Архитектура финансового движка на Rust и фреймворк DevOps-развертывания

Защищённый CI/CD-фреймворк для финансовых системного уровня

zeba.academy

زیبا

Zeba Academy

Впервые опубликовано: Zeba Academy и Zeba Books.

Год издания: 2026

Серия: Zeba Academy Blueprints -- Технические директивы суверенных систем (Sovereign Systems Technical Directives)

Цель серии: Этот цикл директив разработан для борьбы с «энштификацией» и избыточной перегруженностью современного ПО. Наша цель — вернуть суверенный контроль над нашими системами, сократив разрыв между глубокой академической теорией и критически важной промышленной реализацией. Мы убеждены, что программное обеспечение должно быть быстрым, долговечным и, прежде всего, понятным для его владельца и пользователя.

Главный архитектор: Суфян бин Узайр, сертифицированный Google Cloud Professional DevOps-инженер.

Основной стек: Linux, Rust, Zig, C++, Flutter и PHP.

Email: hello@zeba.academy

Архитектура финансового движка на Rust и фреймворк DevOps-развертывания

Защищённый CI/CD-фреймворк для финансовых системного уровня

Назначение директивы

Настоящий документ устанавливает архитектурный регламент по внедрению и эксплуатации финансового вычислительного движка на базе Rust в инфраструктуре аналитики Zeba Academy.

Данная директива предписывает замену устаревших компонентов на C++ реализацией на языке Rust с безопасным управлением памятью. Развертывание осуществляется исключительно через полностью автоматизированный конвейер с использованием Google Cloud Build и Terraform.

Директива закрепляет операционные принципы SRE (Site Reliability Engineering), устраняя классы ошибок времени выполнения, характерные для неуправляемых систем на C++, и гарантируя детерминизм развертывания, воспроизводимость сборок и надежность промышленного уровня (production-grade).

Архитектурный мандат

Все сервисы финансовых вычислений должны соответствовать следующей архитектуре:

Уровень языка (Language Layer)

- rust (стабильная ветка/stable toolchain)

- управление зависимостями через Cargo
- бинарные файлы с оптимизацией LLVM

Инфраструктурный уровень (Infrastructure Layer)

- Google Cloud Build для оркестрации CI/CD
- Terraform для управления инфраструктурой как кодом (IaC)
- Контейнеризация артефактов развертывания

Операционная модель

- неизменяемые развертывания (immutable deployments)
- автоматический откат (automated rollback)
- воспроизводимость инфраструктуры
- контроль безопасности на этапе компиляции

Разработка новых модулей финансовых вычислений на C++ запрещена ввиду неконтролируемого поведения памяти и неопределенных состояний среды выполнения.

Контроль безопасности памяти: Borrow Checker и валидация времен жизни (Lifetimes)

Гарантии безопасности памяти в Rust обеспечиваются механизмом Borrow Checker - системой анализа на этапе компиляции, которая проверяет права владения, ссылки и времена жизни до сборки программы. Borrow Checker гарантирует, что ссылка никогда не «переживет» данные, на которые она указывает. Это предотвращает появление «висячих» ссылок (*dangling references*) и исключает целый класс ошибок повреждения памяти, характерных для языков с ручным управлением.

В C++ доступ к памяти часто осуществляется через сырые указатели (*raw pointers*). Указатели типа T* предоставляют неограниченный доступ к памяти, но не несут встроенных гарантий валидности, алиасинга или времени жизни. Рассмотрим типичный паттерн в устаревших финансовых движках на C++:

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

Компилятор не блокирует подобный сценарий. После освобождения памяти указатель продолжает существовать, и его разыменование приведет к неопределенному поведению (*undefined behavior*) или ошибке сегментации (*segmentation fault*).

Rust заменяет эту модель двумя типами явных ссылок:

- **&T** - неизменяемая ссылка (*immutable reference*)
- **&mut T** - изменяемая ссылка (*mutable reference*)

Использование этих ссылок регулируется строгими правилами заимствования, которые проверяются еще на этапе компиляции.

Пример:

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

Borrow Checker отслеживает время жизни (*lifetime*) переменной price и гарантирует, что p_ref не может существовать дольше переменной-владельца. Любая попытка создать ссылку, которая живет дольше самих данных, приведет к ошибке компиляции.

Что еще важнее, Rust жестко контролирует алиасинг (создание нескольких ссылок на одну область памяти):

1. Разрешено любое количество одновременных неизменяемых ссылок (&T).
2. В конкретный момент времени может существовать только одна изменяемая ссылка (&mut T).

Это правило критически важно для предотвращения состояний гонки (*data races*) в многопоточных системах.

