Table of Contents

[Summary 2](#_Toc209038742)

[General Advice for All Projects: 2](#_Toc209038743)

[PROJECT LIST 3](#_Toc209038744)

[EASY Projects 3](#_Toc209038745)

[Market Data Simulator & Basic Analytics Tool (Python/Java) 3](#_Toc209038746)

[Order Book Implementation (Python/Java/C++) 3](#_Toc209038747)

[Simple Portfolio Tracker with SQL Backend (Python/Java + SQL) 3](#_Toc209038748)

[Configuration Management Tool (Python/Java) 4](#_Toc209038749)

[Basic Event-Driven Logger (Python/Java/C++) 4](#_Toc209038750)

[MEDIUM Projects 5](#_Toc209038751)

[High-Performance Order Book & Matching Engine (C++ / Java) 5](#_Toc209038752)

[Real-Time Data Stream Processor with Basic Alerting (Python/Java + Kafka/RabbitMQ) 5](#_Toc209038753)

[Algorithmic Trading Strategy Backtester (Python/Java) 5](#_Toc209038754)

[Distributed Task Queue with Worker Pool (Python/Java) 5](#_Toc209038755)

[RESTful API for Portfolio Management with Authentication (Python/Java + SQL) 6](#_Toc209038756)

[HARD Projects 7](#_Toc209038757)

[Low-Latency Market Data Feed Handler & Gateway (C++ / Java with messaging) 7](#_Toc209038758)

[Distributed Algorithmic Trading Platform (Python, Java/C++, Message Queue, SQL, Cloud) 7](#_Toc209038759)

[Real-Time P&L and Risk Aggregator (Python/Java + Distributed Cache + SQL/NoSQL) 8](#_Toc209038760)

[Containerized ETL Pipeline for Historical Data (Python/Java + AWS/Kubernetes + SQL) 8](#_Toc209038761)

[Cloud-Native Microservices for Configuration and Feature Toggles (Java/Python + AWS/Kubernetes + SQL/NoSQL) 8](#_Toc209038762)

[Lessons Learned 9](#_Toc209038763)

[1. Market Data Simulator & Basic Analytics Tool (Python/C++) 9](#_Toc209038764)

# General Advice for All Projects

* The key is not just *what* you build, but *how* you build it
  + demonstrating clean code
  + testing
  + performance considerations
  + an understanding of relevant concepts
* **Version Control**
  + Use Git diligently.
  + Commit frequently with clear messages.
* **Testing**
  + Write unit tests for all critical components.
  + For harder projects, consider integration tests.
* **Documentation**
  + Clear READMEs
  + architectural diagrams (even simple ones)
  + inline comments where necessary.
* **Clean Code:**
  + Follow best practices
  + design patterns
  + maintain readability.
* **Performance**
  + For medium/hard projects, discuss performance considerations, profiling techniques, and any optimizations made.
* **Containerization:**
  + For medium/hard projects, containerize your applications using Docker. This will make them much easier to deploy and demonstrate your understanding of modern deployment practices.
* **Think like an interviewer:** For each project, be prepared to discuss trade-offs, design decisions, challenges faced, and how you overcame them.

By building out a few of these projects, especially moving into the medium and hard categories, you'll have a very compelling story to tell about your skills and understanding of high-performance, distributed systems in a financial context. Good luck!

# PROJECT LIST

## EASY Projects

Focus: Fundamentals, Language Proficiency, Basic Concepts

These projects are about demonstrating solid foundational coding skills, understanding data structures, and basic application design. They should be well-tested and clearly documented.

#### Market Data Simulator & Basic Analytics Tool (Python/Java)

* + **Description:** Create a program that simulates a stream of stock or crypto prices (e.g., random walk within bounds, or reading from a static file). Store this data and then build a simple CLI or GUI to calculate basic analytics like moving averages (SMA, EMA), Bollinger Bands, or RSI over user-defined periods.
  + **Skill Showcase:** Python/Java proficiency, data structures (lists, deques), basic statistical calculations, understanding of financial data concepts, command-line parsing or simple GUI.
  + **Stretch Goal:** Implement the simulator as a separate thread/process producing data and another consumer process analyzing it (basic concurrency).

#### Order Book Implementation (Python/Java/C++)

* + **Description:** Implement a data structure for a Limit Order Book. It should support adding buy/sell limit orders, cancelling orders, and matching orders (e.g., a new incoming market order matching against resting limit orders). Keep it in-memory.
  + **Skill Showcase:** Language proficiency, efficient data structures (e.g., sorted dictionaries/maps for price levels, hash maps for order IDs), object-oriented design, understanding of core trading mechanics.
  + **Stretch Goal:** Add basic price-time priority logic for order matching.

#### Simple Portfolio Tracker with SQL Backend (Python/Java + SQL)

* + **Description:** Build an application that allows a user to input their stock trades (buy/sell symbol, quantity, price, date). Store these trades in a local SQL database (e.g., SQLite). The application should then be able to display the current holdings, average cost, and basic profit/loss for each position.
  + **Skill Showcase:** Python/Java proficiency, SQL database interaction (CRUD operations), basic data modeling, financial data representation.
  + **Stretch Goal:** Integrate with a free public API (e.g., Alpha Vantage, Yahoo Finance, carefully checking usage limits) to fetch current prices and calculate real-time portfolio value.

