Simplifying Backtesting with QuantForge

Shenbao Lu* Han Wu† Zihan Zhang‡ Shuming Zhao§

Fundamentals of Web Engineering HS25
Interactive Visualization & Intelligence Augmentation Lab (IVIA)
ETH Zürich

Figure 1: Our Logo

ABSTRACT

While online brokers have democratized access to financial markets, the ability to create and test quantitative strategies remains limited to those with significant mathematical and programming expertise. This report presents a web-based solution that removes this technical barrier by converting natural language inputs into executable trading strategies. We detail the implementation of a web application that uses a Natural Language Processing (NLP) pipeline to interpret user descriptions and an automated backtesting framework that evaluates strategy performance. Results demonstrate that our system allows non-programmers to successfully define, test, and iterate on complex algorithmic strategies, effectively making quantitative trading accessible to a broader audience.

Index Terms: Algorithmic trading, backtesting, large language models (LLMs), natural language processing (NLP), financial technology (FinTech).

1 MOTIVATION

The landscape of financial markets has undergone a fundamental shift over the last decade, characterized by the unprecedented democratization of retail investing. According to a recent report by the JPMorgan Chase Institute, retail investing flows increased by approximately 50% between 2023 and early 2025, reaching levels that rival the pandemic-era savings surge [1]. This is not merely a temporary spike but part of a long-term trend where the stock market is playing an increasingly central role in household wealth accumulation.

Critically, the demographics of this new investor class have broadened. The share of 25-year-olds with investment accounts rose six-fold from 6% in 2015 to 37% in 2024, and participation among lower-income individuals has grown faster than that of higher-income groups [1]. While this lowered barrier to entry allows broader market participation, it introduces significant risks for inexperienced participants. The report highlights that this new cohort often relies on mobile interfaces where investment tools "mingle with other apps that distract or entertain," potentially leading to "fad-chasing" behavior driven by social media rather than fundamental analysis [1].

This creates a critical disconnect in the current financial technology ecosystem. While modern brokers have democratized the execution of trades (making it easy to buy and sell), they have not democratized the validation of strategies. Professional quantitative traders mitigate risk through rigorous backtesting and algorithmic discipline—techniques that require advanced programming skills (e.g., Python, C++) and financial domain knowledge. The emerging retail demographic, described in the report as needing "financial education tailored to new entrants" [1], generally lacks access to these technical safeguards.

Consequently, there is an urgent need for systems that bridge the gap between natural language intent and quantitative rigor. By enabling layman users to describe trading strategies in plain English and automatically generating backtested results, we can shift retail behavior from emotional, "trend-chasing" speculation toward data-driven decision-making. This work addresses that need by leveraging Large Language Models (LLMs) to lower the technical barrier for algorithmic trading, ensuring that the democratization of access is matched by a democratization of sophistication.

2 PROJECT SCOPE AND GOALS

2.1 Project Scope

To ensure a feasible delivery within the project timeline, we established strict boundaries regarding the complexity of the financial modeling and the flexibility of the NLP engine.

Boundaries and Limitations:

- **Market Focus:** The platform is restricted to US equities and major cryptocurrencies accessible via public APIs (Yahoo Finance).
- **Timeframe:** Analysis is limited to daily intervals. High-frequency trading (HFT) and intraday data are excluded due to data costs and computational complexity.
- **Execution:** The system is a simulation environment. Live brokerage integration is out of scope to avoid regulatory and security liabilities.

*e-mail: shenlu@student.ethz.ch
†e-mail:hanwuh@student.ethz.ch
‡e-mail:zhangziha@student.ethz.ch
§e-mail:zhaoshu@student.ethz.ch

Inclusions:

- **Vectorized Backtesting:** Implementation of a high-speed engine using `vectorbt` to simulate strategies over long historical periods instantly.
- **Strategy Library:** A comprehensive set of standard strategies including Trend Following (SMA, EMA, Turtle), Mean Reversion (RSI, Bollinger Bands), and Momentum (MACD).
- **AI Integration:** A conversational interface powered by Google's Gemini API to configure strategies, explain concepts, and diagnose strategy performance.
- **Risk Management:** Tools for setting stop-losses, take-profits, and trailing stops, along with stress testing against historical market crashes (e.g., Dot-com bubble, 2008 Financial Crisis).

2.2 Project Goals

The primary goal of QuantForge is to bridge the gap between technical complexity and user accessibility in quantitative finance.

