Designing a Rust-Based Financial Engine Architecture & DevOps Deployment Framework

A Hardened CI/CD Framework for Systems-Level Finance

zeba.academy

زیبا

Zeba Academy

First Published by Zeba Academy and Zeba Books.

Publication Year: 2026

Document Series: Zeba Academy Blueprints -- Sovereign Systems Technical Directives

Purpose: This series of blueprint directives is authored to combat the "enshittification" and unnecessary bloat of modern software. Our goal is to reclaim sovereign control over our systems by bridging the gap between deep academic theory and high-stakes industrial implementation. We believe that software should be fast, permanent, and most importantly, understandable to the person who owns and uses it.

Principal Architect: Sufyan bin Uzayr, Google Cloud-Certified Professional DevOps Engineer.

Core Stack: Linux, Rust, Zig, C++, Flutter, and PHP.

Email: hello@zeba.academy

Designing a Rust-Based Financial Engine Architecture and DevOps Deployment Framework

A Hardened CI/CD Framework for Systems-Level Finance

Purpose

This document provides the architectural directive for the design and operation of the Rust-based financial computing engine that will be used within the Zeba Academy financial analytics infrastructure.

The directive replaces legacy C++ code components with memory-safe Rust code and requires deployment via Google Cloud Build and Terraform's fully automated infrastructure pipeline.

This directive follows the operational principles set forth by Site Reliability Engineering (SRE) by removing failure classes from unmanaged C++ code.

Architectural Mandate

All financial calculation services must adhere to the following architecture:

Language Layer

- Rust (using the stable toolchain).
- Cargo-managed dependencies
- LLVM optimized binaries

Infrastructure Layer

- Google Cloud Build enables CI/CD orchestration.
- Terraform for Infrastructure as Code (IoC)
- Containerized deployment artifacts.

Operational Model

- Immutable deployments
- Automated rollback
- Infrastructure reproducibility
- Compile-time safety enforcement

C++ is not permitted in financial computing modules due to uncontrolled memory access and unknown runtime conditions.

Memory Safety Enforcement: Borrow Checker and Lifetime Validation

Rust's memory safety features are enforced by the borrow checker, a collection of compile-time checks that ensure memory safety. It verifies the code before compilation. It ensures that references do not outlive the data they reference. This totally eliminates the memory corruption bug, which is widespread in low-level programming.

In C++, memory is typically accessed indirectly using raw pointers. Raw pointers, often known as T* pointers, have no memory limits. Let us explore the following popular development paradigm in vintage C++-based financial systems.

```
Price* price = get_price();
delete price;
process(price);   // Dangling pointer
```

This is not anything that the compiler prevents. After clearing the memory, the pointer remains available and may be used, potentially leading to undefined behavior or a segmentation fault.

Rust introduced two reference types:

- &T (Immutable Reference)
- &mut T (Mutable Reference)

These references are subject to borrowing constraints, which are examined during build time.

Example:

```
let price = Price::new();
let p_ref = &price;
process(p_ref);
```

The borrow checker will monitor the lifetime of the price variable to guarantee that the price reference does not outlive the variable it references. Any effort to build a reference that outlives the data it refers to results in a compiler error.

Most significantly, Rust enforces the following aliasing constraints:

- Multiple immutable references (&T) can coexist.
- Only one changeable reference (&mut T) can exist at any given moment.

This limitation is particularly crucial for preventing data races in concurrent systems.

Pricing calculations are performed in worker threads within the concurrent evaluation module of the pricing engine. Simultaneous read/write access to shared data in C++ is often accomplished by manually coordinating access with mutexes or atomic types. However, incorrect implementation of these strategies results in race situations.

Rust's type system ensures that this type of programming blunder cannot occur. If two threads attempt to acquire mutable references to common price data at the same time, the code will not even compile. Synchronization of common mutable state is managed manually using concurrency primitives such as Arc<Mutex<T>>.

This programming style ensures memory safety and concurrency correctness without relying on runtime errors.

Segmentation Fault Elimination

Segmentation faults occur when a program attempts to access memory locations beyond its valid range.

Rust prevents segmentation faults through:

- Bounds checked indexing
- Enforced lifetimes
- Controlled unsafe blocks

Example:

```
let prices = vec![100.0, 101.5, 102.2];
println!("{}", prices[1]);
```

Index safety ensures memory correctness.

When manual memory control is necessary, Rust encapsulates the operation within an unsafe block.

Zero-Cost Abstractions

Rust abstractions compile to machine code equivalent to hand-written low-level logic.

No runtime penalty is introduced.