#### Configuration Management Tool (Python/Java)

* + **Description:** Create a small utility that can read configuration settings from different sources (e.g., JSON file, environment variables, command-line arguments) and apply them to a mock application. The tool should handle different environments (dev, prod) and potentially validate configurations against a schema.
  + **Skill Showcase:** Language proficiency, file I/O, parsing (JSON/YAML), error handling, understanding of application lifecycle/deployment.
  + **Stretch Goal:** Implement a mechanism for hot-reloading configurations without restarting the mock application.

#### Basic Event-Driven Logger (Python/Java/C++)

* + **Description:** Develop a simple logging framework that can log messages with different severity levels (INFO, WARNING, ERROR) to multiple destinations (console, file). Implement a basic event system where different "modules" can send log events to a central logger.
  + **Skill Showcase:** Language proficiency, object-oriented design (Observer pattern or similar), file I/O, basic error handling, understanding of utility libraries.
  + **Stretch Goal:** Make the logger thread-safe and add asynchronous logging to a file without blocking the calling thread.

## MEDIUM Projects

Focus: Performance, Concurrency, Distributed Concepts, API Integration, Intermediate System Design

These projects require a deeper understanding of language features, performance tuning, and the introduction of distributed system patterns.

#### High-Performance Order Book & Matching Engine (C++ / Java)

* + **Description:** Rebuild the "Order Book" project, but with a focus on performance and low-latency. Implement it in C++ or Java, ensuring minimal garbage collection impact (for Java) or efficient memory usage (for C++). Introduce multi-threading for processing incoming orders and market data concurrently. The matching logic should be highly optimized.
  + **Skill Showcase:** C++/Java performance tuning, concurrent data structures (lock-free or fine-grained locking), memory management, advanced algorithms, understanding of execution latency.
  + **Stretch Goal:** Introduce different order types (market orders, FOK, IOC) and simulate multiple concurrent market participants.

#### Real-Time Data Stream Processor with Basic Alerting (Python/Java + Kafka/RabbitMQ)

* + **Description:** Build a system where a data generator (e.g., the market data simulator from easy projects) publishes "ticks" to a message queue (e.g., Apache Kafka or RabbitMQ). A separate consumer application should process this stream, calculate a simple metric (e.g., 5-second moving average), and trigger an alert (e.g., print to console, send an email via a mock service) if the metric crosses a threshold.
  + **Skill Showcase:** Message queuing (producer/consumer patterns), real-time data processing, stream analytics, basic alerting mechanisms, Python/Java for both producer and consumer.
  + **Stretch Goal:** Store the processed metrics in a time-series database (e.g., InfluxDB) or SQL database for historical analysis.

#### Algorithmic Trading Strategy Backtester (Python/Java)

* + **Description:** Develop a framework to backtest simple trading strategies. The framework should read historical price data (CSV file), execute a user-defined strategy (e.g., buy when SMA(5) crosses above SMA(20), sell when it crosses below), and report performance metrics (total profit/loss, maximum drawdown, Sharpe ratio).
  + **Skill Showcase:** Python/Java numerical computing (NumPy/Pandas if Python), object-oriented design for strategies and backtest engine, data loading and manipulation, basic financial metrics calculation.
  + **Stretch Goal:** Allow for multiple concurrent strategies to be backtested, and optimize the backtesting engine for speed.

#### Distributed Task Queue with Worker Pool (Python/Java)

* + **Description:** Implement a simple distributed task queue. A "master" process or service adds tasks to a shared queue (e.g., Redis list, database table). Multiple "worker" processes pick up tasks from the queue, execute them (e.g., a mock computationally intensive task), and report results.
  + **Skill Showcase:** Distributed systems fundamentals, inter-process communication, concurrency, fault tolerance (what happens if a worker dies?), Python/Java for client/worker.
  + **Stretch Goal:** Implement basic retry logic for failed tasks and a monitoring interface to see task status.

#### RESTful API for Portfolio Management with Authentication (Python/Java + SQL)

* + **Description:** Build a secure RESTful API using a framework (e.g., Flask/Django for Python, Spring Boot for Java) that manages portfolios and trades. It should include user authentication (e.g., JWT), endpoints to add/view trades, create/view portfolios, and fetch basic portfolio summaries. Data should be persisted in a SQL database.
  + **Skill Showcase:** Web API design, security (authentication, authorization), SQL database integration, Python/Java web frameworks, HTTP protocols.
  + **Stretch Goal:** Implement rate limiting on API endpoints and add input validation for all requests.

# HARD Projects

Focus: Full-Stack Distributed Systems, Cloud Deployment, Low-Latency, Advanced Algorithms, Robustness

These projects combine multiple skill sets, require thoughtful system design for scalability and reliability, and ideally involve cloud deployment.

