Portfolio Analytics System - Project Documentation

Home - Portfolio Analytics System

Welcome to the Portfolio Analytics System Wiki!

This system is designed to process financial transactions, calculate portfolio analytics such as positions, valuations, and performance, and expose them via REST APIs. It follows a modular, event-driven architecture using Kafka, PostgreSQL, and Python microservices.

Key Features:
- Modular microservices for ingestion, calculation, and exposure of portfolio data
- Event-driven architecture powered by Kafka
- Real-time and historical analytics: Positions, PnL, valuation, performance
- RESTful API layer for easy access to analytics
- PostgreSQL as the central analytics store; MongoDB optional for logs/snapshots
- Kubernetes-ready, with Dockerized services

Target Audience:
- Developers working on financial systems and analytics
- Data Engineers designing scalable data pipelines
- DevOps Teams deploying event-driven microservices
- Quantitative Analysts using analytics APIs for investment insight

System Architecture

The Portfolio Analytics System is a scalable, event-driven microservices platform designed to process financial transactions and deliver a suite of portfolio analytics through APIs.

System Inputs:
- Transactions: Buy/Sell orders, dividends, transfers
- Market Prices: Historical and intraday prices
- FX Rates: Foreign exchange rates for conversion
- Instrument Data: Metadata about securities (ISIN, asset type, etc.)
- Portfolio Metadata: Portfolio ID, open/close dates, base currency, risk horizon

Core Components:
- Ingestion Service (FastAPI, PostgreSQL, Kafka): Accepts incoming transaction data, validates it, persists it to PostgreSQL, and publishes to Kafka.

Portfolio Analytics System - Project Documentation

- Kafka + Zookeeper: Event bus for communication between calculators in the pipeline.
- Analytics Calculators (Python Kafka consumers): Stateless services that process events, persist analytics, and emit downstream events.
- PostgreSQL: Stores structured and calculated analytics data.
- MongoDB (Optional): For raw ingestion logs, market data snapshots, or audit trail.
- API Service (FastAPI): Serves calculated analytics data through REST endpoints.

Event-Driven Analytics Flow:
Each calculator subscribes to a Kafka topic, processes data, and emits the next event.
Example pipeline:
- transaction.received
- transaction.calculated
- position.calculated
- valuation.calculated
- performance.calculated

Parallel Calculations:
After certain stages (like position.calculated), parallel analytics services can run:
- allocation.calculated
- risk.calculated
- etc.

Kafka Topics (Example Naming):
- raw_transactions
- calculated_transactions
- calculated_positions
- calculated_valuations
- calculated_performance

REST API Endpoints:
- /portfolios/{id}/transactions
- /portfolios/{id}/positions
- /portfolios/{id}/valuation
- /portfolios/{id}/performance
- /portfolios/{id}/allocation
- /portfolios/{id}/risk

Portfolio Analytics System - Project Documentation

Ingestion Service

Overview:

The Ingestion Service is responsible for accepting raw transaction data via REST API, validating the input, persisting it into PostgreSQL, and publishing a Kafka event to trigger downstream analytics calculators.

Key Responsibilities:

- Provide a FastAPI endpoint /transactions to receive transaction data.
- Validate incoming transaction payloads.
- Persist transactions into PostgreSQL using SQLAlchemy and Alembic for schema migrations.
- Publish transaction.received events to the Kafka topic raw_transactions upon successful persistence.
- Use a shared Kafka producer utility for event publishing.

Technology Stack:

- FastAPI (API framework)
- PostgreSQL (relational database)
- SQLAlchemy (ORM)
- Alembic (DB migrations)
- confluent-kafka-python (Kafka producer)
- Kafka + Zookeeper (message broker)

Implementation Details:

API Endpoint:

- POST /transactions: Accepts transaction JSON, validates, writes to DB, and emits Kafka event.

Database:

- Transaction data stored in PostgreSQL transactions table.
- Alembic scripts manage schema versioning.

Kafka Integration:

- After storing the transaction, produce a transaction.received event.
- Event schema defined in a shared events.py file using Pydantic models.

Common Utilities:

- Kafka producer logic is centralized in common/kafka_utils.py for reuse.

Testing:

- Test ingestion with sample transaction payloads.
- Verify persistence in PostgreSQL.

- Confirm event emission to Kafka (using a console consumer or test consumer service).

