ChatGPT

Atlas: AI Trading System Architecture

Setup & Quickstart

Setting up **Atlas** locally involves preparing your Python environment, configuring Google Sheets access for logging, and setting API keys for any external services (brokers, LLMs, etc.). Here's how to get started:

- **Python Environment:** Install Python 3.9+ and create a virtual environment. Install required libraries (e.g. `pandas`, `numpy`, trading APIs, OpenAI SDK, `gspread` for Google Sheets). Using a requirements file or `pip install -r requirements.txt` helps keep the environment consistent.

- **Google Sheets API Config:** Enable the Google Sheets API and obtain credentials for your app (OAuth client ID or a service account) [1]. If using a service account for automation, share the target Google Sheet with the service account's email [2]. Install the Google API client or gspread library. For example, using **gspread** you would authenticate with a JSON key file and open the sheet by name or ID.

- **API Keys and Secrets:** Gather all necessary API keys: broker API keys (for Alpaca, Zerodha Kite, IBKR, etc.), OpenAI API key (for LLM calls), and any database keys (for a vector DB service). Store these in a secure config (environment variables or a `config.yaml`). For local testing, you can define them in a `.env` file and use Python's `os.getenv` to load them. Never hard-code secrets in code repositories.

- **Quick Test – Trade Logging:** To verify your setup, perform a dry run of logging a trade to Google Sheets. For example, append a new row to your Google Sheet with test values (timestamp, symbol, action, price, etc.). Using gspread, this can be as simple as:

```
import gspread
service_account = gspread.service_account(filename="credentials.json")
sheet = service_account.open("Atlas Trade Log").sheet1
sheet.append_row(["2025-08-08 10:00", "TEST", "BUY", 100, 1, "Filled"])
```

This should add a new row to your sheet. **Minimal steps**: (1) authenticate to Sheets, (2) open the spreadsheet, (3) call an append function to log a sample trade. If the row appears, your Google Sheets logging pipeline is working. (Remember that if using a service account, the sheet must be shared with it [2].)

- **Broker API Test:** Similarly, test connectivity to your broker in paper trading mode. For instance, using Alpaca's API, try fetching your account balance or submitting a small paper order (and

immediately cancel it) to ensure your keys and network access are correct. This ensures the system can interface with the market.

With the environment configured and a simple Sheets logging test passed, you're ready to explore the system's design and functionality in depth. The quickstart above validates that your local setup can connect to all external components (APIs and data sinks) before running the full trading system.

System Overview

Atlas is an AI-driven trading orchestration system with the goal of automating trade decision-making under strict risk-management constraints. It is designed to operate across multiple markets while enforcing conservative trading principles. Below is an overview of its scope and guiding rules:

- **Markets & Instruments:** Atlas supports both the Indian (NSE/BSE) and US markets, trading instruments like equities (stocks) and derivatives (options). The system can be extended to other markets with minimal changes by swapping broker modules (e.g., Zerodha for India, Alpaca or IBKR for US) and adjusting configurations such as currency and trading hours.

- **Capital Allocation Limits:** The system follows the classic *2% risk per trade* rule – no single trade should risk more than 2% of total capital [3]. For example, with a ₹10,00,000 portfolio, the maximum loss if a stop-loss hits would be ₹20,000. This constraint inherently sizes positions smaller for more volatile assets. It would take an unlikely string of many consecutive losing trades at 2% each to significantly draw down the account [4], protecting against rapid blow-ups.

- **Risk/Reward Targets:** Every trade is evaluated for a favorable risk-to-reward (RR) ratio. Atlas uses a minimum **3:1 RR ratio** guideline – meaning the potential profit target is at least three times the amount risked. A 3:1 ratio allows a trader to be profitable even with a win rate around only 30%, since one win can cover several losses [5]. Atlas will prefer trades where, say, ₹1 of risk could lead to ₹3 or more of gain. Lower RR trades are filtered out or require special justification from the signal agent.

- **Stop-Loss Enforcement:** The system **always uses stop-loss orders** (or an internal stop mechanism if the broker doesn't support server-side stops) for every position. Strict loss parameters are defined at trade entry and never widened – cutting losses is non-negotiable [3]. This disciplined approach ensures no single trade loss exceeds the predefined limit. **Take-profit levels** may also be set to secure the 3:1 reward when reached. By automatically placing stop-loss (SL) and take-profit (TP) orders, Atlas enforces its risk/reward rules without relying on human intervention.

