ChatGPT

GoldenSignalsAIv4 Project Analysis

Project Overview and Structure

GoldenSignalsAIv4 is an ambitious AI-powered trading signal platform aiming to provide real-time market data visualization with intelligent trading signals. The repository is organized as a multi-component system: a **FastAPI** backend (Python) under `src/`, a **React + TypeScript** frontend under `frontend/`, and supporting modules for data, ML models, and infrastructure. The backend (`src/main_simple.py`) launches a FastAPI app serving trading signals, market data, and analytics endpoints [1] [2]. The frontend is a React application (built with Vite) that provides an interactive dashboard for viewing live charts, signals, and analytics. The project uses a microservices-inspired architecture (multiple Dockerfiles for backend, frontend, "mcp" controller, ML, etc.) and extensive documentation for design and planning. Key directories include: `src/` for backend code (with subpackages for services and API routes), `frontend/` for the web UI, `ml_training/` for model files, and `agents/` (intended for multi-agent logic). This organization reflects the goal of separating concerns (data processing, ML, UI, etc.) as in an enterprise-grade system [3] [4]. Dependencies on the backend include **FastAPI**, **pandas/numpy** for data handling, **yfinance** and other APIs for market data, and **scikit-learn** for machine learning models [5]. The frontend leverages **React 18**, **Material-UI (MUI)** for UI components, **Zustand** for state management, and **Lightweight Charts** (by TradingView) for the charting library [6]. Overall, the repository provides a blueprint of a scalable trading assistant platform, combining live data feeds, algorithmic signal generation, and a professional UI.

Implemented Features vs. Desired Goals

The project's core goal is an AI-enhanced live stock charting system for day trading options. Many of the desired features are present in the current implementation, at least in preliminary form:

- **Real-Time Stock Chart with Predictive Overlays:** Implemented via the React frontend using TradingView-style **Lightweight Charts** for live candlestick and line charts [6]. The chart component (`TradingChart.tsx`) is designed to display real-time price data and overlay predicted price trajectories. The README confirms *"Price Projections – ML-powered price movement predictions"* as a feature [7]. In practice, the backend provides an option to **toggle ML-based price predictions** on the chart [8]. The backend's ML model service can generate forecasted price movements (the code loads a pre-trained `forecast_model.pkl` and outputs predicted price changes [9] [10]). These predictions are sent to the frontend, which then draws a trajectory or projected target on the chart. This means the system can overlay future price estimates on the live chart, though the accuracy depends on the underlying model.

- **Entry and Exit Signals (Buy/Sell indicators): Yes –** the platform generates trading signals and plots them on the chart as overlays. The README explicitly lists *"Signal Overlays – Buy/sell signals directly on charts"* as a feature [7]. In the backend, a `SignalData` structure encapsulates each signal (with fields for `signal_type` like "BUY" or "SELL", confidence, etc.) [11]. The FastAPI endpoint `/api/v1/signals/{symbol}` returns a signal for the requested symbol, including whether it's a buy/sell/

hold decision [2] [12]. The returned JSON contains the signal type and associated data, which the frontend uses to display **buy/sell markers** on the chart. For example, if a "BUY" signal is generated for a stock, the chart will show an entry marker at the corresponding price/time. This satisfies the entry/exit signal requirement – the logic for determining these signals is implemented in the backend using either technical indicators or an ML classifier (discussed below).

- **Take-Profit and Stop-Loss Markers: Partially implemented.** Each generated signal comes with suggested exit levels – the backend provides a `price_target` (take-profit) and `stop_loss` value as part of the SignalData [12]. These represent the recommended profit-taking price and the stop-loss threshold for the trade. The inclusion of these fields indicates the system is designed to compute and share TP/SL markers for each signal. While the frontend's exact implementation of displaying these markers isn't explicitly shown in the repository, it's likely that the chart component plots horizontal lines or flags at those price levels when a signal is active. The presence of `price_target` and `stop_loss` in the API response confirms the backend logic for computing them is in place (possibly using simple rules or an ML regression model). Thus, the architecture supports visualizing take-profit/stop-loss points on the live chart alongside entry signals.

