ChatGPT

High-Performance Backtesting Engine & LLM-Driven Trading Stack

The Core Backtesting Engine (The "Muscle")

- **NautilusTrader** – *Rust core + Python bindings.* An open-source, **ultra-high-performance** algorithmic trading platform that supports event-driven backtests with **nanosecond** timestamp precision [1]. Its engine is written in Rust for speed, wrapped in Python for usability. **Why it fits:** NautilusTrader was built for exactly this purpose – **fast, accurate** simulation of multi-asset portfolios at tick level (it even advertises being fast enough for training RL agents) [1] [2]. **Pros:** Robust design with advanced order types and multi-venue support (ideal for realistic FX/Crypto backtesting) [3]; active maintenance and good docs. Python-side strategy code means you get C++/Rust speed with Python flexibility [4]. **Cons:** Steeper learning curve – it's a complex "production-grade" system. **Stitchability:** High. Nautilus is modular (adapters for data, brokers, etc.) and Python-native, so you can integrate its backtester as the engine within your custom pipeline with relative ease (feed it data, get callbacks on events, etc.).

- **HFTBacktest** – *Rust (with Python API).* A free open-source engine specifically for **high-frequency trading and market-making** strategies [5]. It accounts for **limit order book dynamics, queue position, and latency**, using full tick data (trades and Level2/Level3 order book) for realism. **Why it fits:** If you need to simulate a low-latency FX or crypto strategy that sits on the order book, this is purpose-built. **Pros:** Written in Rust for performance; models order execution delay and matching (crucial for HFT). MIT licensed with real-world exchange examples (Binance, Bybit) [5] [6]. Has a `py-hftbacktest` Python wrapper for ease of use [7]. **Cons:** Being niche, the community is smaller than general platforms. **Stitchability:** Moderate-High. You can use the Python API to plug this engine into your system – for instance, run backtests and then pass results to your analysis layer. It's a self-contained engine, so it can serve as the "muscle" that you call with prepared data.

- **Barter** (aka *barter-rs*) – *Pure Rust framework.* An open-source Rust library for building event-driven trading systems, supporting both live trading and backtesting [8]. **Why it fits:** Barter provides the foundational components (market data streams, strategy traits, portfolio/account, etc.) to assemble a custom engine. It recently added support for **tick-by-tick trading data** and a multi-threaded engine for running many pairs in parallel [9]. **Pros:** High-performance Rust implementation; very **modular** (each component can be extended or replaced). It's simpler than NautilusTrader – good if you want to "rip out" the core and customize. **Cons:** Lacks the polish of a full package (you'll be writing Rust code to use it; no ready-made Python UI). **Stitchability:** High for Rust users. Since it's essentially a collection of Rust crates (for data connectors, execution logic, etc.), you can include its components in your Rust project or even write Python bindings if needed. If you prefer not to code in Rust, this might not be the top choice (Nautilus or HFTBacktest would give you Rust speed with Python ease).

(Note: QuantConnect's Lean engine) – (C#) – is another battle-tested backtester (event-driven, stock/forex/etc.), though not in Rust/C++. It's open-source and very capable, but requires C# or using it via Python CLI. Lean is heavy to self-host, so if you want a lighter custom stack, the Rust options above are more attractive.*

The Valuation & Metrics Layer (The "Gold Standard" Analytics)

- **QuantStats** – *Python analytics library.* This is an excellent tool for after you have backtest results (e.g. a series of daily returns or a list of trades). QuantStats computes all the **classic performance metrics** – Sharpe, Sortino, Calmar, max drawdown, Win rate, Value-at-Risk, etc. – and can generate **beautiful reports/tear sheets** in one line of code [10]. **Why it fits:** It brings "quant rigor" to your stack, ensuring you can evaluate a strategy like a pro. **Pros:** Very easy to use – e.g. `qs.reports.full(ts, benchmark='SPY')` gives you an HTML report with tables and charts. It covers risk-adjusted metrics and even rolling stats and factor analysis. Under the hood it uses pandas, so it's compatible with any DataFrame/Series output. **Cons:** Being Python/Pandas-based, extremely large datasets (millions of trades) might need some downsampling for performance. Also, it's geared toward *portfolio-level* analysis (equity curve focus). **Stitchability:** Very high. Since your backtest engine (Rust) will likely output results to Python (Nautilus does this natively), you can feed those into QuantStats directly. For example, after a Nautilus run you take the Pandas `DataFrame` of P&L and pass it to QuantStats for analysis [10].