- **Risk Controls:** In addition to per-trade limits, Atlas can enforce exposure limits like a max of, say, 5% of capital in any one stock or sector, or a cap on total open positions. It monitors leverage (for options or margin trades) to ensure it stays within safe bounds. If multiple signals trigger in close time proximity, a risk governance layer may prioritize or limit how many trades actually execute to avoid over-trading. The philosophy is that **preservation of capital comes first** – aggressive strategies are only executed if they fall within the predefined safety envelope. Automated systems can trade rapidly, so without such safeguards one could incur significant losses very fast [6]. Atlas's design explicitly mitigates this by implementing *dynamic stop-loss mechanisms and position limits* so the bot cannot make unchecked, risky trades [6].

- **High-Level Goals:** Ultimately, Atlas aims to **consistently compound returns** while avoiding catastrophic drawdowns. It blends AI-driven decision making (to capture opportunities in real time) with rule-based risk management (to ensure longevity). Success is measured not just by raw ROI, but by metrics like Sharpe ratio, maximum drawdown, and win/loss consistency. Atlas's constraints (market support, capital limits, risk rules) are intentionally conservative to align with professional trading practices and regulatory compliance, making the system robust rather than trying to "swing for the fences" on each trade.

Architecture Design

Atlas's architecture is modular and **agent-based**, with a central orchestrator governing specialized components. The design centers on a **Retrieval-Augmented Generation (RAG)** approach for decision making, combining a Large Language Model with various knowledge stores. Below is a breakdown of the major components and their interactions:

- **LLM Orchestrator:** At the heart of Atlas is an orchestrator agent powered by an LLM (e.g. GPT-4 via OpenAI API, with a local Llama2-based model as fallback). This agent is the **central decision-maker**, responsible for interpreting signals, querying memory, and deciding actions [7]. It receives inputs from other agents (signals, risk analysis, etc.), and uses the LLM to analyze and synthesize a trading decision or commentary. The LLM orchestrator can also chain through reasoning steps: for example, draft a trade plan, ask a "risk agent" to evaluate it, then refine the plan. This cognitive loop allows complex reasoning with natural language explanations for each trade decision.

- **Signal Agents:** These are modules that generate trade ideas. Atlas can host multiple signal agents: e.g. a **Technical Analysis Agent** (scans price patterns, technical indicators), a **Fundamental Analysis Agent** (checks earnings, valuations), and even a **Sentiment/NLP Agent** (scans news or social sentiment). They run in parallel and feed their suggestions to the orchestrator. For instance, the technical agent might flag a bullish breakout pattern, while the fundamentals agent notes the stock has strong financials – both inputs are considered. Each agent is relatively independent and focuses on its domain of expertise [8]. The orchestrator aggregates these perspectives to form a holistic view.

- **Memory Layers:** Atlas implements a hybrid memory system comprising multiple storage types:

- A **Vector Database** (e.g. Pinecone, Weaviate, or Chroma) stores unstructured textual insights – trade rationales, past strategy "lessons learned," significant market events, etc. When the LLM needs context (e.g. "Have we seen a similar signal in the past and what happened?"), it performs a semantic search on this vector store. Vector DBs enable semantic similarity search by storing embeddings of text data, allowing Atlas to **retrieve relevant past experiences by meaning** rather than exact words [9]. This is crucial for the RAG approach: the LLM retrieves relevant memories (like how a previous trade with 3:1 RR in this sector fared) and uses them to inform its generation. For example, the sentiment agent stores historical sentiment trends in a Pinecone vector store, which the system can query to see how sentiment shifts affected prices in the past [10].
- A **Structured Database** (SQL or time-series DB) stores numeric and tabular data: e.g. trade logs, performance metrics, current portfolio state, price history snapshots. This is used for fast lookups like "current exposure in each sector" or "P/L of the last 7 days". The structured DB complements the vector DB by handling data that is better suited to relational queries.

- **Google Sheets** serves as a logging and reporting interface. Every trade and key decision is logged to a Google Sheet (with timestamp, symbol, action, size, price, reasoning notes, etc.). This acts as both a redundancy (audit trail) and an accessible dashboard for developers or stakeholders to review trades in real-time. It's effectively a lightweight database for end-of-day summaries and can trigger email alerts or further analysis via Google scripts if needed.