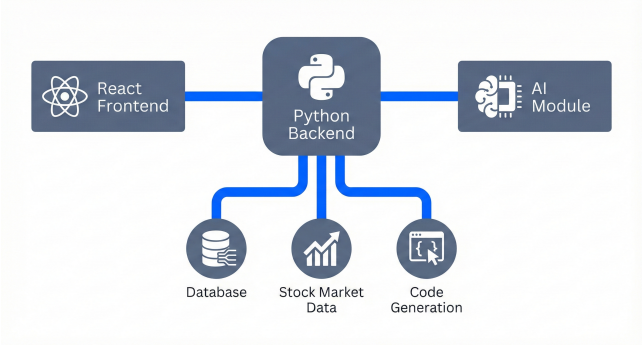
Specific Objectives:

1. **Democratize Strategy Creation:** Enable users to define complex trading logic using natural language (e.g., "Buy when RSI is below 30 and sell when it crosses 70"), removing the coding barrier. Additionally, move away from a conventional hi-tech design language in favor of "cuter" design for more approachability to non-tech-savvy users.
2. **High-Performance Simulation:** Achieve near-instant backtesting results for multi-year datasets to allow for rapid iteration.
3. **Educational Value:** Provide transparent insights into how strategies work by generating the underlying Python code and offering AI-driven explanations of performance metrics.
4. **Robust Risk Assessment:** Go beyond simple profitability metrics by incorporating stress testing and regime analysis to show how strategies perform under adverse conditions.

3 Approach

The solution is architected as a modern web application with a decoupled frontend and backend, leveraging yfinance for financial data and a Gemini API to provide NLP capabilities.

3.1 System Architecture



- **Frontend:** Built with **React 19** and **TypeScript**, utilizing **Vite** for the build toolchain. The UI is styled with **Tailwind CSS** for a responsive design. **Recharts** is used for rendering interactive financial charts.
- **Backend:** Developed in **Python 3.12** using **FastAPI**. Python was chosen for its dominance in the data science ecosystem.
- **Database:** **SQLite** is used for user data and strategy persistence, offering a lightweight and serverless solution suitable for this scale.

3.2 Core Backend Components

3.2.1 Vectorized Backtesting Engine

Instead of traditional event-driven backtesting (iterating through data row by row), QuantForge uses **VectorBT**. This library leverages NumPy and Pandas to perform vectorized operations, allowing the engine to calculate signals and portfolio stats for thousands of data points in milliseconds. The `SakuraEngine` class encapsulates this logic, handling data ingestion from `yfinance` and signal generation.

3.2.2 AI-Driven Configuration

The platform integrates **Google Gemini API**. A specialized system prompt defines the AI's persona as a "Quantitative Trading Mentor." The AI is instructed to output structured JSON data to map natural language requests (e.g., "Make it safer") to specific parameter updates (e.g., decreasing position size or adding a stop-loss). This ensures the deterministic execution of AI suggestions.

3.2.3 Code Generation

To foster learning, the backend includes a `code_generator.py` module. It translates the current strategy configuration into standalone Python scripts (compatible with VectorBT or Backtrader) that users can download and run locally.

3.3 Core Frontend Components

The frontend implements a modular dashboard architecture broadly divided into the input column on the left and the visualization column on the right. For the input components, we aim for familiarity and simplicity to reduce the cognitive load for the user.

The visualization components focus on presenting the results of the complex backtesting computation in a visually pleasing and interpretable manner. We implemented modules The visualization layer focuses on three distinct areas of user interaction:

- **Dynamic Strategy Charting:** Utilizing the **Recharts** library, the central panel renders high-performance time-series data. It features a dual-layer rendering system:
  1. **Price & Indicators:** The asset's close price is overlaid with dynamic technical indicators (e.g., Short and Long Simple Moving Averages) that update in real-time as users manipulate the configuration sliders.
  2. **Signal Annotation:** Buy (Green) and Sell (Red) signals are plotted directly onto the price action, allowing users to visually correlate trade execution with market movements. A vertical dashed line explicitly demarcates "In-Sample" (training) data from "Out-of-Sample" (validation) data to prevent overfitting.
- **Performance Telemetry & Badges:** The interface abstracts statistical output into digestible KPI cards. Beyond standard metrics (Total Return, Final Capital), the system calculates risk-adjusted metrics such as the **Sharpe Ratio** and **Max Drawdown**. These metrics are augmented with semantic "Health Badges" (e.g., `POOR`, `SAFE`) that provide immediate qualitative feedback on the strategy's viability based on pre-defined thresholds.

- **Granular Execution Log:** For auditability, the bottom panel renders a tabular view of every trade executed during the backtest. This includes exact timestamps, execution prices, and the specific signal logic (e.g., "Signal / Risk Trigger") responsible for the action, providing transparency into the engine's decision-making process.