- **Live Integration with Trading Data Sources:** Implemented. The system fetches real-time market data from external APIs and streams updates to the frontend. The backend's **MarketDataService** uses Yahoo Finance via the `yfinance` library for live quotes and also supports multiple data providers (Polygon.io, Alpha Vantage, IEX, Finnhub) through a unified **RateLimitHandler** module [13] [14]. This design allows failover: e.g., try Yahoo first, then fall back to another API if needed, with caching to reduce calls [15]. Real-time updates are delivered via WebSockets – the project includes a **WebSocketService** that connects to a streaming endpoint (planned `wss://stream.goldensignals.ai`) and broadcasts price ticks to subscribers [16] [17]. If the websocket fails, it can fall back to polling/SSE to ensure continuous data [18]. In summary, the backend actively integrates live market feeds: current price, bid/ask, volume, etc., are fetched and cached, then served to the UI. This achieves the goal of live data integration, albeit relying on third-party data services (with proper rate limit handling and caching to stay within API quotas [19]). The **live chart** on the frontend subscribes to these updates, so prices refresh in near real-time without page reload [20]. Historical data for charts can be retrieved as well (either from API or a mocked generator), enabling the user to switch timeframes on the chart.

- **AI/ML Components for Predictions and Recommendations: Present in a basic form.** The project incorporates AI/ML in the signal generation pipeline. Specifically, it uses **technical indicators** and pre-trained **machine learning models** to generate signals and forecasts:

- The backend computes a suite of **technical indicators** (moving averages, RSI, MACD, Bollinger Bands, volatility, volume ratios) on the fly using pandas [21] [22]. These indicators feed into decision logic or ML models.
- A lightweight ML layer is included via **scikit-learn models**. The code attempts to load several pickled models at startup: a price forecast model, a signal classifier model, and a risk assessment model [9] [23]. These were likely trained offline (in `ml_training/`) on historical data. If loaded successfully, they enable the system to **predict future price moves**, classify the market as bullish/bearish, and score the trade's risk. For example, `MLModelLoader.predict_price_movement()` uses the `forecast` model to predict the next price change given feature inputs [10]. Similarly,

`classify_signal()` returns probabilities for neutral/bull/bear signals using a classifier model [24], and `assess_risk()` outputs a risk score 0–1 using a risk model [25].

- The **trading signal generation** logic combines these outputs. In practice, when a `/api/v1/signals/{symbol}` request comes in, the backend likely gathers recent market data, computes indicator features, feeds them into the ML models, and decides a signal. The result is a recommended action (BUY/SELL/HOLD) with confidence and suggested TP/SL levels (possibly derived from the forecasted price or fixed rules). The code shows that if no model output is available, it safely defaults to a “HOLD” or neutral stance [26], ensuring the system responds even if AI models aren’t loaded or confident.
- **Multi-Agent System (AI agents):** The project envisions a sophisticated multi-agent approach (e.g. separate agents for technical analysis, sentiment analysis, risk, etc., reaching consensus). Documentation notes mention **“30+ specialized agents”** and a **Byzantine Fault Tolerance** style consensus for signal reliability [27]. In the current code, this multi-agent system is only partially realized. There is an `agents/` directory and references to an `agent orchestrator` (CrewAI integration) in the README [28], but the runtime uses a simplified pipeline at the moment. For instance, `main_simple.py` tries to include a `hybrid_signals` router for a hybrid sentiment system, but catches an import error if that module isn’t ready [29]. This implies that while the groundwork for a multi-agent AI system exists (in design and some code stubs), the **current implementation likely uses a single-agent or rule-based approach** for generating signals. Advanced AI features like transformer models or real-time sentiment from news (mentioned as goals [30]) are not yet integrated into the live system – these remain aspirational or in development. In summary, **basic AI/ML functionality is implemented (via indicator computations and scikit-learn models)**, providing a foundation for predictions and recommendations. However, more complex AI techniques (deep learning, NLP for sentiment, multi-agent consensus) appear to be planned but not fully in use yet, which is an important consideration when evaluating the system’s current capabilities.

Charting and Frontend Integration

The frontend charting is implemented in a **professional and interactive manner.** The project uses the **TradingView Lightweight Charts** library in the React app to achieve a fast, interactive stock chart similar to those on trading platforms [31]. This allows candlestick, line, and bar charts that are lightweight and efficient. Key integration points: - The React app’s Dashboard page displays the main trading chart along with other panels (signal list, agent performance, etc.) [32]. The chart component is designed to handle multiple timeframes (1D, 5D, 1M, etc.) and chart types, as indicated in the features [8]. Users can interactively switch time periods and symbols, and the chart will update accordingly. - Real-time updates are fed to the chart through WebSockets or polling. The backend provides a **WebSocket endpoint** (or pushes via a server-sent events fallback) that streams price updates and signal updates. The frontend subscribes to these, enabling **live price ticks** and immediate display of new signals without manual refresh [20] [33]. In effect, the chart and signal feed update in near real-time, which is critical for day trading use. - The chart overlay functionality includes drawing **signal markers** (to denote buy/sell points) and **price projection lines/areas** (to visualize the predicted trajectory). For instance, when “price projections” are enabled by the user, the frontend calls an API (or uses recent model output) to get the forecast data and then overlays a projected path or target marker on the chart [8]. Similarly, when a new signal (buy/sell) comes in, the chart component can draw an arrow or icon at the corresponding candle with perhaps a label of the action. - The UI also includes other components like **Agent Performance Monitoring** and **Market Pulse**, though these are beyond the chart itself [34]. The chart, however, is central and built to professional standards (the

