Backend Matching Engine - Project Charter

Project Name: Backend Matching Engine

Programming Language: Rust

Document Version: 1.0

Date: February 13, 2026

Executive Summary

This project will develop a high-performance, deterministic backend matching engine in Rust designed to support multiple trading platforms through protocol-agnostic interfaces. The engine will support order lifecycle management, price-time priority matching, market data generation, and comprehensive testing infrastructure suitable for both demonstration and production environments.

Project Objectives

Primary Objectives

1. Build a production-grade matching engine with deterministic behavior for reliable testing
2. Support multiple external trading platforms through protocol abstraction
3. Implement comprehensive security and governance controls
4. Create synthetic market data generation for testing and demonstration
5. Establish clear user onboarding and certification workflows

Success Criteria

- Order processing with price-time priority matching
- Sub-millisecond latency performance targets
- Deterministic matching behavior for reproducible testing
- Support for FIX 4.4/5.0, REST, WebSocket, and gRPC protocols
- Complete audit trail for all orders and executions
- Successful certification of at least 3 test users through sandbox environment

Scope

In Scope

1. Order Handling

Order Types:

- Limit orders
- Market orders (simulated)
- Cancel requests
- Modify requests

Order States:

- New
- Partially Filled
- Filled
- Canceled
- Rejected

Time-in-Force:

- GTC (Good-Till-Cancel)
- IOC (Immediate-or-Cancel)
- FOK (Fill-or-Kill)

Order Identification:

- Internal Order ID generation
- Client Order ID mapping and tracking

2. Matching Logic

- Price-time priority algorithm
- Partial fill support
- Deterministic matching for testing repeatability
- Basic self-trade prevention
- Configurable matching parameters

3. Order Book Management

- Full depth order book with configurable maximum levels
- Bid/ask separation
- Snapshot delivery
- Incremental updates
- Real-time book state queries

4. Trade Generation

- Execution report generation

• Unique Trade ID assignment

• Match timestamps (logical clock acceptable for initial version)

• Trade confirmation delivery

5. Market State Management

Market states: Open, Halted, Closed

• Trading session controls

• State transition events

6. Synthetic Market Data Generator

Price movement models:

• Random walk

• Mean-reverting processes

Configurable parameters:

• Volatility

• Spread

• Volume

• Deterministic seed support for repeatable tests

• Support for multiple asset classes including crypto

7. Instrument Management

• Dynamic instrument addition/removal via Admin API

• Enable/disable trading per instrument

• Parameter updates (tick size, lot size, price bands)

• Instrument metadata management

8. Protocol Layer (Decoupled Architecture)

• FIX 4.4 / FIX 5.0 protocol support

• REST API

• WebSocket streaming

• gRPC interface

• Protocol abstraction layer separating core engine from connectivity

9. Security & Governance

Authentication & Authorization:

• API key authentication

• Certificate-based authentication

Role-based access control (RBAC):

• Trader role

• Admin role

• Market Operator role

Risk Controls:

• Maximum order size limits

• Maximum position limits

• Price collar enforcement

• Rate limiting / throttling

Audit & Compliance:

• Immutable event log

• Complete order-to-execution traceability

• Admin action logging

• Regulatory reporting foundations

Environment Isolation:

• Separate Demo and Production environments

• No shared credentials across environments

• Environment identification in UI/API responses

10. Testing Strategy

Unit Tests:

• Order book behavior

• Matching edge cases

• Price-time priority validation

Deterministic Simulation Tests:

• Seeded market data generation

• Known expected outcomes

• Execution replay and verification

Property-Based Testing:

• Random order stream generation

Invariant validation:

• No negative quantities

• No crossed book post-match

• Conservation of quantity

Integration Tests:

• FIX client to engine workflows

• Cancel/replace race conditions

• Multi-protocol interoperability

Performance Tests:

• Orders per second baseline

• Memory stability over time

• Latency distribution analysis

11. User Onboarding Flow

Phase 1: Platform Onboarding

• Organization creation

• API credential issuance

• Role assignment

• Instrument access definition

Phase 2: Certification / Sandbox

• Order lifecycle testing

• FIX tag / API contract validation

• Risk limit verification

• Integration testing requirements

Phase 3: Live Demo Environment

• Resettable accounts

• Pre-funded demo balances

• Clear migration path to production

Out of Scope (Initial Release)

• Post-trade settlement processing

• Margin calculation

• Complex derivatives pricing

• Multi-leg order strategies

• Advanced order types (Iceberg, PEG, etc.)