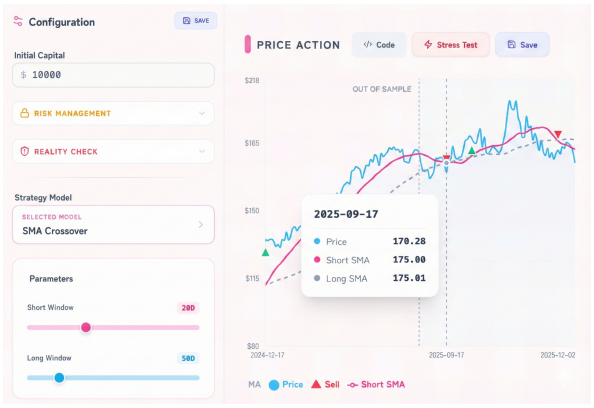
- **Configuration & AI Interaction:** The left sidebar hosts the **Configuration Control**, utilizing range sliders for continuous parameters (e.g., Lookback Windows) to facilitate rapid sensitivity analysis. Concurrently, the **AI Mentor** panel provides a conversational interface where users can input natural language goals (e.g., "Reduce drawdown"), which the system translates into parameter adjustments via the backend Gemini integration.

4 Results

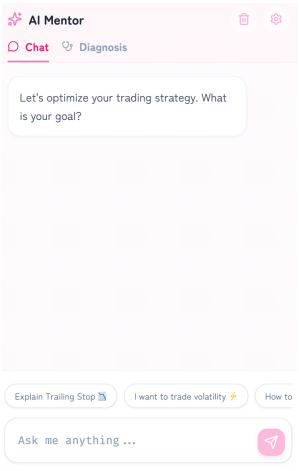
The project successfully delivered a functional and interactive trading platform.

4.1 Key Accomplishments

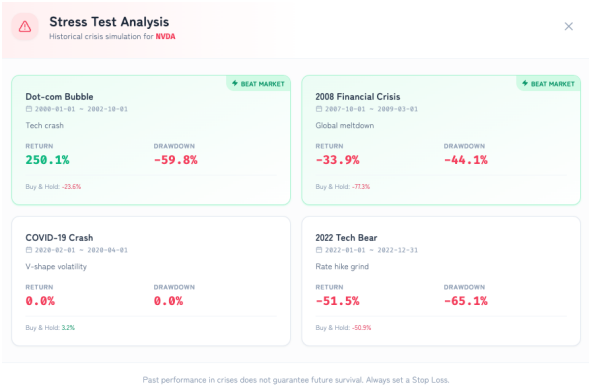
- **Interactive Interface:** Users can visualize price data with buy/sell markers, analyze equity curves, and toggle between different performance metrics (Sharpe Ratio, Max Drawdown, Win Rate).



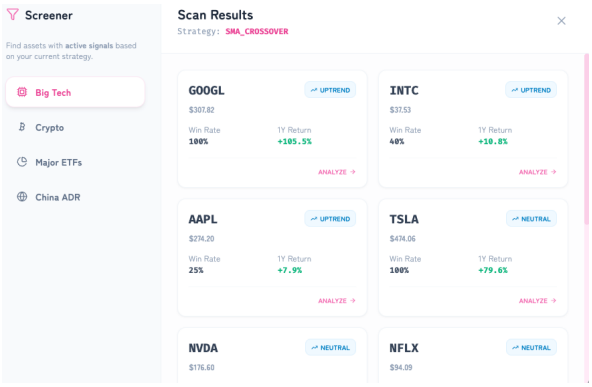
- **AI Assistant:** The Chat Interface effectively interprets user intent. For instance, asking "Use a Golden Cross strategy" automatically configures the SMA Crossover parameters (Short=50, Long=200) without manual input.



- **Stress Testing Module:** The application successfully simulates strategy performance during historical crash periods (e.g., COVID-19 crash), providing users with a realistic view of tail risks.



- **Market Screener:** A multi-threaded screener was implemented to scan sectors (Tech, Crypto, ETFs) for active signals, aiding in asset selection.



4.2 Performance

The vectorized approach proved highly effective. Backtesting a 10-year daily dataset for a standard strategy completes in under 200ms, enabling a fluid user experience where parameters can be tweaked in real-time.

5 Execution

The project execution followed an iterative development lifecycle.

5.1 Development Process

1. **Phase 1: Core Engine:** The initial focus was on building the `SakuraEngine` in Python to ensure accurate calculation of indicators and portfolio metrics.

2. **Phase 2: API & Frontend:** A FastAPI wrapper was created around the engine, and the React frontend was built to consume these endpoints.

3. **Phase 3: AI Integration:** The Gemini service was integrated. A significant challenge here was "hallucination" where the AI would suggest non-existent parameters. This was mitigated by enforcing a strict JSON schema in the system prompt.

4. **Phase 4: Advanced Features:** Stress testing and the market screener were added in the final stages to enhance the analytical depth.