README notes it's *"TradingView-style"* and inspired by TD Ameritrade's interface [35]). It supports interactive tooltips, real-time zoom/pan, etc., courtesy of the chosen chart library. - In terms of integration, the frontend communicates with the backend via a REST API and WebSockets. There is a dedicated API client module (`frontend/src/services/api.ts`) that handles all requests and real-time subscriptions [36] . This abstraction makes it easier to fetch data (using TanStack Query for caching updates) and propagate it to React components. The use of Zustand (state management) ensures that price and signal data are globally available to any component that needs them (e.g., both the chart and a ticker tape might consume the same live price state). - **Responsive Design:** The UI is designed to be responsive and uses MUI components and a dark theme for a trader-friendly experience [35] . Although not directly related to charting, this ensures the chart and dashboard can be viewed on different screen sizes (desktop monitors or tablets) without layout issues.

In summary, **chart integration is achieved through a modern web tech stack (React + TypeScript) with a specialized financial chart library**, giving users an experience comparable to professional trading platforms. The charts are interactive, update live with data from the backend, and include overlays for AI-driven insights (signals and predictions). This satisfies the core requirement of a real-time stock chart with extra intelligence layered on top.

Current Limitations and Shortcomings

While GoldenSignalsAIv4 is quite comprehensive in design, a few limitations or areas for improvement are evident on reviewing the implementation:

- **Incompleteness of Advanced AI Features:** Many cutting-edge features are outlined in the documentation (e.g. multi-agent consensus, transformer-based analysis, adaptive learning) [30] [27] , but these do not yet have a concrete presence in the code. The multi-agent system in particular appears to be in early stages – the codebase references it but mostly runs in a simplified mode (single pipeline). For instance, the *Hybrid Sentiment* router is optional and likely not fully implemented (the server logs a warning if it fails to load) [29] . Similarly, sentiment analysis from news or social media is not actively used; although the code can fetch news articles via yfinance [37] , there's no NLP processing of that news into sentiment scores or signals. This means current signals may rely predominantly on technical indicators and basic ML models, lacking the richer context that was envisioned.

- **Basic ML Models & Potential Accuracy Issues:** The ML components in use (scikit-learn models loaded from pickle) provide a starting point but might be too simplistic for reliable trading signals. There is no evidence of deep learning models or large-scale training – the mention of *transformer models* is aspirational [30] , but in code we only see classical models. The performance of these models is uncertain; no training code or dataset is provided in the repository. Moreover, the code is robust to missing models (it defaults to neutral predictions if models aren't loaded [38]), which is good for continuity but implies that if the model files are absent or not working, the system will still run but essentially produce generic outputs. Without sophisticated AI, the "predictive" overlays and signals might not be much better than simple rule-based signals. **In short, the AI might currently be more of a proof-of-concept** – using a few simple predictors – and would need further development or better models to truly add value for day trading.

- **Reliance on Third-Party Data & Latency:** The platform depends on external data APIs (Yahoo Finance by default, with others as backup). This introduces limitations:

- **Latency and Timeliness:** Free APIs like Yahoo Finance may not provide true real-time tick-by-tick data (often there's a slight delay). The architecture tries to mitigate this (e.g., using websockets where possible, caching aggressively, and targeting sub-10ms internal processing latency [39]), but ultimately the data speed is bounded by API limits. High-frequency traders might find this insufficient, though for typical retail day trading it's likely acceptable.
- **Rate Limits and Data Gaps:** The code includes a complex RateLimitHandler to avoid hitting API limits [40] [14] . Despite this, during fast market conditions or with many symbols tracked, the system might struggle if API quotas are exceeded or if network calls fail. The design has caching layers (in-memory, optional Redis, disk) to serve stale data if needed [15] [41] . While this is good engineering, it also means sometimes the system will serve cached data (with a warning) if live data isn't available [42] . Users should be aware when data is not fresh. Additionally, if any of the external providers change their API or experience downtime, the platform's data feed could be disrupted unless updated.

