Project Proposal: Event-Driven Bitcoin Backtester

Objective: Build a Python-based, event-driven backtesting engine to simulate high-frequency trading strategies on 10 years of intraday Bitcoin data.

Goal: Demonstrate proficiency in Object-Oriented Programming (OOP), system design, and market microstructure handling for quantitative developer roles.

1. The Concept: "Event-Driven" vs. "Vectorized"

Unlike simple scripts that calculate returns on a static DataFrame (Vectorized), we will build an **Event-Driven** system. This simulates a live trading environment where the system processes data bar-by-bar (minute-by-minute) in a loop.

- **Why?** This prevents "look-ahead bias" and allows us to model execution latency and slippage, which is critical for minute-level data.
- **The Loop:** Data Handler $\rightarrow$ Queue $\rightarrow$ Strategy $\rightarrow$ Portfolio $\rightarrow$ Execution $\rightarrow$ Loop.

2. Role Delegation

To avoid merge conflicts and allow parallel work, we will split the codebase into **Infrastructure** and **Logic**.

👤 Person A: Infrastructure Engineer

Focus: Data Pipeline, Event Loop, and Execution Simulation.

- **Responsibility:** Ensure the system feeds data correctly without looking into the future and handles the mechanics of "filling" orders.
- **Key Challenge:** Implementing a generator that yields data one row at a time efficiently.

👤 Person B: Quant Researcher

Focus: Strategy Logic, Portfolio Management, and Math.

- **Responsibility:** Decide *when* to trade and track *how much* money we have.
- **Key Challenge:** accurate Mark-to-Market (MtM) accounting and calculating risk metrics (Sharpe/Drawdown) dynamically.

3. File Architecture & Tasks

We will create a specific set of files. Each person owns specific files.

📂 Shared Files (Do this together first)

- **events.py**
  - Define the "Objects" passed between modules.
  - MarketEvent: Contains timestamp, OHLCV data.
  - SignalEvent: Contains symbol, strength, direction (Long/Short).
  - OrderEvent: Contains symbol, quantity, order type (Market/Limit).

📂 Person A Files (Infrastructure)

- **data.py**
  - Class: CSVDataLoader
  - Task: Read the massive CSV. Implement a get_new_bar() method using Python yield to push the next minute's data to the event loop.
- **execution.py**
  - Class: SimulatedExecutionHandler
  - Task: Take an OrderEvent and return a FillEvent.
  - **Feature:** Add **Slippage Model**. If we buy at $50,000, execute at $50,005 to simulate market impact.

📂 Person B Files (Logic)

- **strategy.py**
  - Class: MovingAverageCrossStrategy (inherits from a base Strategy class).
  - Task: Keep a rolling list of the last $N$ prices. When the Short MA crosses the Long MA, generate a SignalEvent.
- **portfolio.py**
  - Class: Portfolio
  - Task: Update cash and holdings based on FillEvents.
  - **Feature:** update_signal() method that checks current holdings vs. new signal to decide if we need to close a position or flip it.

4. The Workflow

1. **Setup:** Initialize the GitHub Repo.
2. **Contract:** Agree on events.py definitions (variables and data types) immediately.
3. **Development:**
   - Person A builds the loop to print data bar-by-bar.
   - Person B writes the logic to calculate a moving average on incoming data.
4. **Integration (The main.py):**

```
# Pseudo-code for main.py
while data.continue_backtest:
    event = data.get_new_bar()
    if event.type == 'MARKET':
        strategy.calculate_signals(event)
        portfolio.update_timeindex(event)
    elif event.type == 'SIGNAL':
        portfolio.generate_order(event)
    elif event.type == 'ORDER':
        execution.execute_order(event)
```

5. Future Extensions (After MVP)

- **Visualization:** Use Matplotlib to plot the equity curve vs. Benchmark (Buy & Hold).
- **Risk Management:** Add a RiskManager class that rejects orders if leverage exceeds 2x.
- **Optimization:** Implement a parameter optimizer to find the best Moving Average window lengths.