1. Executive Summary

This project implements a scalable, event-driven backend architecture designed to:

- Detect trades directly from blockchain activity
- Compute user-level metrics (PnL, volume, fees)
- Maintain real-time portfolio balances
- Serve real-time updates via WebSocket

The system is designed to be highly modular and environment-agnostic.

It can:

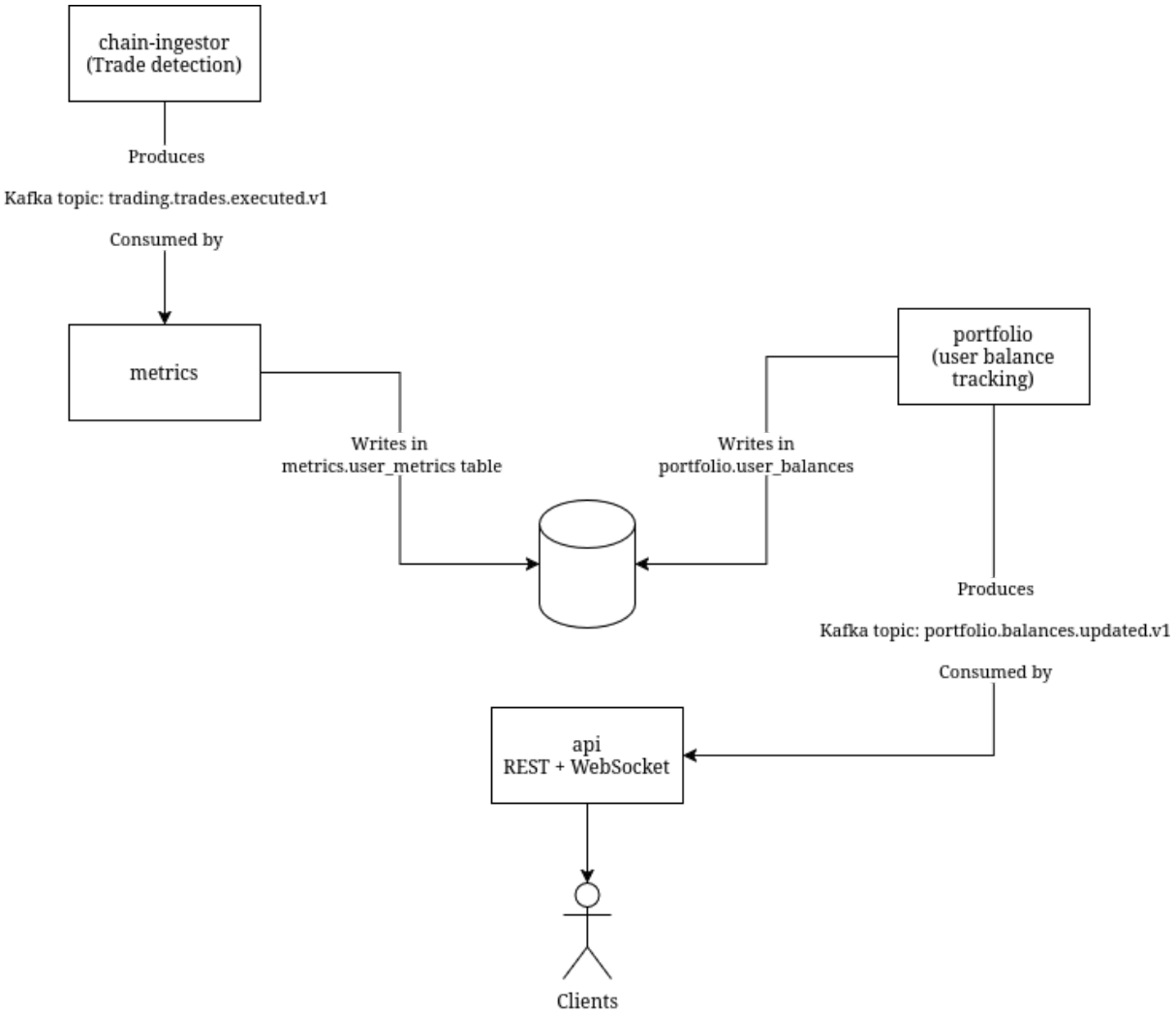
- Run on mainnet or testnet
- Operate fully in mocked mode for development
- Be deployed independently per environment
- Be scaled horizontally per service