- **VectorBT** – *Python vectorized backtesting & analysis.* VectorBT (by Polakow) operates entirely on NumPy/Pandas, enabling you to **backtest entire strategy parameter grids in one go** via vectorization [11]. It's not a traditional event engine, but a high-speed analysis layer that can also compute metrics, drawdowns, and even produce charts. **Why it fits:** Use vectorbt in the research phase – e.g. scan through 1000 combinations of indicators on historical data in seconds – to identify promising strategies. Then you'd switch to the event-driven engine for fine-grained simulation. **Pros:** Blazing fast for compute-heavy tasks (leverages NumPy, Numba, and even GPUs in some cases). It has built-in technical indicators and can output performance stats for each run. One user reported running **1,000,000 backtest simulations in 20 seconds** with vectorbt [12]. **Cons:** Since it ignores market microstructure (everything is computed bar-by-bar or tick-by-tick without order book), it's not suitable for simulating slippage or latency. Memory can also be a limitation if you over-vectorize huge data. **Stitchability:** High, as a standalone tool. You might use vectorbt as a sub-module: run it on your data to get a quick idea of what works, then feed those parameters into your Rust engine. It's complementary – not integrated with the Rust backends but living alongside in your workflow.

- **Alphalens** – *Python factor analysis.* If your strategy development involves generating alpha "factors" (e.g. a signal time series that predicts returns), Alphalens (from the Quantopian ecosystem) is handy. It evaluates predictive power through IC (information coefficient), quantile returns, tearsheet of factor efficacy, etc. **Why it fits:** For **robustness**, you might want to ensure your strategy's signal truly has predictive value and isn't a random artifact. Alphalens helps answer that. **Pros:** Well-documented analysis of factor deciles, forward returns, turnover, etc., which is great for bias detection (e.g. is your strategy just picking up a sector bias or truly generating alpha?). **Cons:** It's somewhat dated and tied to Pandas data formats; and it's more useful for cross-sectional equity strategies (e.g. ranking stocks) than for single-instrument strategies. **Stitchability:** Medium. You'd use it separately from the backtest engine – e.g. compute your strategy's daily signal values and use

Alphalens to see if those signals had alpha before you ever placed a trade. It's an extra step for insight into the "why" of your strategy's performance.

- **Additional metrics & testing:** You might also incorporate libraries like **Empyrical** (from Pyfolio, for individual metric functions) or use **Monte Carlo simulations** and **Walk-Forward Analysis** on your results. For example, after a backtest, perform a Monte Carlo reshuffle of your trade returns to see result variability (some trading frameworks have this built-in; *Jesse* added Monte Carlo sim in recent updates). For Walk-Forward, you can either roll your own or use a tool like the open-source WFO Backtester (which uses expanding/rolling windows to optimize and test parameters). The key is to stress-test the strategy's performance – these aren't plug-and-play libraries in all cases, but rather techniques you'd apply in your analysis pipeline.

The LLM Integration / Agentic Workflow (The "Judge")

- **LLM Agent Trader** – *LLM-powered trading bot (Python).* This open-source project ties a trading strategy backtester to an LLM (like GPT-4) to analyze decisions [13]. It features a Next.js frontend and a FastAPI backend that runs a "streaming" backtest and consults an LLM for strategy logic. **Why it fits:** It's a practical example of using an LLM to **critique and guide** a strategy. For instance, the LLM Agent can generate strategy rules from natural language and even provide daily feedback on the strategy's performance [14]. **Pros:** Provides a template architecture (UI, API, engine, LLM integration) you can borrow from. The LLM is used for "smart strategy" generation and analysis; e.g., it hooks into Azure OpenAI or Google models to evaluate trade decisions in real-time [15]. This could be repurposed for having the LLM flag anomalies ("This week's trades deviated from historical pattern, maybe due to X news?") or suggest optimizations. **Cons:** It's a relatively young project (not a mature library), so you might use it more for inspiration than out-of-the-box. Also, relying on an LLM API introduces latency and cost, so it's not for every single tick – more for higher-level evaluation. **Stitchability:** Moderate. You could integrate a similar "LLM call" at the end of your backtest: e.g., auto-generate a summary of results and feed it to GPT-4 via API. The project's structure [13] can guide how to incorporate this without building everything from scratch.