- **After-Hours and Market Coverage:** Handling of pre-market or after-hours data is likely limited. Documentation hints at *after-hours handling* strategies (there is `AFTER_HOURS_HANDLING.md` in the repo), but real-time after-hour quotes might not be fully integrated. The code's market hours check is basic (it simply defines 9:30–16:00 and treats outside as closed) [43] . This could be expanded for extended trading sessions. Also, the symbol set is somewhat fixed to major stocks/ETFs by default (a predefined list of symbols is returned for availability queries [44]), which might limit out-of-the-box exploration of less common tickers unless the user supplies API keys for robust symbol search.

- **Use of Mock Data in Places:** Not all functionality is wired to live data sources yet. For example, the endpoint for historical price data currently returns **mocked data** generated by a random walk algorithm in the code [45] . This is clearly labeled as a placeholder ("mock implementation" [46]) and produces synthetic historical candles rather than fetching true historical quotes. This shortcut, while useful for testing the UI, means features like backtesting or viewing past price trends aren't fully realized with real data unless the code is extended to use an API. It underscores that some parts of the system (likely those deemed lower priority for an initial version) are not complete. Similarly, the "analytics" endpoint returns a hard-coded performance summary (success rates, returns, etc.) for demonstration [47] , rather than calculated from real trading results. These placeholders should eventually be replaced with actual data processing logic or removed to avoid confusion.

- **Complex Architecture Overhead:** The architecture, while modular and enterprise-grade, might be **over-engineered** for the current stage of the project. There are many moving pieces – microservice separation, multi-agent framework, caching layers, etc. – which increases the complexity of development and deployment. For a single-developer or small team project, maintaining Kubernetes charts (`/k8s` directory) or Terraform scripts, for example, can be burdensome if the core application logic itself is still evolving. We see evidence of this complexity in places like dynamic import hacks in `main_simple.py` (manually loading the service module to avoid import issues [48]) and numerous config files. The team has done a commendable job setting this up, but the overhead might slow down iteration. Some components (e.g., a separate "MCP" server or the multi-agent orchestrator) may not be strictly necessary until the core functionality stabilizes. Streamlining the architecture – perhaps running as a single service for now – could make development easier. The presence of *"Streamlined Architecture"* and *"Redesign Roadmap"* docs suggests the team is aware of this and has been continuously refactoring. In summary, **complexity is a concern**: ensuring all these subsystems work in harmony is non-trivial, and there may be occasional integration bugs or resource usage inefficiencies due to this multi-layer design.

- **Lack of Trading/Broker Integration:** The system currently focuses on generating and displaying signals; it does not place trades or integrate with brokerage APIs. This isn't a flaw per se (it wasn't explicitly listed as a goal), but it's worth noting as a limitation for a "trading platform." A user would have to manually execute the trades based on the signals, or extend the platform to connect to a broker's API (like TD Ameritrade or Interactive Brokers) for automated trading. Given the scope, this feature is likely beyond the current project, but as an AI trading assistant, it might be an area of future growth.

- **Testing and Reliability:** Although the repository includes a testing framework setup (PyTest config) and even a `status-check.sh` utility, the actual test coverage is unclear. The README claims a high test coverage (87%) [49], which is promising if accurate. Still, the presence of many external integrations means unpredictable failures can occur (network issues, API changes). Ensuring reliability in a live trading scenario would require extensive testing (including simulation of data spikes, network latency, etc.). It's not obvious if the current system has been tested under such conditions or with real money scenarios – likely not yet. This is more a cautionary point: any trader using it should treat it as beta software until proven.

In summary, **GoldenSignalsAIv4 is feature-rich but some features are in prototype form**. The core functionality (real-time charts and basic AI signals) works with some clever engineering, but truly delivering on the full AI promise will require addressing these limitations. Filling in the gaps (e.g., replacing mocks with real data, improving model sophistication, simplifying where needed) will strengthen the platform's practicality and reliability.