- **Internal State:** The orchestrator also maintains an in-memory state (ephemeral memory) for the current trading session – e.g., the list of open positions, intraday calculations, and any interim reasoning chain of the LLM. This short-term memory resets or condenses each day to prevent stale data from persisting unnecessarily. By combining these layers, Atlas uses a **hybrid memory** approach: short-term scratchpad, long-term vector recall, and structured records [11].

- **Retrieval-Augmented Generation (RAG):** When the LLM orchestrator needs to make a decision or produce a rationale, it first performs retrieval from the memory layers. For example, if the technical agent flags a pattern, the orchestrator might retrieve similar historical instances from the vector DB and any notes on their outcomes. This retrieved context is then embedded in the LLM's prompt before it generates a response. RAG essentially **combines the LLM with an external knowledge base to improve outputs**, updating the LLM's "knowledge" in real-time without retraining [12] [13]. In Atlas, this means the LLM's trade reasoning can reference specific past events or data points ("According to memory, the last time we saw a signal like this, the trade failed due to earnings news – proceed cautiously.").

- **Risk Management Agent:** A dedicated risk agent acts as the watchdog. When the orchestrator proposes a trade (e.g. buy 100 shares of XYZ with stop at ₹95), the risk agent evaluates it against all risk rules *before* execution. It computes position size based on the 2% rule, ensuring the stop-loss distance and capital at risk align with the limit. It also checks the proposed trade against portfolio limits (sector concentration, number of open positions, etc.). If something is off, the risk agent will flag or adjust it (for instance, scale down the quantity to meet the 2% risk). This agent has authority to veto trades that violate risk parameters. In architecture terms, the risk agent sits between the "idea" phase and the "execution" phase as a gatekeeper [14], only green-lighting trades that pass all checks.

- **Execution Agent:** Once a trade idea is approved by risk management, it's handed to the execution module. This component interfaces with the broker API to place orders. It abstracts away broker-specific quirks so the rest of the system can issue generic trade commands. The execution agent is responsible for order formatting, handling acknowledgments, and tracking order status (filled, partially filled, etc.). It also immediately logs the action to the Google Sheet and the structured DB. If the broker supports bracket orders (entry with attached stop-loss and take-profit), the execution agent uses those to ensure stops are in place server-side. Otherwise, it will schedule internal monitoring to send a market order to exit if stop conditions are met.

- **Monitoring & Feedback:** The architecture includes a feedback loop where the outcome of trades and the performance of signals are fed back into the memory and agents. A **Monitoring service** tracks live positions and triggers the execution agent to exit positions if stop-loss or take-profit levels are hit (in case these aren't already automated by the broker). It also observes if any anomalies occur (e.g., an order rejection or a sudden price gap) and alerts the orchestrator to potentially adapt (the LLM could reason "the order failed, should I retry or cancel?" based on the error). Post-trade, the

system updates the vector memory with what happened (e.g. “Trade on XYZ hit the stop loss due to sudden news”) so that the LLM and agents learn from it.

- **OpenAI + Local LLM Fallback:** Atlas integrates with OpenAI’s GPT models for their strong capabilities, but it also includes a local LLM (such as Llama 2 running on local hardware) as a backup. This serves both as a **redundancy** and cost-control measure. The orchestrator is implemented such that if the OpenAI API call fails (due to downtime or rate limits) or if a query is deemed suitable for a smaller model, it will automatically route the request to the local LLM. Such a fallback ensures high availability – for instance, if OpenAI is experiencing an outage, the local model can pick up seamlessly with slightly reduced quality [15]. Moreover, less critical tasks (like parsing a news headline for sentiment) might be assigned to the local model to conserve API credits, whereas critical decision-making uses GPT-4. The two models operate in a “primary/secondary” configuration, where the secondary is always on standby. The system can also be configured to run both in parallel for a “race” condition, taking whichever returns first, which some production setups use to reduce latency and handle provider slowdowns [16] [15].

Overall, the architecture is **modular**: each agent (signal, risk, execution, etc.) can be developed and tested in isolation, and the orchestrator ties them together. The design emphasizes a separation of concerns (analysis vs. risk vs. execution) and uses the LLM as a high-level “brain” that can reason with natural language explanations using the data and tools provided. This provides transparency – the LLM can generate a rationale for each trade that can be logged (e.g., “Going long on ABC because technicals strong and risk within limits”). The block diagram would show the LLM orchestrator in the center, connected to the various agents (as inputs: signal agents and memory; as outputs: risk agent and executor), forming a loop of continuous learning and acting.

