Here's a **full-stack design** for developing and deploying a high-frequency trading simulator with a matching engine in C++. This covers architecture, technologies, tools, and the deployment pipeline.

---

1. Project Architecture

Core Components:

1. **Order Matching Engine**:
    - Handles buy/sell order matching based on price and time priority.
    - Key data structures: Priority queues, hash tables, and linked lists for quick operations.
2. **Order Book**:
    - Stores unmatched buy/sell orders.
    - Optimized for retrieval and updates.
3. **Market Data Feed Simulator**:
    - Simulates real-time market updates and incoming orders.
4. **Trader Interface (Frontend)**:
    - GUI for traders to place and monitor orders.
5. **Backend Service**:
    - API to interact with the matching engine and fetch results.
6. **Analytics & Monitoring**:
    - Tracks performance metrics like latency, trade volume, and order execution speed.

---

2. Tech Stack

Frontend:

- **Framework**: React (or wxWidgets if sticking to C++).
- **Purpose**: Allows traders to input orders and view the order book, matched trades, and performance metrics.

Backend:

- **Language**: C++ for the matching engine.
- **Framework**:
    - **gRPC/REST API**: For communication between frontend and backend.
    - **Boost.Asio**: For handling networking and real-time data feeds.
- **Database**: SQLite or PostgreSQL for storing historical trades and unmatched orders (optional for basic simulation).

Deployment:

- **Platform**: Docker for containerization.
- **Cloud Service**: AWS, Azure, or Google Cloud for deployment.
- **Monitoring Tools**: Prometheus & Grafana for real-time performance monitoring.

---

3. Development Workflow

Step 1: Setting Up the Core Matching Engine

- Design the engine as a C++ library.
- Core algorithm:
  - Price-time priority matching.
  - Separate buy and sell books with priority queues (or balanced trees).
- Implement operations:
  - Add order, cancel order, and match orders.

Step 2: Adding Real-Time Market Data Feed Simulation

- Create a module to simulate market events.
- Feed randomized orders to test engine performance.

Step 3: Designing the Order Book

- Use data structures optimized for insertion, deletion, and quick lookups.
- Expose operations to view the order book and trade history.

Step 4: Backend Service API

- Use **gRPC** or **REST API** to expose matching engine functionalities.
- API Endpoints:
  - `/addOrder`: Add a new order.
  - `/cancelOrder`: Cancel an existing order.
  - `/getOrderBook`: Fetch the current order book.
  - `/getTradeHistory`: Fetch trade execution history.

Step 5: Frontend Development

- Build a lightweight UI for traders to:
  - Place buy/sell orders.
  - View the order book.
  - Monitor trade executions and performance metrics.
- If using C++: Use wxWidgets for a simple GUI.
- If using React: Integrate with the backend via REST API.

Step 6: Adding Analytics

- Track:
  - Execution latency: Measure the time between order placement and matching.

    - Trade volume.
    - Order book size.
- Use Prometheus to collect metrics and Grafana for dashboards.

---

4. Deployment Pipeline

1. **Local Development**:
    - Use a **CMake** or **Meson** build system to manage the C++ code.
    - Use Docker to containerize the application locally.
2. **Testing**:
    - Unit tests for matching engine logic using Google Test or Catch2.
    - Integration tests for API endpoints using Postman or a similar tool.
3. **CI/CD Pipeline**:
    - Use GitHub Actions or Jenkins to automate:
        - Building the project.
        - Running tests.
        - Building Docker images.
4. **Production Deployment**:
    - **Containerization**: Package the backend into a Docker container.
    - **Cloud Deployment**:
        - AWS ECS, Azure Kubernetes Service, or Google Kubernetes Engine for scalability.
        - Use AWS RDS or Cloud SQL if using a database.
    - **Frontend Hosting**: Deploy the frontend on a static site hosting service like Netlify or AWS S3.
5. **Monitoring**:
    - Set up Prometheus to track latency and performance metrics.
    - Use Grafana for visualizing key metrics.

---

5. Tools for Each Stage

Development:

- **Editor**: Visual Studio Code or CLion for C++.
- **Libraries**: STL, Boost, Google Test.

Networking:

- **gRPC or REST Framework**: gRPC (C++) or Boost.Beast (for REST).
- **Real-Time Updates**: WebSocket integration using Boost.Asio.

Database:

- **SQLite**: For lightweight local storage.
- **PostgreSQL**: For cloud storage and analysis.

Deployment:

- **Docker**: To containerize the app.
- **Kubernetes**: To orchestrate containers for scaling.
- **AWS/GCP/Azure**: To deploy and run the application.

Monitoring:

- **Prometheus**: For metrics collection.
- **Grafana**: For dashboard visualization.

---

6. Iterative Plan

1. **Phase 1: Core Matching Engine**
    - Focus entirely on C++ implementation of the engine.
    - Test with hardcoded orders and ensure correct execution.
2. **Phase 2: Real-Time Data Simulation**
    - Add market data simulation and test the engine under stress.
3. **Phase 3: Backend API**
    - Expose engine functionalities via REST or gRPC.
4. **Phase 4: Frontend Development**
    - Build an interactive UI.
5. **Phase 5: Deployment**
    - Containerize and deploy the system on a cloud platform.
    - Monitor with Prometheus and Grafana.

Resources for Reference

1. **Matching Engine Basics**:
    - Matching Engine Explained
2. **Boost.Asio**:
    - Boost.Asio Documentation
3. **REST/gRPC**:
    - gRPC C++ Tutorial
    - Boost.Beast for REST
4. **Frontend Basics**:
    - React Documentation
    - wxWidgets
5. **Testing**:
    - Google Test Framework

Start by implementing the matching engine and testing it locally. Once it's functional, expand to other components step-by-step. This design allows you to scale as you learn and progress.