5.2 Challenges

- **Data Consistency:** Handling missing data points or delisted tickers from Yahoo Finance required robust error handling and data cleaning steps in the backend. More concretely, we had to resolve the following issues:
  - Enforcing valid ranges by capping ‘endDate‘ to yesterday, and rejecting empty ranges with clear HTTP errors (404/400).
  - Implementing a 365-day warm-up buffer so indicators (MA/RSI/MACD) initialize correctly despite gaps, while metrics are computed strictly over the user's window to avoid distortion.
- **State Management:** Synchronizing the complex state of strategy parameters between the React frontend, the AI context, and the Python backend required careful design of the API payloads. We utilized the following measures to achieve this:
  - Enforcing a single source of truth in ‘App.txs‘ for data such as ticker, date range, strategy, params, fees, slippage, as well as child components like MarketBar, ConfigPanel, ChatInterface. This data is read and updated via props and callbacks.
  - Implementing stateless service functions (‘services/quantEngine.ts‘) which send strict DTOs that map to Pydantic models on the server. The TypeScript types (‘types.ts‘) mirror backend schemas to prevent drift. - Implementing the quant engine (‘SakuraEngine‘) to consume normalized OHLCV with a datetime index and executes signals at the next bar's open (with fees/slippage) to avoid look-ahead bias. The server converts boolean signals into per-bar markers before sending them to the UI, ensuring charts render consistently without extra client logic.

6 Next Steps

While QuantForge achieves its primary goals, several avenues for future development exist:

- **Live Trading Integration:** Bridging the gap between simulation and execution by integrating with brokerage APIs (e.g., Alpaca, Interactive Brokers). This would initially support ”paper trading” to validate strategies against real-time market microstructure—such as slippage and order fill rates—before enabling live capital deployment.
- **Portfolio Optimization:** Expanding the engine from single-ticker analysis to full portfolio management. Future updates will include Mean-Variance Optimization (MVO) and Risk Parity models, allowing users to construct diversified baskets, analyze correlation matrices, and simulate automated rebalancing schedules.
- **Community Ecosystem:** Developing a ”GitHub for Quants” social layer where users can publish, fork, and collaboratively improve strategies. This includes a leaderboard system based on risk-adjusted returns (Sharpe Ratio) rather than raw profit, fostering a meritocratic marketplace of ideas.
- **AI-Driven Hyperparameter Tuning:** Evolving the AI from a static configurator to an active researcher. By integrating optimization libraries (e.g., Optuna), the system could perform Bayesian optimization or genetic algorithms to mathematically discover optimal parameter sets, with the LLM interpreting the results to explain *why* a specific configuration minimizes drawdown.
- **Monetization:** Turning QuantForge from a project into a product requires a monetization strategy that respects our educational mission while acknowledging the reality of API and compute costs. We envision a hybrid 'Freemium' model where the core backtesting playground remains free—preserving the low barrier to entry for students—while power users subscribe for the heavy lifting, such as intraday data feeds, massive parameter optimizations, and deep-dive AI diagnostics. Beyond simple subscriptions, the long-term vision includes a 'Strategy Marketplace,' transforming the platform into a meritocratic economy where users can monetize their alpha by licensing verified strategies to the community. Coupled with potential affiliate pipelines for live brokerage execution, this approach aligns our revenue with user success, ensuring we profit when our users actually learn and improve, rather than just encouraging the kind of blind gambling we set out to prevent.

7 Conclusion

Building QuantForge wasn't just about slapping a React frontend onto a Python script; it was a crash course in the messy, exhilarating reality of modern financial engineering. We started with a simple question—can we make quantitative rigor accessible to the average retail trader?—and ended up wrestling with the complexities of vectorized backtesting, state management hell, and the unpredictable nature of LLMs. While the final product successfully democratizes strategy creation, letting users spin up complex algorithms with a single sentence, the journey revealed just how deep the rabbit hole goes. We realized that ”democratization” isn't just about UI; it's about latency, data integrity, and the terrifying speed at which a bad algorithm can lose money in a simulation. The integration of Gemini was a game-changer, turning a static tool into a dynamic mentor, but it also taught us that AI is only as good as the guardrails you build around it. Ultimately, QuantForge stands as a proof-of-concept that the gap between institutional power and retail access is closing, not because the math is getting easier, but because the tools to wield it are finally getting better. We didn't just build a backtester; we built a sandbox where the next generation of quants can fail safely, learn quickly, and maybe, just maybe, find an edge in the noise.

References

[1] C. Wheat, G. Eckerd, and B. Sullivan, “The Rise of the Retail Investor: Demographics and Behaviors in the Post-Pandemic Era,” *JPMorgan Chase Institute*, 2025. 1