- **Agentic Trading (FinAgents)** – *Multi-LLM-agent framework (Python).* This is a bleeding-edge research-level framework that reconceives a trading system as a **collection of LLM-driven agents** rather than fixed modules [16]. For example, a *Data Agent* prepares data, an *Alpha Agent* generates signals, an *Execution Agent* places trades, and importantly an *Audit/Evaluation Agent* monitors performance – all are LLMs collaborating. **Why it fits:** It's the "grand design" for LLM integration – going beyond a single chatbot to an autonomous, self-improving trading workflow. In this setup, an LLM-based Audit Agent could read the backtest results and cross-examine the Alpha Agent's logic, providing a bias check or suggesting a strategy tweak, much like a human team would [17]. **Pros:** Highly innovative – this is how a future quant fund **might use AI to avoid blind spots**. It emphasizes memory and context (a Memory Agent stores logs and insights for continuity). **Cons:** This is experimental and heavy – orchestrating multiple LLMs (and possibly vector databases for memory) is non-trivial and currently beyond typical individual use due to resource requirements. **Stitchability:** Low (for now). You likely wouldn't plug this whole system in directly; instead, you might adopt pieces of the philosophy. For example, you could implement a simple "Critique agent" that after each backtest run takes the trade log and uses an LLM to answer questions like "Where is this strategy most vulnerable?" – that's a scaled-down version of what Agentic Trading proposes.

- **FinGPT and Others** – The idea of an "LLM judge" for trading is very fresh, and there are a few other projects to watch or draw ideas from. **FinMem** [18] [19], for instance, is an academic project where an LLM trading agent is given a structured memory of past market events to enhance decision-making (to mimic how human traders learn from experience). There's also **AlphaSwarm** which is about agents debating trades. While these are not plug-and-play, they suggest workflows like: *after a backtest, have an LLM analyze the equity curve and trades in context of market data/news.* In practice, you can achieve a lot simply by leveraging LLM APIs with carefully designed prompts. For example, you might feed the LLM a summary: *"Strategy X had a Sharpe of 1.2 over 5 years, but all the gains came from two trades in 2021. Here is a list of yearly returns and major trades... Given this, how would you suggest improving the strategy or what regime does it seem to exploit?"* The LLM's response could highlight issues (perhaps it notes those two trades correspond to a known event) which you as the developer can then investigate.

Data Handling (The "Feed")

- **Polars DataFrame (with Parquet/Arrow)** – *Rust-powered DataFrame library.* Polars is an **extremely fast** DataFrame library available in Python, built on Apache Arrow's columnar format. It's designed for **efficient handling of large datasets** – exactly what you have with FX/Crypto ticks. **Why it fits:** Instead of using Pandas (which would chug on tens of millions of rows), Polars can comfortably group, resample, and slice huge tick data tables using multiple threads and SIMD optimizations [20]. It can read/write Apache Parquet files (which are great for storing compressed columnar data on disk) and memory-map them via Arrow for fast I/O. **Pros:** Polars is MIT-licensed, open-source, and has a pandas-like API. It often yields **order-of-magnitude speedups** and lower memory usage versus pandas (e.g. 5-10x faster is common, and it can handle >100 million rows in-memory on a laptop) [21] [22]. This means you can preprocess tick data (like building midprice series, or aggregating to candles for a quick scan) without waiting ages or crashing. **Cons:** Minor learning curve if you haven't used it, and some pandas features aren't 1:1 (but most common operations are supported). **Stitchability:** Very high. You can use Polars in your Python workflow before feeding data to the backtester – for example, use it to load a Parquet of EURUSD ticks, filter the date range, and downsample or convert to the engine's expected format (perhaps a Python list of events or a CSV). Polars will happily deliver a Pandas DataFrame or NumPy arrays if needed for interoperability.

- **ArcticDB** – *High-performance time-series DB by Man Group.* ArcticDB is the successor to the popular Arctic time-series store, re-engineered for speed and scalability. It's essentially a **database optimized for tick data** – allows **billions of rows** and fast query of slices of time-series [23]. **Why it fits:** If you accumulate a large amount of historical data (say multiple terabytes of crypto ticks), storing and retrieving that efficiently becomes a challenge. ArcticDB provides a solution that's free for use (it's open-source under Apache-2.0) and battle-tested in industry. **Pros:** It's designed for **Python integration** – you can read/write DataFrames directly. It's built on a modern C++ core for speed, with options to use local disk or cloud storage as the backend. Compared to flat files, it can index your data by symbol/time for lightning-fast access. In a community speed test, ArcticDB was found to be as fast or faster than plain Parquet/Feather files for time-series, while also offering versioning and snapshot capabilities [24] [25]. ArcticDB "gets data out of the way" and into your analysis environment quickly [26] [27] – exactly what you need for an agile research pipeline. **Cons:** It adds complexity – you have to run a database (though it can be embedded or even just use an `lmdb://` path locally). If your data volume is modest, Parquet files + Polars might suffice without a DB layer. **Stitchability:** Moderate. You could set up ArcticDB as your central data store (for example,

ingest your tick data feed into ArcticDB continuously). Your backtest pipeline then queries ArcticDB for the date range and instruments needed – returning a DataFrame that you pass to the engine. It works with Python, so it can sit naturally in your data-loading code. If you don't want that overhead, a simpler route is to use CSV/Parquet files and just load them via Polars, but ArcticDB will shine as your data grows or if you want features like *time-travel queries* (e.g., "give me the dataset as of 2021 for a simulation of what I knew then").

