Jutsu Lab: PRD

Document:	**Product Requirements Document (PRD)**
Project:	"Vibe" - Modular Backtesting Engine
Author:	Anil Goparaju , Padma Priya Garnepudi
Version:	1.0
Date:	10/30/2025

1. Introduction

1.1. The Problem

Building, testing, and validating trading strategies is a complex, time-consuming, and often rigid process. Existing tools are frequently "black boxes," making it difficult to test unique ideas, integrate custom indicators, or connect to specific data sources without a massive engineering effort.

1.2. The Vision

To create a lightweight, modular, and expandable Python-based backtesting engine. This engine will empower a solo developer to rapidly prototype, test, and iterate on trading strategies. It prioritizes flexibility and "pluggable" components over a one-size-fits-all solution.

1.3. Guiding Principles

- **Modularity over Monolith:** Every core component (data, strategy, risk, metrics) should be an independent module that can be swapped out.
- **Simplicity & Vibe:** The core engine should be simple, well-documented, and easy to "vibe code" with. Avoid over-engineering the core loop.
- **Expandability First:** The design must assume new data sources, new strategies, and new analysis tools (like Monte Carlo) will be added later.
- **Trustworthy & Transparent:** The calculations for PnL, drawdown, and metrics must be simple, clear, and auditable. No "black boxes."

2. System Architecture: The "Pluggable" Modules

The engine will be built around 5 core, independent modules that communicate through a central EventLoop.

1. **DataHandler (The "Source"):**
   - **Job:** To fetch market data from any source and feed it, bar by bar, to the EventLoop.
   - **Must-Have:** A standardized "data-point" format (e.g., a Python dictionary or a pandas row) that all other modules will receive.
2. **StrategyEngine (The "Brain"):**
   - **Job:** To hold and execute the trading logic.
   - **Must-Have:** A Strategy base class (an interface) that all new strategies must inherit. This is the key to modularity. (e.g., class NewStrategy(Strategy): def on_bar(self, data): ...)
3. **IndicatorLibrary (The "Toolbox"):**
   - **Job:** A collection of functions (e.g., calculate_sma, calculate_rsi) that the StrategyEngine can call.
   - **Must-Have:** Must be "stateless." Functions simply take in data (like a pandas Series) and return a result. This allows you to easily copy/paste code from TA-Lib, pandas-ta, or other places.
4. **PortfolioSimulator (The "Bookkeeper"):**
   - **Job:** To receive trade "Signals" (Buy/Sell) from the StrategyEngine, execute "Fills," and keep track of the portfolio (cash, positions, and PnL).
   - **Must-Have:** Handles all state management. The Strategy module is "dumb"; it just sends signals. The PortfolioSimulator is "smart"; it manages the actual state.
5. **PerformanceAnalyzer (The "Report Card"):**

    - **Job:** To take the final trade log and portfolio history from the PortfolioSimulator and generate all key performance metrics.
    - **Must-Have:** This module is run *after* the backtest is complete. It is completely separate from the EventLoop.

3. Feature Requirements (User Stories)

Module 1: DataHandler

- **As a user, I want to** connect to the Schwab API **so that I can** fetch historical OHLCV (Open, High, Low, Close, Volume) data for a given ticker and timeframe.
- **As a user, I want to** have a DataHandler base class **so that I can** later create a CSVDataHandler or BinanceDataHandler without changing the core engine.
- **As a user, I want to** have the DataHandler provide a simple get_next_bar() method for the EventLoop to call.

Module 2: StrategyEngine

- **As a user, I want to** define a Strategy interface (base class) that includes init() and on_bar(data) methods.
- **As a user, I want to** be able to write a new strategy in a separate Python file (e.g., sma_crossover.py) and have the engine load it.
- **As a user, my** Strategy module must be able to call self.buy(ticker, quantity) or self.sell(ticker, quantity), which sends a signal to the PortfolioSimulator.

Module 3: IndicatorLibrary

- **As a user, I want to** have a indicators/ folder where I can drop a Python file (e.g., my_indicators.py).
- **As a user, I want to** be able to import and use any function from this library in my Strategy (e.g., from indicators.my_indicators import calculate_rsi).
- **As a user, I want to** pre-populate this library by copy/pasting 2-3 common indicators (e.g., SMA, EMA, RSI) to prove the concept.

Module 4: PortfolioSimulator

- **As a user, I want to** initialize the portfolio with a starting cash amount (e.g., $100,000).
- **As a user, when** my Strategy calls buy(), **I want** the PortfolioSimulator to:
    1. Check if I have enough cash.
    2. If yes, debit the cash.
    3. Credit the position (e.g., self.positions['AAPL'] += 100).
    4. Log the trade.
- **As a user, I want to** track the current_value of my portfolio at the end of each bar

(Mark-to-Market).
- **As a user, I want to** be able to set a simple commission and slippage model (e.g., $0.01 per share).

Module 5: PerformanceAnalyzer

- **As a user, when** a backtest is complete, **I want** to receive a dictionary or JSON object containing the following "absolute necessary" metrics:
    - **Total Return (%)**
    - **Annualized Return (%)**
    - **Max Drawdown (%)**: The peak-to-trough decline.
    - **Sharpe Ratio**: (Annualized Return - RiskFreeRate) / Annualized Volatility
    - **Sortino Ratio**: (Measures downside volatility only)
    - **Calmar Ratio**: (Annualized Return / Max Drawdown)
    - **Total Trades**
    - **Win Rate (%)**
    - **Profit Factor**: (Gross Profits / Gross Losses)
- As a user, I want to see a timeseries of all the metrics above. For example, Drawdown% by time plot.

4. Phase 1: Minimum Viable Product (MVP)

The first "vibe code" session should focus on getting a single end-to-end run.

1. **Data:** DataHandler that *only* connects to Schwab API for 1 ticker (e.g., SPY) on 1 timeframe (e.g., 1-Day).
2. **Strategy:** A single, simple SmaCrossover.py strategy.
3. **Indicators:** A single indicators.py file with a calculate_sma() function.
4. **Portfolio:** A simple PortfolioSimulator that assumes 100% of capital on buys and sells (no complex position sizing).
5. **Metrics:** A simple PerformanceAnalyzer that *only* calculates Total Return, Max Drawdown, and Sharpe Ratio.
6. **Output:** All results are printed to the console.

5. Future Enhancements (The "Expandable" Roadmap)

This design allows for the following features to be "plugged in" later without a rewrite:

- **[Feature] Monte Carlo Simulation:** A new module that runs *after* the PerformanceAnalyzer. It will take the trade log and run permutations (e.g., random shuffling of trade order) to test the strategy's robustness.
- **[Data] Multiple Data Sources:** Create CSVDataHandler, YahooFinanceDataHandler, etc., that all conform to the DataHandler base class.
- **[Strategy] Parameter Optimization:** A wrapper around the EventLoop that can run the *same* backtest 100s of times, varying the inputs for a strategy (e.g., testing SMA(20, 50) vs. SMA(30, 60)).
- **[Portfolio] Advanced Risk/Sizing:** Create a RiskManager module that the PortfolioSimulator consults before executing a trade (e.g., "never risk more than 2% of capital on one trade").
- **[Reporting] Web Dashboard:** A simple Streamlit or Flask app that reads the JSON output from the PerformanceAnalyzer and displays it in a beautiful UI.