Transaction Calculator

Overview:
The Transaction Calculator service consumes raw transaction events (transaction.received), processes them to calculate transaction-level analytics such as cost, realized PnL, and cash flows, persists the results, and emits a new event for downstream calculators.

Key Responsibilities:
- Consume transaction.received events from Kafka topic raw_transactions.
- Perform calculations: cost basis, realized profit and loss (PnL), cash flow impacts.
- Persist calculated transaction data in PostgreSQL (calculated_transactions table).
- Emit transaction.calculated events to Kafka topic calculated_transactions after processing.

Technology Stack:
- Python with FastAPI or lightweight consumer framework (Kafka consumer).
- confluent-kafka-python (Kafka consumer/producer).
- SQLAlchemy (ORM).
- PostgreSQL for persistence.

Implementation Details:
Kafka Consumer:
- Listens on raw_transactions topic.
- Processes each transaction message reliably with idempotency to avoid duplicate processing.
Calculation Logic:
- Uses historical FX rates and market prices as needed (initially mock data or simple lookup tables).
- Implements financial formulas for cost and realized PnL.
Persistence:
- Stores calculation results in the calculated_transactions table.
- Alembic migration scripts for table schema.
Event Emission:
- Publishes a transaction.calculated event upon successful persistence.
- Event schema defined in shared models.

Testing:

- Test with sample Kafka messages for various transaction types.
- Verify correct calculations stored in DB.
- Confirm event emission triggers downstream services.

Position Calculator

Overview:
The Position Calculator service aggregates calculated transactions into current portfolio positions, such as quantity held and book cost per instrument. It consumes transaction.calculated events, performs aggregation, persists results, and emits position.calculated events for downstream processing.

Key Responsibilities:
- Consume transaction.calculated events from Kafka topic calculated_transactions.
- Aggregate transactions into positions per portfolio and instrument.
- Maintain or query current position state (stored in PostgreSQL).
- Persist position data into the positions table.
- Publish position.calculated events to Kafka topic calculated_positions.

Technology Stack:
- Python Kafka consumer (confluent-kafka-python).
- SQLAlchemy ORM for database interaction.
- PostgreSQL for position storage.

Implementation Details:
Kafka Consumer:
- Listens on calculated_transactions topic.
- Processes messages with proper error handling and idempotency.
Position Aggregation Logic:
- Calculate total quantity and book cost per instrument per portfolio.
- May use a state store or direct DB queries/updates; initially stored in PostgreSQL.
Persistence:
- Store aggregated position data in the positions table.
- Alembic migration scripts manage schema changes.
Event Emission:
- Emit position.calculated events after successful persistence.
- Event schemas are shared and versioned.

Testing:
- Verify position calculations for various transaction scenarios.
- Confirm positions persist correctly in PostgreSQL.
- Validate event emission to trigger valuation calculators.

Valuation Calculator

Overview:
The Valuation Calculator service consumes position.calculated events, calculates the market value and unrealized profit and loss (PnL) of portfolio positions using market prices, persists the valuation results, and emits valuation.calculated events for further processing.

Key Responsibilities:
- Consume position.calculated events from Kafka topic calculated_positions.
- Retrieve current market prices (via service or mocked data).
- Calculate market value and unrealized PnL per position.
- Persist valuation results in PostgreSQL (valuations table).
- Emit valuation.calculated events to Kafka topic calculated_valuations.

Technology Stack:
- Python Kafka consumer (confluent-kafka-python).
- SQLAlchemy ORM for database persistence.
- PostgreSQL for valuation data storage.

Implementation Details:
Kafka Consumer:
- Listens to calculated_positions topic.
- Handles message consumption with idempotency and error recovery.
Calculation Logic:
- Uses market price data and FX rates as necessary.
- Calculates current market value and unrealized PnL per instrument position.
Persistence:
- Stores valuations in the valuations table.
- Schema managed with Alembic migrations.
Event Emission:

- Publishes valuation.calculated events upon successful persistence.
- Shares event schema definitions.

Testing:
- Validate valuations against sample positions and market prices.
- Confirm persistence of correct valuation metrics.
- Verify event emission for triggering performance calculations.

Performance Calculator

Overview:
The Performance Calculator service consumes valuation.calculated events, combines valuation data with cash flow information, calculates portfolio returns and contribution metrics, persists the results, and emits performance.calculated events.