Recommendations and Effective Use of LLMs (Claude Opus)

Improving a complex project like this can be accelerated by leveraging Large Language Models (LLMs) such as **Claude Opus** or GPT-4. Given the breadth of the system, an LLM can assist both in **code generation** and **architectural decision-making**. Here are some suggestions for enhancements and how an LLM could be utilized:

- **Refine and Extend the Codebase:** Use an LLM to help write boilerplate code for new features or to refactor existing modules for clarity. For example, as the team implements true multi-agent consensus or advanced analytics, they can prompt Claude with high-level instructions to generate initial code drafts. The repository's numerous planning documents suggest a workflow where an LLM might have been used to outline implementations. This can continue: developers can supply Claude with a description of a needed function (e.g., *"generate_signal should combine technical indicators X, Y, Z and model outputs to decide buy/sell with these rules…"*) and let it draft the code. The result can then be tested and refined. LLMs are especially good at producing structured, commented code quickly, which could save time on writing repetitive sections (such as additional API endpoints, data parsing routines, or new indicator calculations).

- **Improve Logic and Algorithms:** An LLM like Claude can analyze the current logic and suggest improvements. For instance, one might feed the **signal generation logic** (technical indicator thresholds, etc.) into the LLM and ask for ideas to make it more robust or sophisticated. Claude could propose alternative strategies (perhaps using combinations of indicators or different ML model architectures) that the developers can experiment with. Additionally, Claude's knowledge could help incorporate domain best practices – e.g., it might suggest using more advanced technical analysis techniques or confirm if the risk scoring approach is in line with industry standards. This kind of guidance can elevate the quality of the trading signals beyond what a single developer might encode.

- **Architecture Optimization:** The team can leverage an LLM to **review the system architecture** with a critical eye. By providing Claude with a summary of the current architecture (possibly extracted from the docs like the mermaid diagram and descriptions of each component), they can ask questions such as: "How can we simplify this architecture without losing functionality?" or "What are potential bottlenecks or failure points in this design?" The LLM might identify that some microservices could be merged to reduce overhead, or suggest using message queues vs. direct WebSockets in certain places, etc., drawing from patterns it has seen. Essentially, Claude can act as an architectural advisor, helping to validate or refine the chosen design. Its large context window (in the Opus version) means it could even ingest multiple config files or code files at once to get a holistic view.

- **Documentation and Knowledge Base:** Given the extensive documentation in the repo, maintaining and updating it is important. Claude can be used to quickly generate or update documentation whenever the code changes. For example, after implementing a new feature, the developer can have Claude draft the corresponding section in the README or a feature-specific markdown by describing the feature and maybe providing some code snippets. This ensures the docs stay in sync with the code. Moreover, Claude can assist in creating user-facing documentation or tutorials (e.g., "Explain how to use the backtesting module in simple terms"), which could improve adoption for new users. Since the project has a *Documentation Hub*, an LLM can accelerate writing thorough, clear docs for each module.

- **Testing and Simulation:** An innovative way to use an LLM is to generate test scenarios and even fake data to simulate trading conditions. Claude could be prompted to produce sample data streams (price movements, news headlines, etc.) that mimic real market scenarios (like a sudden crash, or a long uptrend) to test how the system's signals perform. It can also help generate unit tests by analyzing functions and suggesting edge cases to verify. For instance, feeding a function like `calculate_indicators()` into Claude and asking "what edge cases should we test for this?" might reveal scenarios (like divisions by zero, short data series, all constant prices, etc.) that the team should ensure are handled. By incorporating these suggestions into automated tests, the system becomes more robust. Additionally, one could use GPT/Claude to fuzz test the API – generating random but plausible API calls and seeing if any errors occur.

- **Enhancing AI Components:** If the project wants to integrate more advanced AI (like the planned transformer models or sentiment analysis), an LLM can assist in different ways. For example, Claude could help write the code to interface with a **transformer model** (perhaps using Hugging Face Transformers or an API) for pattern recognition in price data. It could also aid in building a **Retrieval-Augmented Generation** pipeline: e.g., generating code for storing news articles in a vector database (ChromaDB was mentioned) and using an LLM to answer questions about market context. In fact, Claude itself could be part of the solution – one could feed it relevant market news and ask it to summarize the sentiment or highlight any risk factors for a given stock, then feed that insight into the signal algorithm. Essentially, the development team can use Claude not only offline for coding, but potentially online as a component (with caution, given costs and consistency concerns). For now, using it offline to shape these ML/NLP integrations is safer: have the LLM produce a prototype of a sentiment analysis module or a summary of how to do multi-agent voting logic, then implement accordingly.