• Real-time analytics dashboard (API only)

• Mobile applications

Technical Architecture

Core Components

1. Matching Engine Core (Rust)

• High-performance order book implementation

• Lock-free data structures where possible

• MPSC channels for message passing

• Actor-based concurrency model

2. Protocol Adapters

• Separate adapters for each protocol (FIX, REST, WebSocket, gRPC)

• Protocol-agnostic message translation

• Connection management

• Session state handling

3. State Management

• Event-sourced architecture for order events

• Snapshot + incremental update delivery

• Persistent event log (append-only)

4. Market Data Generator

• Configurable price process simulation

• Order flow simulation

• Deterministic random number generation

5. Admin API

• Instrument configuration

• User management

• System parameter configuration

• Monitoring and metrics

6. Authentication & Authorization Service

• API key management

• JWT token generation and validation

• Role-permission mapping

• Audit logging

Data Models

Order Message

```
struct Order {
    order_id: OrderId,
    client_order_id: String,
    instrument_id: InstrumentId,
    side: Side,  // Buy/Sell
    order_type: OrderType,  // Limit/Market
    quantity: Decimal,
    price: Option<Decimal>,
    time_in_force: TimeInForce,
    timestamp: Timestamp,
    trader_id: TraderId,
}
```

Execution Report

```
    struct ExecutionReport {
        order_id: OrderId,
        exec_id: ExecutionId,
        exec_type: ExecType,  // New/PartialFill/Fill/Canceled/Rejected
        order_status: OrderStatus,
        filled_quantity: Decimal,
        remaining_quantity: Decimal,
        avg_price: Option<Decimal>,
        last_qty: Option<Decimal>,
        last_px: Option<Decimal>,
        timestamp: Timestamp,
    }
```

Trade

```
    struct Trade {
        trade_id: TradeId,
        instrument_id: InstrumentId,
        buy_order_id: OrderId,
        sell_order_id: OrderId,
        price: Decimal,
        quantity: Decimal,
        timestamp: Timestamp,
        aggressor_side: Side,
    }
```

System Workflows

Order Lifecycle Workflow

1. Client submits order via protocol adapter (FIX/REST/WS/gRPC)
2. Protocol adapter translates to internal order message
3. Authentication & authorization check
4. Risk controls validation (size limits, price collars, rate limits)
5. Order submitted to matching engine
6. Matching engine processes order:

If no match: Add to order book, send acknowledgment

If match: Generate trades, update order book, send execution reports

7. Execution reports sent to both parties via their respective protocols
8. Events logged to immutable event log
9. Order book updates published to market data subscribers

Instrument Addition Workflow

1. Admin authenticates via Admin API
2. Admin submits instrument configuration (symbol, tick size, lot size, price bands)
3. System validates configuration
4. Instrument added to active instruments
5. Market data generator initializes price process for instrument
6. Instrument available for trading
7. Event logged

User Onboarding Workflow

1. Organization registers via Admin portal
2. Admin creates user account and assigns role
3. API credentials generated
4. User accesses Sandbox environment
5. User runs certification tests:

• Submit limit order → verify acknowledgment

• Submit market order → verify execution

• Cancel order → verify cancellation

• Modify order → verify modification

• Test risk limit rejections

6. Certification completion verified by Admin
7. User granted access to Demo/Live environment
8. Pre-funded demo balance allocated

Security Architecture

Authentication Flow

1. Client presents API key or certificate
2. System validates credentials against credential store
3. JWT token generated with user claims (roles, permissions, trader_id)