Core abstractions used in the financial engine:

- traits
- iterators
- pattern matching
- algebraic data types

Example:

```
let total: f64 = trades.iter().map(|t| t.price).sum();
```

This is optimized by LLVM into a vectorized loop.

There is no allocation overhead, and no virtual dispatch is needed.

Performance is maintained at parity with optimized C++ code.

Concurrency Safety

Multithreaded computation is a common occurrence in financial engines.

C++ code is unsafe when it comes to thread interaction, unless guarded.

Rust code is safe when it comes to thread interaction, by design.

Key traits:

- Send - data is safe to move across threads
- Sync - references are safe to share across threads

Unsafe data sharing is rejected at compile time.

Example architecture components:

- Arc<T> - atomic shared ownership
- Mutex<T> - synchronized mutation
- asynchronous processing - tokio

Data race conditions cannot be compiled if they violate Rust's ownership model.

Binary Efficiency

The Rust financial engine compiles into a statically linked binary.

Characteristics:

- No interpreter is needed
- No garbage collector is needed
- low dependency footprint

- The memory layout is predictable

This produces deployment artifacts suitable for containerized execution environments.

Binary sizes range from 10 to 30 MB, depending on the feature set.

DevOps Implementation

All deployments must be carried out using SRE-aligned CI/CD pipelines on Google Cloud Build.

Manual deployments are not permitted.

Pipeline Stages:

1. Tasks include source retrieval
2. Dependency resolution
3. Compilation
4. Static analysis
5. Test execution
6. Artifact creation
7. Container packaging
8. Deployment triggers

Here's an example Cloud Build configuration:

steps:

```
- name: rust
  entrypoint: cargo
  args: ["build", "--release"]
```

```
- name: rust
  entrypoint: cargo
  args: ["test"]
- name: gcr.io/cloud-builders/docker
  args: ["build", "-t", "gcr.io/zeba/financial-engine",
"."]
```

Build environments are containerized to ensure deterministic compilation.

Infrastructure as Code

Infra provisioning shall be done only via Terraform.

Direct console configuration is not allowed.

Example service definition:

```
resource "google_cloud_run_service" "engine" {
  name     = "financial-engine"
  location = "us-central1"
  template {
    spec {
      containers {
        image = "gcr.io/zeba/financial-engine"
      }
    }
  }
}
```

Terraform supports :

- infrastructure repeatability

- configuration version control.
- Consistent surroundings for staging and production.

SRE Principles Applied

The operational model explicitly follows Google SRE practices.

Error budgets

use Service Level Objectives (SLOs) to establish reliability goals.

Example:

- Availability SLO: 99.9%.
- Latency SLO: p99 < 20 ms.

Error budgets indicate the allowable error threshold.

The deployment velocity must be within the error budget.

Toil Reduction

Manual operational work is classified as toil.

SRE policy: Automate repetitive operational duties.

Automated Measures:

- Automated CI/CD pipelines
- Infrastructure as Code
- Containerized deployment model.
- Automated rollback policy

Immutable Infrastructure

Production infrastructure is replaced rather than modified.

Main rule:

No configuration drift.

Replacement occurs when fresh containerized images are deployed.

Guarantees:

- Reproducibility
- Auditability
- Rollability

Observability: Distributed Tracing and SLO Enforcement

Observability in the Rust financial engine is achieved through OpenTelemetry instrumentation integrated into the service runtime. The aim is to generate high-fidelity telemetry signals such as metrics, traces, and logs that enable SRE systems to measure conformance to specified Service Level Objectives (SLOs).

Distributed Tracing with OpenTelemetry

Each incoming request is assigned a trace context that is propagated throughout the pipeline while the request is being processed. The Rust service uses the opentelemetry and tracing ecosystem crates to generate spans that represent different stages of execution:

- request ingestion
- market data lookup
- pricing
- risk validation
- response serialization

Example instrumentation pattern:

```
use tracing::{info_span, instrument};
#[instrument]
fn compute_price(order: Order) -> PriceResult {
    // pricing logic
}
```

The #[instrument] attribute generates a trace span with contextual information such as function arguments, execution time, and error states. The layered trace spans display the complete execution path of a request, allowing for precise latency breakdowns.

The trace data is exported using the OpenTelemetry Protocol (OTLP) data exporter and sent to Google Cloud Trace. This gives a complete picture of the request flow from the services.

Metrics Export to Google Cloud Monitoring

The Rust engine collects metrics using the Open Telemetry metrics pipeline. The major metrics obtained are:

- Request latency (p50, p95, and p99).
- Request throughput.
- Error Rate

- Resource utilization

The request latency is measured using histogram instruments, yielding precise percentiles. When a non-successful status code is returned or a computation fails, the error rate increases.