- **Code Quality and Consistency:** Large projects can suffer from inconsistent coding style or technical debt. An LLM can serve as a pair programmer to enforce best practices. The team could prompt it to review certain modules for any obvious bugs or inefficiencies. For example, running the `market_data_service.py` code through Claude and asking "Can you spot any potential performance issues or logical errors?" might reveal issues (maybe the random seed usage for mock data, or thread safety concerns with asynchronous code). LLMs often catch things like missing locks, error cases not handled, or suggest using more efficient data structures. While not a replacement for human code review, this can be a helpful extra set of eyes. Additionally, Claude can reformat or simplify complex code blocks to make them more readable and maintainable, which improves long-term development speed.

- **Prompt Engineering for Better Output:** To utilize Claude Opus effectively, the team should craft clear and specific prompts. Instead of asking broad questions, they can break tasks down. For instance:

- *"Here is our function for generating signals (provide code). Given the context of day trading, does this logic make sense? How can it be improved?"* – This prompt yields targeted feedback.
- *"Generate a Python FastAPI route that accepts a stock symbol and returns a JSON of computed technical indicators (list indicators needed) using our MarketDataService."* – This could be used to quickly stub new API endpoints.
- *"Suggest a consensus mechanism algorithm for combining outputs of multiple agent signals (technical vs sentiment) to decide a final signal, with resilience to one agent being wrong."* – This leverages the LLM's knowledge of consensus and could translate into pseudo-code or a design that the developers then implement.

In all cases, it's important to review and test the LLM-generated outputs. Claude can greatly speed up development by providing drafts and ideas, but the responsibility of verification lies with the team (especially critical in a financial context where errors can be costly).

- **Maintaining Focus on Value:** One recommendation is to use the LLM to help prioritize and possibly **de-scope** unnecessary complexity. By dialoguing with Claude about which features yield the most benefit to end users, the developers might decide to simplify certain aspects. For example, if the multi-agent complexity isn't yielding significantly better signals yet, an LLM might indirectly highlight that by focusing on simpler ensemble methods. This can prevent the team from getting lost in elaborate frameworks that aren't pulling their weight. Essentially, LLMs can not only generate code, but also help ensure the architecture remains *aligned with the project's core goals* by constantly questioning and refining the approach.

Overall, **Claude Opus can act as a versatile assistant**: a source of code snippets, an architecture consultant, a documentation writer, and a testing aide. By integrating such an LLM into the development workflow (via careful prompt engineering and iterative feedback), the team can accelerate development and improve the quality of GoldenSignalsAIv4. The key is to leverage Claude's strengths (vast knowledge, ability to handle the entire code context, fast drafting) while using human expertise for final judgment and tuning. With these improvements and assistance, GoldenSignalsAIv4 can more quickly mature into the fully realized AI-driven trading platform envisioned in its design documents – featuring robust live data handling, accurate predictive analytics, and a maintainable, high-quality codebase.

Sources:

- GoldenSignalsAIv4 README – Project features and architecture [50] [6]
- GoldenSignalsAIv4 Backend Code – FastAPI endpoints and services [2] [9]
- GoldenSignalsAIv4 Services – MarketDataService and RateLimitHandler (data sources, caching) [51] [13]
- GoldenSignalsAIv4 Signal Data Structure – includes signal type, confidence, price target, stop loss [12] [11]
- GoldenSignalsAIv4 Documentation – Ambitions for multi-agent AI and advanced ML [27] [30]
- GoldenSignalsAIv4 Code – Example of mock data usage for historical prices [45] (illustrating current limitations)

---

[1] [2] [12] [26] [29] [42] [43] [44] [45] [46] [47] [48] raw.githubusercontent.com
https://raw.githubusercontent.com/isaacbuz/GoldenSignalsAIv4/main/src/main_simple.py

[3] [4] [5] [6] [7] [8] [20] [31] [32] [33] [34] [35] [36] [50] GitHub - isaacbuz/GoldenSignalsAIv4
https://github.com/isaacbuz/GoldenSignalsAIv4

[9] [10] [11] [21] [22] [23] [24] [25] [37] [38] [51] raw.githubusercontent.com
https://raw.githubusercontent.com/isaacbuz/GoldenSignalsAIv4/main/src/services/market_data_service.py

[13] [14] [15] [19] [40] [41] raw.githubusercontent.com
https://raw.githubusercontent.com/isaacbuz/GoldenSignalsAIv4/main/src/services/rate_limit_handler.py

[16] [17] [18] raw.githubusercontent.com
https://raw.githubusercontent.com/isaacbuz/GoldenSignalsAIv4/main/src/services/websocket_service.py

[27] [28] [30] [39] [49] raw.githubusercontent.com
https://raw.githubusercontent.com/isaacbuz/GoldenSignalsAIv4/main/README.md