В модуле параллельных вычислений котировок расчеты распределяются между рабочими потоками (worker threads). В C++ одновременный доступ на чтение и запись к общим структурам часто требует ручной синхронизации с использованием мьютексов (mutex) или атомарных примитивов. Ошибки в применении этих механизмов неизбежно приводят к состояниям гонки (race conditions).

Rust исключает этот класс ошибок на уровне системы типов. Если два потока попытаются одновременно получить изменяемую ссылку (&mut T) на одну и ту же структуру данных, Borrow Checker заблокирует сборку кода еще на этапе компиляции. В результате разделяемое изменяемое состояние (shared mutable state) должно быть явно синхронизировано с помощью примитивов конкурентности, таких как Arc<Mutex<T>>.

Такой подход гарантирует, что безопасность памяти и отсутствие состояний гонки проверяются статически, а не выявляются постфактум через сбои в продакшене или отладку после инцидентов.

Устранение ошибок сегментации (Segmentation Faults)

Ошибки сегментации возникают, когда программное обеспечение обращается к памяти за пределами допустимых границ.

Rust исключает этот класс ошибок за счет:

- Проверки границ при индексации (Bounds checking)
- Принудительного контроля времени жизни (Enforced lifetimes)
- Изоляции небезопасных операций в блоках unsafe.

Пример:

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

Безопасность индексов гарантирует корректность работы с памятью. В тех случаях, когда требуется ручной контроль, Rust изолирует операции внутри блоков unsafe, делая риски явными и доступными для аудита.

Абстракции с нулевой стоимостью (Zero-Cost Abstractions)

Абстракции в Rust компилируются в машинный код, эквивалентный написанной вручную низкоуровневой логике. Использование высокоуровневых конструкций не вносит накладных расходов во время выполнения (*runtime penalties*).

Ключевые абстракции, используемые в финансовом движке:

- Трейты (Traits)

- Итераторы (Iterators)
- Сопоставление с паттернами (Pattern Matching)
- Алгебраические типы данных (ADT)

Пример:

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

LLVM оптимизирует этот код в векторизованный цикл. При этом отсутствуют накладные расходы на аллокацию и задержки, связанные с динамической диспетчеризацией (*virtual dispatch*). Сохраняется паритет производительности с оптимизированным кодом на C++.

Безопасность параллельных вычислений (Concurrency Safety)

Многопоточные вычисления являются стандартом для финансовых движков. В C++ допускается небезопасное взаимодействие потоков, если оно не защищено явными механизмами синхронизации. Rust обеспечивает потокобезопасность на уровне системы типов.

Ключевые трейты:

- Send - данные могут передаваться между потоками.
- Sync - ссылки на данные могут одновременно использоваться в нескольких потоках.

Небезопасное разделение данных блокируется на этапе компиляции.

Компоненты архитектуры (примеры):

- Arc<T> для атомарного совместного владения.
- Mutex<T> для синхронизированного изменения данных.

- Асинхронная обработка на базе tokio.

Состояния гонки (data races) не могут быть скомпилированы, если они нарушают правила владения Rust.

Эффективность исполняемых файлов

Финансовый движок на Rust компилируется в единый статически слинкованный бинарный файл.

Характеристики:

- Отсутствие интерпретатора среды выполнения.
- Отсутствие сборщика мусора (GC).
- Минимальный объем зависимостей.
- Предсказуемая компоновка данных в памяти.

Такой подход позволяет создавать артефакты развертывания, идеально подходящие для контейнеризированных сред исполнения. Типичный размер бинарного файла составляет от 10 до 30 МБ в зависимости от набора функций.

Реализация DevOps

Все развертывания должны осуществляться через CI/CD-конвейеры, соответствующие принципам SRE, с использованием Google Cloud Build. Ручное развертывание запрещено.

Этапы конвейера:

1. получение исходного кода
2. разрешение зависимостей
3. компиляция

4. статический анализ
5. выполнение тестов
6. создание артефактов
7. контейнеризация
8. триггер развертывания

Пример конфигурации Cloud Build:

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",
"."]
```

Среды сборки контейнеризированы для обеспечения детерминированной компиляции.

Инфраструктура как код (Infrastructure as Code)

Управление инфраструктурой должно осуществляться исключительно через Terraform. Прямая конфигурация через консоль запрещена.

Пример определения сервиса:

```
resource "google_cloud_run_service" "engine" {
  name     = "financial-engine"
  location = "us-central1"
```