Broker and Market Abstraction

Trading brokers have different APIs and behaviors, so Atlas uses an **adapter pattern** to abstract broker interactions behind a common interface. In practice, this means a `Broker` class in Atlas defines standard methods like `get_market_data()`, `place_order(order)`, `get_position(symbol)`, etc., and specific broker implementations (AlpacaBroker, ZerodhaBroker, IBKRBroker, etc.) implement those methods according to their API. This layer makes Atlas *broker-agnostic* – switching from one broker to another is as simple as using a different adapter without changing core logic.

Key aspects of the broker/market abstraction design:

- **Unified Interface:** All broker adapters expose a unified set of capabilities. For instance, `place_order()` should accept a standardized order object (with fields like symbol, quantity, order_type, price, etc.) and internally translate that to the broker’s REST API call or SDK function. This way, the higher-level Atlas code doesn’t need to know if a broker uses REST, gRPC, or FIX – the adapter handles it. It also normalizes return values (e.g., every `place_order` returns an order ID or throws an exception on failure). This design follows the classic adapter/facade pattern from software engineering, making the rest of the system **loosely coupled** to any one broker’s specifics.

- **Supported Brokers:** Initial focus is on Alpaca (for U.S. stocks and crypto), Zerodha Kite (for Indian stocks and derivatives), and Interactive Brokers (global markets). Each has its nuances:

- *Alpaca:* Provides a REST API with both paper and live trading endpoints. Supports bracket orders (one cancels other for stop-loss/take-profit) which is convenient. The adapter for Alpaca will handle authentication (API key/secret), and provide methods like `get_account_balance()` and streaming quotes via WebSocket (for live price feeds if needed).
- *Zerodha (Kite API):* Requires a login flow to get a session token (since API key/secret alone aren't enough after every restart – there's a user PIN step). The Zerodha adapter might include a helper to automate or prompt for this on startup. Also, Indian markets have concepts like MIS/CNC (intraday vs delivery) and separate order validity (DAY/IOC) – the adapter maps Atlas's order parameters to these. Currency is INR and trading hours/timezone are different, so the MarketConfig for Zerodha will reflect that.

- *Interactive Brokers (IBKR):* IB has a comprehensive API (either via their TWS/Gateway or third-party libs). It can trade almost anything globally, but is more complex to integrate. The IBKR adapter would handle things like contract definitions (IB's way of specifying instruments) behind simpler calls. It may also implement additional features like checking if the market is open (since IB covers multiple exchanges). IBKR is known for reliability and low-level control, so Atlas could use it for advanced capabilities (like multi-currency portfolios).

- **Market Configuration:** Each broker adapter comes with a *market profile* – essentially configuration data for that market:

- **Currency and Units:** e.g., Alpaca returns prices in USD for US stocks, Zerodha in INR. Atlas's internal calculations need to be currency-aware. If Atlas runs multi-currency, a conversion step or separate risk accounting per currency might be needed.
- **Trading Mode (Paper vs Live):** The adapter should allow switching to paper trading endpoints for testing. For instance, Alpaca has separate API URLs for paper trading. The configuration can have a flag `paper_mode=True` to route calls accordingly. Similarly, IBKR's paper account or Zerodha's sandbox (if available) can be toggled.
- **Order Types & Features:** Some brokers support certain order types natively. For example, if broker supports **stop-loss (SL) and take-profit (TP) orders** server-side (as Alpaca does via bracket orders), the adapter can use them directly. If not (some brokers might not have native TP/SL on all instruments), the adapter will notify the execution agent to simulate them – meaning Atlas itself will monitor and execute an exit when price reaches SL/TP. The Market config might specify `supports_bracket=True/False`. Also, lot sizes or contract sizes (for options/futures) vary by market – these are defined in the config so that when placing orders the system rounds to the nearest lot size.

- **Timing and Rate Limits:** The adapter encapsulates any rate limiting rules of the broker API (e.g., "no more than 5 requests per second"). If needed, it queues or throttles requests to avoid bans. It also knows market trading hours/holidays for that exchange to avoid sending orders off-hours.