Key Responsibilities:
- Consume valuation.calculated events from Kafka topic calculated_valuations.
- Access historical valuation and cash flow data.
- Calculate portfolio returns, total and by asset contribution.
- Persist performance analytics in PostgreSQL (performance table).
- Emit performance.calculated events to Kafka topic calculated_performance.

Technology Stack:
- Python Kafka consumer (confluent-kafka-python).
- SQLAlchemy ORM.
- PostgreSQL database for performance data.

Implementation Details:
Kafka Consumer:
- Listens on calculated_valuations topic.
- Processes messages with error handling and idempotency.
Calculation Logic:
- Combines valuation and cash flow data.
- Calculates returns over specified periods and contribution analysis.
Persistence:
- Stores results in performance table.

- Alembic migrations handle schema.

Event Emission:

- Emits performance.calculated events after persisting.
- Event schemas managed centrally.

Testing:

- Validate return calculations with sample data.
- Ensure performance data persists correctly.
- Verify event emission.

API Service

Overview:

The API Service exposes the portfolio analytics data via RESTful endpoints. It queries PostgreSQL (and optionally MongoDB) to serve calculated transaction, position, valuation, performance, allocation, and risk data.

Key Responsibilities:

- Provide FastAPI endpoints for portfolio analytics:
  - /portfolios/{id}/transactions
  - /portfolios/{id}/positions
  - /portfolios/{id}/valuation
  - /portfolios/{id}/performance
  - /portfolios/{id}/allocation
  - /portfolios/{id}/risk
- Implement pagination, filtering, and sorting for large datasets.
- Centralize business logic for data retrieval in service layer (e.g., portfolio_service.py).
- Handle error cases and provide meaningful HTTP responses.

Technology Stack:

- FastAPI for REST API framework.
- SQLAlchemy ORM for database interaction.
- PostgreSQL as the main data store.
- Optional MongoDB for snapshots or logs.
- Pydantic models for request/response validation.

Portfolio Analytics System - Project Documentation

Implementation Details:

Routers:

- analytics_router.py handles analytics-related endpoints.
- portfolio_router.py manages portfolio-specific routes.

Services:

- portfolio_service.py contains core logic querying PostgreSQL and MongoDB.
- Supports query parameters for filtering and pagination.

Testing:

- Unit tests for API endpoints.
- Integration tests with DB and mocked data.
- Performance/load testing for critical endpoints.

Infrastructure & DevOps

Overview:

This phase focuses on productionizing the Portfolio Analytics System with containerization, continuous integration/deployment (CI/CD), monitoring, and scalable deployments.

Key Responsibilities:

- Dockerize all microservices with optimized, production-ready Dockerfiles.
- Set up a CI/CD pipeline for automated builds, tests, and deployments.
- Implement centralized logging and monitoring.
- Deploy services on Kubernetes with proper manifests and Helm charts.
- Enable autoscaling based on Kafka event load.

Technology Stack & Tools:

- Docker: Containerize services using multi-stage builds.
- CI/CD: GitHub Actions, GitLab CI, or Jenkins pipelines.
- Logging: ELK stack (Elasticsearch, Logstash, Kibana) or Splunk.
- Monitoring: Prometheus and Grafana.
- Container Orchestration: Kubernetes (AKS, EKS, or GKE).
- Autoscaling: KEDA for Kafka event-driven scaling.
- Secrets Management: Vault or cloud provider secrets manager.
- Infrastructure as Code: Helm charts and Kubernetes manifests.

Implementation Details:

Portfolio Analytics System - Project Documentation

Dockerization:

- Each service has a Dockerfile optimized for minimal image size and faster builds.
- Use environment variables and secrets injection for configuration.

CI/CD Pipeline:

- Automatically build Docker images on commit.
- Run unit and integration tests.
- Push images to container registry.
- Deploy to Kubernetes clusters using Helm or kubectl.

Monitoring & Logging:

- Collect logs from all services to centralized store.
- Set up Prometheus exporters for service metrics.
- Configure Grafana dashboards for real-time monitoring.

Kubernetes Deployment:

- Define manifests for Deployments, Services, ConfigMaps, and Secrets.
- Use Helm charts for templated deployments.
- Configure Kafka consumer autoscaling with KEDA based on queue lag.

Testing & Validation:

- Load testing of entire analytics pipeline.
- Failover and recovery drills.
- Monitoring alert rules for SLA adherence.