```
  template {
    spec {
      containers {
        image = "gcr.io/zeba/financial-engine"
      }
    }
  }
}
```

Состояние Terraform (state) обеспечивает:

- воспроизводимость инфраструктуры.
- версионирование конфигурации.
- идентичность сред разработки (staging) и промышленной эксплуатации (production).

Применение принципов SRE

Операционная модель строго соответствует практикам Google SRE.

Бюджеты ошибок (Error Budgets)

Целевые показатели надежности определяются через задачи уровня обслуживания (SLO - Service Level Objectives).

Пример:

- SLO доступности: 99.9%
- SLO задержки (Latency): p99 < 20 мс

Бюджеты ошибок определяют пороги допустимых сбоев. Скорость развертывания новых функций не должна приводить к исчерпанию остатка бюджета ошибок.

Сокращение операционной рутины (Toil Reduction)

Ручная операционная работа классифицируется как рутина (toil). Политика SRE требует устранения повторяющихся операционных задач посредством автоматизации.

Внедренные меры:

- Автоматизированные конвейеры CI/CD.
- Инфраструктура как код (IaC).
- Развертывание на базе контейнеров.
- Политики автоматического отката (rollback).

Неизменяемая инфраструктура (Immutable Infrastructure)

Промышленные системы должны заменяться, а не модифицироваться.

Ключевое правило: Отсутствие дрейфа конфигурации.

Развертывание осуществляется путем замены запущенных экземпляров новыми образами контейнеров. Это гарантирует:

- воспроизводимость.
- возможность аудита.
- возможность отката.

Наблюдаемость (Observability): Распределенная трассировка и контроль SLO

Наблюдаемость внутри финансового движка на Rust реализована с использованием инструментария OpenTelemetry, интегрированного непосредственно в среду выполнения сервиса. Цель состоит в получении

высокоточных телеметрических сигналов - метрик, трасс и логов, которые позволяют системам SRE измерять соответствие установленным задачам уровня обслуживания (SLO).

Распределенная трассировка с OpenTelemetry

Каждому входящему запросу, поступающему в движок котировок, присваивается контекст трассировки, который передается по всему конвейеру обработки. Сервис на Rust использует библиотеки (*crates*) экосистем opentelemetry и tracing для создания спанов (spans), представляющих дискретные этапы выполнения:

- прием запроса
- поиск рыночных данных
- расчет цены
- валидация рисков
- сериализация ответа

Пример паттерна инструментирования:

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

Атрибут #[instrument] автоматически генерирует спан трассировки, содержащий контекстные метаданные, такие как аргументы функции, время выполнения и состояния ошибок. Вложенные спаны отображают полный путь выполнения запроса, обеспечивая точную декомпозицию задержек (latency).

Данные трассировки экспортируются по протоколу OTLP (OpenTelemetry Protocol) в коллектор, который перенаправляет их в Google Cloud Trace. Это позволяет выстроить сквозную визуализацию потока запросов между сервисами.

Экспорт метрик в Google Cloud Monitoring

Движок на Rust также передает структурированные метрики через конвейер OpenTelemetry. Ключевые показатели включают:

- Задержка запросов (p50, p95, p99)
- Пропускная способность (throughput)
- Частота ошибок (error rate)
- Утилизация ресурсов

Метрики задержки фиксируются с помощью гистограмм для сохранения точности перцентилей. Счетчики ошибок инкрементируются каждый раз, когда сервис возвращает неуспешный статус или сталкивается со сбоями в вычислениях.

Эти метрики экспортируются в **Google Cloud Monitoring**, где они сопоставляются с предустановленными индикаторами уровня обслуживания **(SLI - Service Level Indicators).**

Контроль соблюдения SLO

Соблюдение SLO оценивается на основе двух основных индикаторов:

- **SLI задержки (Latency SLI):** задержка запросов на уровне p99.
- **SLI доступности (Availability SLI):** коэффициент успешных запросов.

Cloud Monitoring непрерывно сопоставляет эти метрики с пороговыми значениями SLO (например, задержка p99 < 20 мс, доступность ⩾ 99,9%). Нарушения расходуют бюджет ошибок сервиса, что активирует автоматические оповещения и ограничивает скорость развертывания при угрозе целевым показателям надежности.

Данный конвейер телеметрии обеспечивает точный контроль надежности в реальном времени в соответствии с операционными стандартами управления SRE.

Состояние безопасности (Security Posture)

Rust существенно снижает количество эксплуатируемых уязвимостей по сравнению с C++.

Устраненные классы уязвимостей:

- переполнения буфера (buffer overflows)
- эксплуатация повреждения памяти (memory corruption)
- эксплуатация состояний гонки (race conditions)

Преимущества безопасности включают:

- гарантии безопасности памяти
- строгую систему типов
- явные границы использования unsafe

Код внутри блоков unsafe подлежит обязательному аудиту.

Операционные требования

Все сервисы финансового движка должны соответствовать следующим ограничениям:

1. Требуется стабильный тулчейн Rust
2. Блоки unsafe должны быть задокументированы
3. Terraform должен определять всю инфраструктуру
4. Cloud Build должен управлять CI/CD-конвейерами
5. Эндпоинты наблюдаемости должны быть открыты
6. Артефакты развертывания должны быть неизменяемыми

Инженерное управление: СОП по модификации архитектуры

Все изменения архитектуры, затрагивающие финансовый движок, должны соответствовать официальному стандартному операционному регламенту (СОП) для сохранения надежности системы, гарантий безопасности памяти и операционной целостности.

Процедура модификации архитектуры

Любое предложение по изменению системной архитектуры должно начинаться с подачи технического предложения по изменениям (TCP) в Совет по архитектуре. Предложение должно включать:

- архитектурные диаграммы модифицируемой подсистемы
- анализ влияния на производительность
- последствия для безопасности
- влияние на задачи уровня обслуживания (SLO)
- стратегию отката (rollback strategy)

Предложение рассматривается совместно:

- Ведущим архитектором систем

- Ведущим инженером DevOps
- Комитетом по обзору безопасности

Ни одна архитектурная модификация не может быть принята к реализации без официального одобрения.

Запрос на использование компонентов C++

Внедрение компонентов C++ в путь исполнения (runtime path) рассматривается как событие высокого риска для безопасности. В C++ отсутствуют гарантии Rust времени компиляции против повреждения памяти, что делает его уязвимым к таким угрозам, как переполнение буфера, использование памяти после освобождения (use-after-free) и состояние гонки (race-condition).

Любой запрос на внедрение кода C++ должен проходить через процедуру формального запроса на исключение (FER), которая требует:

1. документированное обоснование, объясняющее, почему rust не может удовлетворить данное требование.
2. анализ моделирования угроз.
3. обязательный аудит безопасности памяти.
4. явные границы изоляции с использованием механизмов FFI (Foreign Function Interface).

Одобрение требует единогласного разрешения от ведущего архитектора и комитета по анализу безопасности.

Аудит небезопасного кода (Unsafe Code Auditing)

Блоки unsafe в Rust подлежат обязательному аудиту безопасности. Комитет по анализу безопасности (Security Review Committee) несет ответственность за

проверку всех путей выполнения небезопасного кода для подтверждения корректности работы с памятью, безопасности границ и гарантий многопоточности до выдачи разрешения на развертывание.

Заключение

Архитектура финансового движка на Rust создает детерминированную, безопасную с точки зрения памяти и операционно воспроизводимую платформу. Данная директива устраняет устаревшие сценарии отказов C++ и внедряет гарантии корректности на этапе компиляции.

В сочетании с автоматизированной инфраструктурой на базе Google Cloud Build и Terraform, а также операционными принципами SRE, итоговая платформа ставит в приоритет надежность, удобство обслуживания и предсказуемую производительность.

Zeba Academy - это специализированная инициатива в области технических исследований и обучения, основанная на принципах суверенной системной инженерии. Проект, созданный Суфяном бин Узайром - автором, университетским преподавателем и сертифицированным Google Cloud DevOps-инженером, служит мостом между академической теорией и реализацией критически важных систем.

Мы отвергаем «энштификацию» современного ПО. Наша основная миссия - продвижение **архитектуры без балласта (Anti-Bloat Architecture)** через освоение следующих направлений:

- **Системные языки.** Разработка на Rust, Zig и C++ высокопроизводительных фундаментов с упором на безопасность памяти и детерминизм исполнения.
- **SRE и DevOps.** Автоматизация профессионального уровня на базе Google Cloud, Terraform и неизменяемой инфраструктуры (Immutable Infrastructure). Мы ликвидируем операционную хрупкость и ручной труд (toil).
- **Высокопроизводительные интерфейсы**. Проектирование на Flutter кроссплатформенных систем с нативным откликом. Без компромиссов и задержек, присущих стандартным веб-оболочкам.
- **Регенерация веб-издательства.** Возврат WordPress и PHP в строй через радикальную очистку от «шлака». Объектное кэширование в Redis и Unix-сокеты превращают стандартные платформы в скоростные геостабильные движки.
- **Модернизация Legacy-систем**. Перенос классических вычислительных задач и старых кодовых баз на C в современные парадигмы безопасной работы с памятью и актуальные системы сборки.

Zeba Academy не просто учит писать код - мы проектируем надежность. Объединяя аналитическую строгость исторических исследований с точностью сертифицированной облачной инженерии Google, мы вооружаем наших «оперативников» директивами, необходимыми для создания систем, которые будут безопасными, быстрыми и долговечными.

Вебсайт: - https://zeba.academy

زیبا

Zeba Academy

zeba.academy