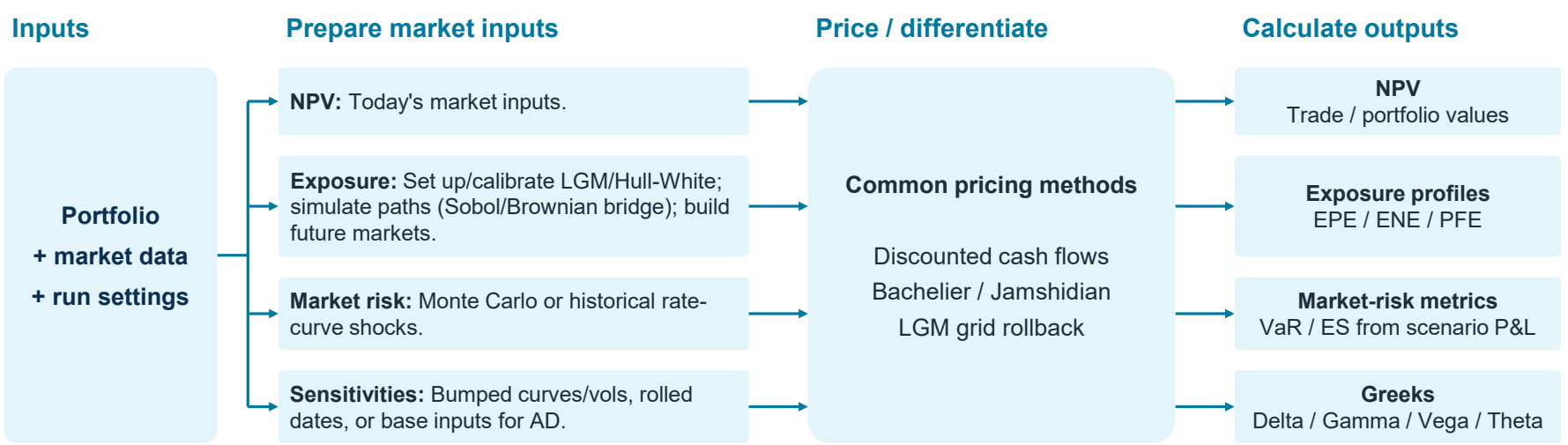
Porting from ORE

Use ORE's source code as a reference to implement its models and pricing algorithms in JAX.

Validate outputs directly against ORE on matching inputs.

Compare execution time against ORE on equivalent workloads.

Risk Engine: Data Flow and Validation



API: Accepts portfolio, market data and run settings for pricing, exposure, sensitivities, market risk and calibration.

Input preparation and validation: Parses dates, tenors and fixings, builds typed inputs and checks required data and compatible model, pricing and precision choices.

Execution: Queues and tracks jobs, checks worker hardware compatibility, and supports cancellation and retained outputs.

Testing: Validates models, pricing and risk against ORE; compares reduced precision with FP64; checks API validation and job execution.

Profiling: JAX/XProf traces Python calls and device kernels by engine phase, with cold and warm runs to separate compilation from repeated execution.

Precision Research

Adjustable precision: Configure numerical precision across risk engine stages.

Product and trade configuration: Set pricing precision separately for selected products or individual trades

Three precision settings:
Storage: format of saved values.
Compute: precision of matrix products.
Accumulation: precision of sums and other arithmetic.

Formats: FP64, FP32, FP16, BF16 and FP8 (E4M3FN, E5M2)

FP64 by default: Uses double precision as the reference baseline, with lower precision configurable for research.

Numerical safeguards: Uses scaling, reference-level deviations, and optional stochastic rounding to manage rounding error

Multi-pass: Optional shared-coefficient residual expansion (2–4 terms) and three-product BF16 multiplication retain more detail.

To-Do: Future Features

XVAs: Add CVA/DVA from exposure profiles, then extend to further valuation adjustments.

More instruments: Expand pricing coverage to FX and equity products.

More low-precision techniques: Explore additional quantization and multi-pass methods, including FP4.

Basel III risk calculations: Add standardised market-risk capital, stressed and liquidity-adjusted Expected Shortfall, and regulatory backtesting.