- **Other options:** There are other open-source data solutions depending on needs. **DuckDB**, for instance, is great for running SQL queries on Parquet files – you could have a DuckDB database that lets you do `SELECT * FROM ticks WHERE symbol='GBPUSD' AND date BETWEEN X and Y` without setting up a server (DuckDB runs in-process). It's not specialized for time-series like ArcticDB, but it's incredibly handy for ad-hoc slicing and joins, and it's also optimized for columnar data on disk. On the time-series database side, **QuestDB** (open-source, high-performance time-series DB in Java) is tailored for financial data with SQL and can ingest millions of ticks per second – but running a Java server might be overkill for a single-machine setup. **TimescaleDB** (an extension on PostgreSQL) is another option for time-series storage if you prefer SQL and relational integration. In summary, for a single powerful machine, a combination of **Parquet files + Polars** is often the simplest and quite effective (you get compression, speed, and no heavy DB server). As your needs grow, ArcticDB can be introduced to handle the **scale of tick data** while still integrating smoothly with Python/Pandas analysis code [23].

Proposed Modern Tech Stack Architecture

Bringing these pieces together, here's a blueprint for a **modern, modular backtesting stack** that meets your criteria:

1. **Data Ingestion & Storage:** Historical tick data is stored in an efficient format (e.g. Parquet files on disk, partitioned by instrument/date). Whenever you need data, **Polars** is used to load and pre-process it (e.g. filter to specific date ranges, convert to the format needed by the engine). If the dataset is huge or distributed, you'd leverage **ArcticDB** as a speedy interface – e.g., call ArcticDB to fetch a chunk of tick data into memory. This ensures that even with billions of rows, you can retrieve what you need in seconds [23]. *(Example: Use Polars to quickly resample tick data to 1-second bars for an initial scan, or pull 5 years of tick data for EUR/USD from ArcticDB for backtesting.)*

2. **Core Backtesting Engine:** The heavy lifting is done by the **Rust-based engine**. For instance, run **NautilusTrader** to execute an event-driven simulation of your strategy on the tick data feed (with all the proper modeling of order execution, spreads, latency). Nautilus consumes the data you prepared (it can handle streaming tick events with nanosecond timestamps) and produces a detailed log of fills, P&L, etc. Since NautilusTrader runs natively in Python, you can define strategies in Python but still get Rust performance [4] [1]. If you have multiple strategies or want to test portfolio interactions, Nautilus can backtest them **concurrently** on multiple instruments. *(Meanwhile, for strategy ideation, you might separately use vectorbt in a Jupyter notebook to quickly iterate on strategy logic at a high level – for example, to optimize indicator thresholds – and then take the best ideas into the Nautilus event-driven backtest for validation.)*

3. **Post-Trade Analysis & Metrics:** Once the backtest completes, you funnel the results into the analysis layer. The engine's output – say a Pandas DataFrame of equity over time or a list of trades –

is fed to **QuantStats** to produce a comprehensive performance report [10]. You'll get all the key metrics (Sharpe, Sortino, drawdown, exposure, etc.) plus nice plots, which serve as your "gold standard" evaluation of that run. Additionally, you perform **robustness checks**: e.g., use a Monte Carlo script to shuffle trade returns and generate a distribution of outcomes (to see if your observed performance might be luck), or run a **walk-forward optimization** if tuning parameters (split the data into rolling IS/OOS periods and optimize within Nautilus or via an external loop). This layer ensures the strategy is evaluated from every angle quantitatively.

