Detailed Project Schedule and Division of Work

(Nov 3 – Dec 1)

The project will be carried out over four weeks and divided into three main workstreams led by each member:
Mingxuan (infrastructure and runtime), Avi (engine and safety), and Paul (strategy, testing, and visualization).

Week 1 (Nov 3–9): Core Runtime, Order Book, and Simulation Foundations

Goal: Build the minimal working version of the system — event flow, order submission, logging, and a simple simulation loop.

- **Mingxuan:**
  - Implement the core runtime, including the `EventBus`, event base classes, and the binary logger with serialization/deserialization logic.
  - Set up the GitHub repository, CMake build system, and CI pipeline.
  - Create `Logger.hpp` and `Logger.cpp` to write serialized events to binary files.

- **Avi:**
  - Develop the initial **Limit Order Book (LOB)** with basic `add`, `cancel`, and `match` operations.
  - Introduce compile-time type safety through strong typedefs (`Price`, `Quantity`, `Notional`) and integrate them into the `OrderBook`.
  - Write small internal tests to ensure matching logic behaves deterministically.

- **Paul:**
  - Set up the **Catch2 testing harness** and integrate it with CMake and GitHub Actions.
  - Write initial **unit tests** for `EventBus`, `OrderBook`, and type safety.
  - Create a simple **simulation driver** (`apps/live_sim.cpp`) that submits a few mock orders (buy/sell) through the `EventBus` to test the entire event flow.
  - Start writing the initial **README** with build and run instructions.

Deliverable:
A minimal working prototype that can submit and log orders through the full event pipeline (submit → match → log).

JSON

```
map-hft/
│
├─ include/map/
│   ├─ core/         ← Mingxuan
│   │   ├─ Event.hpp
│   │   ├─ EventBus.hpp
│   │   └─ Logger.hpp
│   │
│   ├─ book/         ← Avi
│   │   ├─ OrderBook.hpp
│   │   └─ Trade.hpp
│   │
│   ├─ types/        ← Avi
│   │   ├─ Price.hpp
│   │   ├─ Qty.hpp
│   │   └─ Notional.hpp
│   │
│   └─ replay/       ← Mingxuan (for deterministic replays)
│       └─ LogReader.hpp
│
├─ src/
│   ├─ core/         ← Mingxuan
│   │   ├─ EventBus.cpp
│   │   ├─ Logger.cpp
│   │   └─ Clock.cpp
│   │
│   ├─ book/         ← Avi
│   │   └─ OrderBook.cpp
│   │
│   ├─ replay/       ← Mingxuan
│   │   └─ LogReader.cpp
│   │
│   └─ apps/         ← Paul (for simulation & strategy demos)
│       ├─ live_sim.cpp
│       └─ strategy_demo.cpp
│
├─ test/             ← Paul
│   ├─ core/
│   │   └─ test_eventbus.cpp
│   ├─ book/
```

```
│   │   └─ test_orderbook.cpp
│   ├─ integration/
│   │   └─ test_end_to_end.cpp
│   └─ strategy/
│       └─ test_strategy_behavior.cpp
│
├─ docs/             ← Paul (writes user + dev docs)
│
├─ CMakeLists.txt    ← Mingxuan
├─ .github/workflows/ci.yml  ← Mingxuan + Paul
└─ README.md         ← Paul
```

Week 2 (Nov 10–16): Deterministic Replay, Type Safety, and Strategy Integration

Goal: Ensure deterministic execution, compile-time safety, and the ability to replay the same session exactly.

- **Mingxuan:**
    - Build the **deterministic replay engine** that can reload binary logs and reproduce identical order, trade, and PnL sequences.
    - Implement checksum-based verification for replay consistency.
    - Document the log format in `docs/log-format.md`.
- **Avi:**
    - Embed compile-time and runtime **risk checks** into the matching engine (e.g., exposure limits, max order size).
    - Add fail-safe handling for invalid orders and halting conditions when limits are exceeded.
    - Verify `OrderBook` maintains identical behavior under replay.
- **Paul:**
    - Develop **integration tests** for deterministic replay (record + replay must produce the same results).
    - Write **unit tests** for the new risk constraints and type-safe operators.
    - Implement a **lightweight trading strategy class** (`strategy/BasicStrategy.hpp` + `.cpp`) that submits periodic buy/sell orders for simulation.

- Write a small **CLI command** (`replay_test.cpp`) that runs a full replay and prints checksum results.

Deliverable:
The system can record a session, replay it deterministically, and pass replay verification tests using the same binary log.

Week 3 (Nov 17–23): Intent Engine, Strategy Enhancements, and Performance Optimization

Goal: Add adaptive trading behavior (intent engine) and begin performance profiling.

- **Mingxuan:**
  - Profile latency and memory usage.
  - Implement timing logs and `--benchmark` mode in `live_sim`.
  - Identify optimization targets in the EventBus or serialization.
- **Avi:**
  - Add a **queue-depth estimation module** in the OrderBook to estimate order-flow imbalance.
  - Integrate intent-based logic that adjusts aggressiveness (more or fewer orders) based on market depth.
  - Ensure stability of matching logic under high volume.
- **Paul:**
  - Expand the **strategy layer** to include configurable aggressiveness parameters (slow, medium, aggressive).
  - Build a **performance benchmarking suite** (`test/performance/`) to compare live vs. replay performance.
  - Create **visual performance charts** (latency per event, total trades/sec) using CSV logs + matplotlib (Python or C++ plotting lib).
  - Write internal doc: `docs/performance_notes.md`.

Deliverable:
A deterministic engine capable of adaptive trading behavior and measured performance across live and replay runs.

Week 4 (Nov 24–Dec 1): Testing, Visualization, and Final Presentation

Goal: Validate the final system, prepare visualization tools, and present a live demo with identical replay output.

- **Mingxuan:**
    - Finalize profiling and optimize memory allocation and logging.
    - Prepare reproducibility metrics (latency per event, replay checksum report).
    - Clean up and comment on all core runtime and replay code.
- **Avi:**
    - Conduct **stress tests** using randomized and extreme market scenarios.
    - Produce **risk and stability plots** (price convergence, number of rejected trades).
    - Document edge cases and fixes in `docs/risk_tests.md`.
- **Paul:**
    - Integrate all components into the final **demo executable** (`apps/demo_showcase.cpp`).
    - Create a short **live + replay demonstration script** that proves deterministic behavior.
    - Develop a **visual dashboard or CLI viewer** that prints trades, fills, and events clearly.
    - Write the **final report and README**, explaining how to build, run, and verify deterministic replay.
    - Coordinate slides or live demo visuals for the presentation.

Deliverable:
A finalized, presentation-ready `Map` engine demonstrating deterministic replay, type-safe risk control, and verified performance metrics.

Collaboration and Management

- **Version Control:**
  Managed through GitHub with feature branches per member.
  All commits go through pull requests, with weekly merges after review.

- **Kanban Board:**
  Shared GitHub Project with columns: *Backlog → Ready → In Progress → In Review → Done.*
  Each issue includes labels for components (`core`, `book`, `strategy`, `replay`, `test`, `infra`) and owner.

- **Weekly Integration Check:**
  Every Sunday evening — run full build and end-to-end test suite.
  Review open PRs, verify all modules compile cleanly, and update next-week milestones.