These measurements are delivered to Google Cloud Monitoring and assigned to predefined Service Level Indicators (SLIs).

SLO Enforcement

The SLO compliance is monitored based on two main metrics:

- Latency SLI: p99 request latency
- Availability SLI: successful request ratio

The Cloud Monitoring pipeline constantly monitors these metrics against the SLO thresholds (for example, p99 latency is less than 20 ms, and the successful request ratio is greater than or equal to 99.9%).

Security Posture

Rust has significantly reduced the number of exploitable vulnerabilities compared to C++.

Types of vulnerabilities that are eliminated:

- Buffer overflow attacks
- Memory corruption attacks
- Race condition attacks

Security advantages include:

- Memory safety guarantees
- Strict type system
- Explicit unsafe boundaries

The code in the unsafe section must be reviewed.

Operational Requirements

The financial engine services must comply with the following operational requirements:

1. Rust stable toolchain must be used
2. Unsafe blocks must be documented
3. Terraform must be used for defining the infrastructure
4. Cloud build must be used for managing the CI/CD pipelines
5. Observability endpoints must be exposed
6. Immutable artifacts must be used for the deployments

Engineering Governance: Architecture Modification SOP

All such architectural changes must adhere to a standard Standard Operating Procedure (SOP).

Architecture Modification Procedure

All architectural change proposals must start with the submission of a Technical Change Proposal (TCP) to the Architecture Review Board. The change proposal must include the following:

- Architectural diagrams of the subsystem
- Performance impact
- Security implications
- Service Level Objectives (SLOs)
- Rollback strategy

The change proposal must be reviewed jointly by the following:

- Systems Architecture Lead
- DevOps Engineering Lead
- Security Review Committee

No change can occur without the change proposal's approval.

Formal Exception Request for C++ Components

The introduction of C++ components into the system’s runtime path constitutes a high-risk security event. C++ code is vulnerable to memory corruption attacks such as buffer overflows, use-after-free, and race conditions, since these attacks are not protected at compile time as in Rust.

A Formal Exception Request (FER) must be followed for the introduction of C++ code:

1. documented justification that describes why the requirement cannot be met using Rust
2. threat modeling analysis
3. mandatory memory safety audit
4. defined containment boundaries using FFI isolation

Unanimous vote from Architecture Lead and Security Review Committee members required for Approval.

Unsafe Code Auditing

Mandatory security auditing of unsafe code blocks in Rust must be implemented. The Security Review Committee is responsible for auditing all unsafe code paths for memory safety, boundary checks, and concurrency guarantees before code is deployed.

Conclusion

The financial engine architecture implemented in Rust provides a deterministic, memory-safe, and operationally reproducible environment.

The directive eliminates legacy C++ failure modes and enforces compile-time correctness guarantees.

Together with the automated infrastructure provided using Google Cloud Build, Terraform, and SRE operational principles, the resulting platform emphasizes reliability, maintainability, and performance.

Zeba Academy is a specialized technical research and training initiative dedicated to the principles of Sovereign Systems Engineering. Founded by Sufyan bin Uzayr - an author and university instructor as well as Google Cloud-Certified DevOps Engineer - Zeba Academy serves as a bridge between deep academic theory and high-stakes industrial implementation.

We reject the "enshittification" of modern software. Our core mission is the promotion of Anti-Bloat Architecture through the mastery of:

- **Systems Languages:** Using Rust, Zig, and C++ to build high-performance foundations that prioritize memory safety and deterministic execution.
- **SRE & DevOps:** Professional-grade automation via Google Cloud, Terraform, and Immutable Infrastructure to eliminate manual "toil" and operational fragility.
- **High-Performance Interfaces:** Utilizing Flutter for cross-platform development to deliver near-native mobile experiences without the lag of standard web-based wrappers.
- **Lean Web Publishing:** Reclaiming WordPress and PHP by stripping away the "slop", using Redis object caching and Unix sockets to transform standard platforms into high-speed, GEO-stable engines for modern publishing.
- **Legacy Modernization:** Applying memory-safe paradigms and modern build systems to century-old computational problems and aging C codebases.

Zeba Academy doesn't just teach code; we architect reliability. By merging the analytical rigor of Historical Research with the precision of Google-Certified Cloud Engineering, we provide our "Operatives" with the directives necessary to build systems that are safe, fast, and permanent.

Website:- https://zeba.academy

زیبا

Zeba Academy

zeba.academy