4. **LLM Evaluation & Feedback:** Finally, the "Judge" comes into play. With the hard numbers and logs available, you invoke an **LLM-based agent** to review the strategy. For example, you might use a script (or the LLM Agent Trader framework) to have GPT-4 read the summary statistics and even some trade-by-trade notes. The LLM could be prompted to identify any concerning patterns (maybe it notices all big losses happened during news events) or to suggest what market regime the strategy is thriving in. This is done by providing the LLM with a structured report or even natural-language summary of the backtest. The result is an **AI perspective** on your strategy's behavior. This step can catch things you might miss – akin to having a veteran trader mentor looking over your shoulder, but in AI form. It can also generate ideas for the next iteration (e.g., "The LLM noticed performance dropped in 2022; maybe incorporate a volatility filter"). While this part is experimental, it's relatively easy to implement via API calls. In fact, the **LLM Agent Trader** project demonstrates a blueprint where after each backtest, there's a "Daily Feedback API" that queries the LLM for analysis [14] – you can adapt that concept to suit your needs.

In summary, the architecture would look like this: **Polars/ArcticDB for data** (fast, efficient feeding of massive tick sets), **Rust-powered engine for execution** (accuracy and speed in simulating trades) [1], **Python analytics for metrics** (to deeply understand performance) [10], and an **LLM overlay for qualitative insight** (closing the loop by learning from each backtest). Each component is the best in its class (open-source and free to use), and by keeping them decoupled, you maintain flexibility to swap or upgrade parts. For example, if a new faster engine comes out in Rust, you can plug it in; if you get more data, you might scale out to a distributed data store; if LLMs get better (they will), your "AI coach" only becomes more valuable. This stack, therefore, is not only high-performance and comprehensive today, but also **future-proof** for your algo-trading endeavors.

Sources:

- NautilusTrader docs – Rust core with Python, nanosecond event-driven simulation [1] [2]
- HFTBacktest (nkaz001) – models limit order latency, tick-level accuracy [5] [6]
- Barter-rs announcement – open-source Rust backtester, multi-threaded engine, tick support [8] [9]
- QuantStats library – comprehensive performance metrics and reports for strategies [10]
- VectorBT library – vectorized backtesting on pandas/NumPy for rapid strategy testing [11] [12]
- LLM Agent Trader – example architecture integrating GPT-4 into backtesting workflow [13] [15]
- Agentic Trading paper – multi-agent (LLM) framework with dedicated evaluation (Audit) agents [16] [17]
- Polars official site – Rust-based DataFrame, optimized for parallel, cache-efficient processing [20] [22]
- ArcticDB guide – high-performance time-series storage for financial data, built for fast analytics [23]
- Reddit discussion on ArcticDB – designed to quickly bring data into Python for analysis, favorably compared to flat files [24] [26]

[1] [2] [3] [4] GitHub - nautechsystems/nautilus_trader: A high-performance algorithmic trading platform and event-driven backtester
https://github.com/nautechsystems/nautilus_trader

[5] [6] [7] GitHub - nkaz001/hftbacktest: Free, open source, a high frequency trading and market making backtesting and trading bot, which accounts for limit orders, queue positions, and latencies, utilizing full tick data for trades and order books(Level-2 and Level-3), with real-world crypto trading examples for Binance and Bybit
https://github.com/nkaz001/hftbacktest

[8] [9] Barter-rs Major Update: Pure Rust Live-Trading & Backtesting Framework : r/rust
https://www.reddit.com/r/rust/comments/qw7gxr/barterrs_major_update_pure_rust_livetrading/

[10] GitHub - ranaroussi/quantstats: Portfolio analytics for quants, written in Python
https://github.com/ranaroussi/quantstats

[11] vectorbt: Getting started
https://vectorbt.dev/

[12] 1,000,000 backtest simulations in 20 seconds with vectorbt
https://www.pyquantnews.com/the-pyquant-newsletter/1000000-backtest-simulations-20-seconds-vectorbt

[13] [14] [15] GitHub - jason8745/llm-agent-trader
https://github.com/jason8745/llm-agent-trader

[16] [17] GitHub - Open-Finance-Lab/AgenticTrading
https://github.com/Open-Finance-Lab/AgenticTrading

[18] [19] GitHub - pipiku915/FinMem-LLM-StockTrading: FinMem: A Performance-Enhanced LLM Trading Agent with Layered Memory and Character Design
https://github.com/pipiku915/FinMem-LLM-StockTrading

[20] [21] [22] Polars — DataFrames for the new era
https://pola.rs/

[23] ArcticDB Financial and Historical analysis just way faster. - DEV Community
https://dev.to/tikam02/arcticdb-financial-and-historical-analysis-just-way-faster-5afa

[24] [25] [26] [27] Speed Test - ArcticDB, HDF, Feather, Parquet : r/algotrading
https://www.reddit.com/r/algotrading/comments/17z7hhd/speed_test_arcticdb_hdf_feather_parquet/