- **Example – Placing a Trade:** When the Execution agent wants to execute a trade, it calls a generic `Broker.place_order(trade)` method. Under the hood, the specific adapter is invoked. For example, `AlpacaBroker.place_order` will construct the REST POST to `/orders` with API keys, while `ZerodhaBroker.place_order` might use the official Kite SDK's function. Both will return a standardized response (like an `OrderResult` object indicating success or failure and order ID). By **integrating with broker APIs** in this manner, Atlas can trade on exchanges via those platforms (e.g.

sending orders to Alpaca, IBKR, etc.) [17] without the higher layers caring about *which* broker is in use.

- **Error Handling & Fallbacks:** The broker abstraction also handles errors gracefully. If an order is rejected (say due to insufficient margin or a stale price), the adapter catches that and passes a clear error message back to the system. In some cases, Atlas might have multiple broker connections (for instance, trading equities through a stock broker and crypto through a crypto exchange). The architecture could allow multiple active brokers, each with its own adapter, and route orders to the correct one based on instrument type. This could be configured via instrument tags (e.g., symbols starting with "BTC" go to CryptoExchangeAdapter, others go to StockBrokerAdapter).

In summary, the Broker Abstraction layer decouples Atlas's core logic from broker specifics. This not only makes Atlas flexible to run in different regions (IN/US) by swapping adapters, but also improves reliability: the rest of the system can assume trading operations behave uniformly. It simplifies testing too (one could make a PaperBrokerAdapter that just simulates trades for backtesting purposes). By **standardizing the API integration**, Atlas ensures that whether it's connected to a live account or a simulated one, the behavior remains consistent and safe.

Trade Loop Logic

At the core of Atlas is an automated **trade loop** – a cycle that continuously listens for opportunities, evaluates them, executes trades, and learns from the outcomes. This loop runs either in real-time (during market hours) or in backtest mode (simulated data). Here's the logical flow of the trade loop:

1. **Signal Generation:** The process begins with the signal agents scanning the environment. For example, the Technical agent might run every minute, checking for any stock hitting a breakout above a resistance level, or an option strategy meeting criteria. Simultaneously, a News/Sentiment agent could be parsing news feeds for significant headlines, and a Fundamentals agent may update any valuation metrics (though fundamentals change slower, this could run daily). When a signal agent finds a potential trade, it creates a **trade proposal** object (e.g. "Buy NIFTY 50 futures" or "Short TSLA stock") including context like entry price, reason, confidence score, etc. These proposals are fed into the orchestrator's queue for consideration.

2. **Context Retrieval (RAG):** Upon receiving a trade proposal, the orchestrator LLM enriches it with additional context. It queries the vector memory for related cases (e.g., "find past instances of breakout trades in this sector") and any stored playbooks or rules relevant to the scenario. For instance, if the proposal is to buy a tech stock, Atlas might retrieve a "tech sector playbook" or recall that an upcoming Fed meeting is today (stored as an event in memory). All this context is compiled into the prompt for the LLM. This step ensures the LLM decision-making is **informed by historical data and knowledge** beyond just the raw signal [13] .

3. **LLM Decision & Planning:** The orchestrator uses an LLM prompt that includes the signal, the retrieved context, and a request to draft a *trade plan*. The LLM might output something like: "Signal suggests buying XYZ. Stop loss at ₹95 (2% risk), target at ₹110 (3:1 RR). Rationale: strong earnings and technical breakout. Proceed with caution due to upcoming earnings release next week." This response is essentially the AI's decision and rationale. It may also decide to reject the signal: e.g., "No trade – conditions not favorable (low RR or conflict with another position)." The natural language

output is parsed (or structured via a JSON format in the prompt) to extract the action (trade or no trade, and trade details).

4. **Risk Assessment:** Before any trade action is taken, the plan goes through the **Risk Management agent** (the rule-enforcer). The risk agent computes the position sizing: using the account's current equity and the stop-loss distance, it calculates how many shares or contracts correspond to a 2% risk. For example, if capital is ₹1,00,000 and stop distance is ₹5, risking ₹2,000 means 400 shares (since ₹5 * 400 = ₹2000). If the LLM's suggested size is larger, the risk agent will downsize it to 400. It also checks total exposure: if the trade is in the same direction as existing positions, does it push sector exposure over the limit? If yes, it may reduce size further or reject the trade. It ensures the **3:1 reward-to-risk** is satisfied – i.e., the target price is checked relative to the entry and stop. If the LLM suggested target yields less than 3:1, the risk agent might adjust the target or decide the trade doesn't meet criteria. Only if the plan passes all risk checks (position size, SL/TP levels, capital availability, etc.) does it proceed. This is essentially a compliance gate that *must* approve the trade before execution [6].