This architecture ensures:

- Fault isolation
- High scalability
- Operational visibility
- Easy extensibility to new blockchains

2. System Architecture

The backend is composed of **four independent services**, communicating via Kafka.



Service Responsibilities

chain-ingestor

Responsible only for:

- Monitoring blockchain activity
- Detecting trades on the platform
- Publishing canonical trade events

It does not compute metrics.
It does not expose HTTP endpoints.
It is fully stateless.

metrics

Consumes trade events and:

- Computes user volume
- Computes earned fees
- Computes realized PnL
- Persists aggregated metrics

It does not expose write endpoints.

portfolio

Tracks user balances through an external provider (ex: Moralis).

Responsibilities:

- Upsert user balances
- Publish portfolio update events

api

Read-only gateway for clients:

- REST endpoints
- WebSocket real-time updates
- No write endpoints exposed

This prevents performance bottlenecks and isolates failures.

3. Database Architecture

Each service owns its own database schema:

- metrics.*
- portfolio.*

This ensures:

- No coupling between services
- No accidental data corruption
- Independent schema evolution
- Clear domain ownership

All schemas live inside a single PostgreSQL instance, but are logically separated.

4. Multi-Environment Support

The system supports multiple environments:

```
.env
.env.dev
.env.staging
.env.prod
```

Each service loads configuration independently:

```
ConfigModule.forRoot({
  isGlobal: true,
  envFilePath: [...]
})
```

This allows:

- Different Kafka clusters per env
- Different DB credentials
- Different RPC providers
- Different blockchain networks

5. Observability & Error Handling

Structured Logging

Each log entry includes:

- service
- event
- eventId

This enables distributed tracing across services.

Example:

```
{
  "service": "metrics",
  "eventId": "evt-123",
  "event": "metrics.updated"
}
```

Kafka Resilience

- Retry with exponential backoff
- Idempotent upsert operations
- Consumer group offset tracking

Health Endpoints

The API exposes:

GET /health

6. Scalability Strategy

The system is designed for horizontal scaling.

Stateless Services

- chain-ingestor: stateless
- metrics: stateless
- portfolio: stateless

Scaling = add more instances.

Kafka Parallelism

- Partition-based parallelism
- Multiple consumers per group
- Offset-based fault recovery

7. Security Considerations

- Only the API is exposed publicly
- No write endpoints exposed
- Events validated at every stage
- Clear separation of responsibilities
- Database schema isolation

8. Extending to a New Blockchain

The design allows seamless extension.

Example:

- Add chain-ingestor-solana
- Publish the same canonical TradeExecutedV1 event

No change required in:

- metrics
- portfolio
- api

Only inbound adapter changes.

This demonstrates:

- True decoupling
- Stable event contracts
- Blockchain-agnostic domain logic

9. Code Architecture – Hexagonal Design

Each service follows **Hexagonal Architecture (Ports & Adapters)**.

Core Principle

The business logic:

- Has zero framework dependency
- Depends only on interfaces (ports)
- Is fully testable in isolation

Advantages

Framework Independence

Business logic does not depend on NestJS or TypeORM.

Replaceable Providers

You can switch:

- Moralis → Alchemy

- Kafka → another broker
- Postgres → another DB

Without changing core logic.

Testability

Adapters can be mocked easily in unit tests.

10. Deployment Model

Each service can be deployed independently:

- Different scaling levels
- Different compute requirements
- Independent CI/CD pipelines