4. Token used for subsequent requests
5. Token expiration and refresh handled

Authorization Model

Trader Role:

- Submit/cancel/modify orders
- View own orders and executions
- View market data

Admin Role:

- All Trader permissions
- Add/remove instruments
- Configure system parameters
- View all user activity

Market Operator Role:

- All Admin permissions
- Control market state (Open/Halted/Closed)
- Emergency halt capability
- View system-level metrics

Audit Trail

All events logged with:

- Timestamp (nanosecond precision)
- Actor ID (trader, admin, system)
- Action type
- Entity ID (order, instrument, user)
- Before/after state (for modifications)
- Result (success/failure)
- IP address / session ID

Risk Management

Technical Risks

Risk	Impact	Probability	Mitigation
Performance degradation under high load	High	Medium	Comprehensive performance testing, load testing early

Data corruption in event log	High	Low	Write-ahead logging, checksums, backup strategy
Protocol compatibility issues	Medium	Medium	Extensive integration testing with real clients
Security vulnerabilities	High	Medium	Security audit, penetration testing, code review
Rust learning curve for team	Medium	High	Training, pair programming, code review standards

Operational Risks

Risk	Impact	Probability	Mitigation
Incorrect matching logic	High	Low	Extensive unit tests, property-based testing, certification suite
Regulatory compliance gaps	High	Medium	Compliance review, audit trail completeness verification
User onboarding bottlenecks	Medium	Medium	Automated certification process, clear documentation
Production deployment issues	Medium	Low	Staged rollout, canary deployments, rollback procedures

Testing Strategy Details

Unit Testing

Order Book Operations:

• Add order at various price levels

• Cancel order

• Modify order price/quantity

• Verify book integrity after operations

Matching Logic:

• Price-time priority enforcement

• Partial fills

• Self-trade prevention

• Edge cases (empty book, single order, etc.)

Property-Based Testing

Generate random order streams and verify invariants:

Conservation of Quantity: Sum of filled quantity = sum of trade quantities

No Negative Quantities: All quantities >= 0

No Crossed Book: Best bid < best ask (or one side empty)

Price-Time Priority: Earlier orders at same price match first

Order State Consistency: State transitions follow valid paths

Integration Testing Scenarios

1. **FIX Client Full Lifecycle:**

• Connect → Logon → Submit Order → Receive Execution → Logout

2. **Cancel/Replace Race Conditions:**

• Submit order

• Immediately submit cancel

• Verify correct state handling

3. **Multi-Protocol Interoperability:**

• FIX client submits buy order

• REST client submits matching sell order

• Both receive execution reports via their protocols

4. **Risk Limit Enforcement:**

• Submit order exceeding size limit → Reject

• Submit order outside price collar → Reject

• Exceed rate limit → Throttle

Performance Testing

Baseline Metrics:

Target: 10,000 orders/sec sustained throughput

Target: p99 latency < 1ms for order acknowledgment

Target: p99 latency < 5ms for match execution

Load Testing:

• Gradually increase order rate from 1k to 20k orders/sec

• Monitor latency distribution

• Monitor memory usage

• Identify breaking point

Soak Testing:

• Run at 50% max throughput for 24 hours

• Verify no memory leaks

• Verify consistent latency distribution

Development Phases

Phase 1: Core Engine (Weeks 1-4)

• Order book implementation

• Basic matching logic (price-time priority)

• Order state management

• Unit tests

Deliverables:

• Working order book with add/cancel/modify

• Matching logic with unit test coverage > 90%

• Basic execution report generation

Phase 2: Protocol Layer (Weeks 5-7)

• Protocol abstraction design

• FIX 4.4 adapter implementation

• REST API implementation

• WebSocket streaming

• Integration tests

Deliverables:

• FIX connectivity tested with QuickFIX client

• REST API with OpenAPI specification

• WebSocket market data streaming

Phase 3: Security & Governance (Weeks 8-10)

• Authentication service

• Authorization / RBAC implementation

• Risk controls (limits, collars, throttling)

• Audit logging

• Admin API

Deliverables:

• Working authentication with API keys

• Role-based permissions enforced

• Complete audit trail

• Admin API for configuration

Phase 4: Market Data & Testing (Weeks 11-13)

• Synthetic market data generator

• Deterministic testing framework

• Property-based tests

• Performance testing harness

Deliverables:

• Market data generator with configurable models

• Deterministic test suite with seeded data

• Performance benchmarks documented

Phase 5: User Onboarding & Documentation (Weeks 14-16)

• Onboarding workflow implementation

• Certification test suite

• API documentation

• Deployment procedures

• Sandbox environment setup

Deliverables:

• Automated user onboarding

• Certification test suite

• Complete API documentation

• Sandbox environment deployed

Success Metrics

Functional Metrics

• [ ] All order types supported and tested

• [ ] Matching logic passes 100% of unit tests

• [ ] All protocol adapters functional

• [ ] Risk controls enforced correctly

• [ ] Audit trail captures all required events

Performance Metrics

[ ] Order throughput: 10,000+ orders/sec

[ ] Order acknowledgment latency: p99 < 1ms

[ ] Match execution latency: p99 < 5ms

• [ ] Memory usage stable over 24-hour soak test

• [ ] Zero data loss under normal operation

Quality Metrics

• [ ] Unit test coverage > 90%

• [ ] Integration test coverage for all critical paths

• [ ] Zero critical security vulnerabilities

• [ ] Documentation completeness score > 95%

• [ ] Code review approval for all commits

User Metrics

• [ ] 3+ test users successfully certified

• [ ] Average certification time < 2 hours

• [ ] User satisfaction score > 4/5

• [ ] API integration time < 1 day for FIX clients

Stakeholders

Project Sponsor

Role: Provides vision, funding, and strategic direction

Engagement: Weekly status updates, major milestone reviews

Development Team

Rust Engineers (2-3): Core engine development

Protocol Engineers (1-2): FIX, REST, WebSocket, gRPC adapters

QA Engineers (1): Testing strategy and execution

DevOps Engineer (1): Deployment, monitoring, infrastructure

External Stakeholders

Test Users: Provide feedback during certification phase

Compliance Advisor: Ensure regulatory requirements met

Security Auditor: Review security architecture and implementation

Communication Plan

Status Reporting

Daily Standups: 15-minute sync on progress, blockers

Weekly Status Reports: Progress against milestones, risks, issues

Bi-weekly Demos: Demonstrate completed functionality

Monthly Steering Committee: Strategic decisions, scope changes

Documentation

Technical Design Documents: Architecture, data models, protocols

API Documentation: OpenAPI specs, FIX data dictionary, usage examples

User Guides: Onboarding procedures, certification requirements

Operations Runbooks: Deployment, monitoring, incident response

Assumptions and Constraints

Assumptions

• Rust development expertise available or can be acquired

• Test users willing to participate in certification

• Deployment infrastructure (cloud or on-premise) available

• No real monetary transactions in demo environment

Constraints

• 16-week development timeline

• Budget for cloud infrastructure and testing tools

• Must support FIX 4.4 as minimum protocol requirement

• Must comply with basic financial services security standards

Approval

Role	Name	Signature	Date
Project Sponsor			
Technical Lead			
Security Lead			

Appendices

Appendix A: Glossary

Order Book: Data structure maintaining all active buy and sell orders

Price-Time Priority: Matching algorithm prioritizing better prices, then earlier timestamps

Self-Trade Prevention: Logic preventing orders from same trader matching with each other

FIX Protocol: Financial Information eXchange protocol, industry standard for trading

Time-in-Force: Order duration instruction (GTC, IOC, FOK)

Execution Report: Message confirming order state change or fill

Market Data: Real-time information about orders, trades, and book state

Appendix B: References

• FIX Protocol Specification 4.4 / 5.0

• Financial Services Security Standards

• Rust Async Programming Best Practices

• Property-Based Testing with Proptest

Document Control:

Version: 1.0

Last Updated: February 13, 2026

Next Review: End of Phase 1 (Week 4)