#### Low-Latency Market Data Feed Handler & Gateway (C++ / Java with messaging)

* + **Description:** Build a system that connects to a simulated (or real, if using a free tier API) market data source, processes raw data (e.g., FIX messages, binary feeds), normalizes it into an internal format, and then publishes it to a fan-out mechanism (e.g., multicast, high-performance messaging like ZeroMQ or Aeron) for multiple downstream consumers. Focus heavily on latency, throughput, and error handling.
  + **Skill Showcase:** C++/Java low-latency programming, network programming (sockets, potentially multicast), binary protocol parsing, high-performance messaging, robust error handling, concurrency.
  + **Stretch Goal:** Implement a conflation/snapshot mechanism to reduce data volume for slower consumers, and add a web-based monitoring dashboard.

#### Distributed Algorithmic Trading Platform (Python, Java/C++, Message Queue, SQL, Cloud)

* + **Description:** Design and implement a miniature, end-to-end algorithmic trading platform. This would involve:
    - A **Market Data Service** (from Hard Project 1 or Medium Project 2) feeding normalized data.
    - A **Strategy Engine** (Python/Java) that subscribes to market data, executes user-defined strategies (e.g., backtested ones), and generates trade signals.
    - An **Order Management System (OMS)** (Java/C++) that receives trade signals, manages order state (pending, filled, cancelled), and simulates sending/receiving execution reports.
    - A **Risk Management Component** (Python/Java) that monitors portfolio exposure and applies pre-trade/post-trade checks.
    - All components communicate via a message queue (Kafka/RabbitMQ). Persistence for orders, trades, and portfolio state in a SQL database.
  + **Skill Showcase:** Complex distributed system design, inter-service communication, polyglot programming, state management, fault tolerance, data consistency.
  + **Stretch Goal:** Deploy this entire system on AWS using Kubernetes, demonstrating CI/CD pipelines, logging (e.g., ELK stack), and monitoring (e.g., Prometheus/Grafana).

#### Real-Time P&L and Risk Aggregator (Python/Java + Distributed Cache + SQL/NoSQL)

* + **Description:** Build a service that aggregates positions and trades from multiple simulated sources (or your OMS from project 2), calculates real-time Profit & Loss (P&L), and key risk metrics (e.g., Delta, Gamma if you can simulate options positions). Data should be stored in a performant way (e.g., in-memory data grid like Redis or Hazelcast for current state, SQL/NoSQL for historical). Provide a REST API for querying aggregated P&L and risk.
  + **Skill Showcase:** Distributed caching, real-time aggregation, complex financial calculations, API design, scalability, data consistency across distributed components.
  + **Stretch Goal:** Implement scenario analysis – allowing users to query P&L/risk under hypothetical market movements.

#### Containerized ETL Pipeline for Historical Data (Python/Java + AWS/Kubernetes + SQL)

* + **Description:** Create an automated ETL (Extract, Transform, Load) pipeline for processing large volumes of historical market data (e.g., daily stock prices, tick data).
    - **Extract:** Pull data from an S3 bucket or external API.
    - **Transform:** Clean, validate, and enrich the data (e.g., calculate daily returns, add volume-weighted average price).
    - **Load:** Store the processed data into a PostgreSQL or MySQL database, optimized for analytical queries.
  + **Skill Showcase:** AWS (S3, EKS/ECS, RDS), Kubernetes (for deploying ETL jobs), Python/Java for data processing, SQL database design and optimization, robust error handling, scheduling (e.g., Airflow-like concept, or simple cron jobs within k8s).
  + **Stretch Goal:** Implement data versioning and schema migration strategies.

#### Cloud-Native Microservices for Configuration and Feature Toggles (Java/Python + AWS/Kubernetes + SQL/NoSQL)

* + **Description:** Develop a set of microservices to manage application configurations and feature toggles in a distributed environment.
    - A **Config Service** (REST API) to store and retrieve application configurations.
    - A **Feature Toggle Service** (REST API) to enable/disable features dynamically, potentially based on user groups or percentages.
    - Clients (mock applications) would consume these services.
  + **Skill Showcase:** Microservices architecture, API Gateway patterns, service discovery (e.g., using Kubernetes), data persistence (SQL/NoSQL), security, resilience (circuit breakers, retries), full cloud deployment (AWS/Kubernetes).
  + **Stretch Goal:** Implement a UI for managing configurations and feature toggles, and add real-time updates to client applications using WebSockets or server-sent events.

# Lessons Learned

## [Market Data Simulator & Basic Analytics Tool (Python/C++)](#_Market_Data_Simulator)

### Project Brief

* **Description:** Create a program that simulates a stream of stock or crypto prices (e.g., random walk within bounds, or reading from a static file). Store this data and then build a simple CLI or GUI to calculate basic analytics like moving averages (SMA, EMA), Bollinger Bands, or RSI over user-defined periods.
* **Skill Showcase:** Python/Java proficiency, data structures (lists, deques), basic statistical calculations, understanding of financial data concepts, command-line parsing or simple GUI.
* **Stretch Goal:** Implement the simulator as a separate thread/process producing data and another consumer process analyzing it (basic concurrency).

### Lessons Learned