5. **Execution:** Now the Execution agent takes over to actually place the trade. It constructs the order according to the broker adapter's requirements. For instance, a buy order for 400 shares of XYZ at market with a stop-loss at ₹95 and take-profit at ₹110 would be placed (if the broker supports bracket orders, it places a bracket with those exit orders attached; if not, Atlas will handle exits manually). The trade is sent through the `Broker` interface, which returns an order ID and confirmation. The execution step is critical to do quickly and correctly – Atlas optimizes for low latency here to avoid slippage. If multiple exchanges or liquidity sources are available, the system could incorporate smart order routing (though for simplicity, in one broker context this isn't needed). Once an order is confirmed, Atlas immediately logs the details.

6. **Logging & Reporting:** As soon as a trade is executed (or even when it's sent), Atlas logs it. A row is added to the Google Sheet (with all relevant fields: timestamp, symbol, side (buy/sell), quantity, entry price, stop, target, etc., along with a comments column for the AI's rationale). This provides a real-time ledger of trades for transparency. Additionally, an internal log (in the structured DB) records it for performance metrics. The logging system could also trigger notifications – e.g., send a Slack message or email summarizing the trade: "Atlas BOT: Bought 400 XYZ @100, SL=95, TP=110. Reason: breakout + earnings beat." Such notifications are useful for monitoring and audit.

7. **Monitoring & Management:** After execution, Atlas monitors the open position. If it placed actual stop-loss and take-profit orders with the broker, it waits for one to execute (or the user to intervene). If not (in case of a broker or instrument that doesn't allow pre-set SL/TP), Atlas runs a monitoring loop: it continuously checks the latest price (via broker API or data feed) and will send an order to close the position if the stop or target is reached. This monitoring also looks for **trailing stop** adjustments if that's part of the strategy (for instance, if the price moves favorably, the system could trail the stop to lock in profit, as per rules). It's essentially the position management phase.

8. **Post-Trade Analysis:** Once the trade closes (either hit target, hit stop, or manually exited), Atlas moves into a learning mode. The outcome (profit or loss, and any notable events during the trade) is recorded. The LLM orchestrator (or a dedicated *Self-Critique agent*) will analyze the completed trade. This might involve prompting the LLM with details: "Trade result: stopped out. What might have been missed?" The LLM can reflect on whether the rationale held up or if, say, a news event changed

the outcome. This reflection and any lessons (e.g., *"Stock was downgraded by a bank after entry, causing stop-hit. In future, check for scheduled analyst reports."*) are then saved into the vector DB memory as a new data point [18]. Over time, this builds a knowledge base of do's and don'ts that the system can draw upon for future decisions.

9. **Repeat and Evolve:** The loop then continues back to signal scanning. Atlas runs continuously during market hours, and possibly does an end-of-day summary. The architecture also supports **learning loop** improvements: for example, if a certain pattern of trades consistently fails, Atlas can flag that pattern in memory and the LLM might start to ignore or question those signals. In a more advanced extension, this could transition into a reinforcement learning style update, but initially, it's more of a rule-based learning (augmented by LLM's ability to summarize patterns).

Throughout this loop, there are numerous **safety checks** and **monitoring hooks**. For example, a separate watchdog process can monitor if Atlas's P&L for the day exceeds a loss threshold and automatically halt trading (a kill switch to prevent a runaway loss day). Additionally, performance metrics like win rate, average R:R achieved, etc., are computed on the fly to adjust strategy if needed (e.g., if trades are consistently only achieving 2:1 RR instead of 3:1, that's feedback to adjust targets or be more selective).

By structuring the trading logic in this stepwise manner – Signal → Plan (LLM) → Risk Check → Execute → Monitor → Learn – Atlas ensures a disciplined approach. It's not blindly following signals; it's analyzing them in context, enforcing risk, and learning from each result. This cycle can run very fast (for high-frequency signals) or at a human-like pace (for daily swing trade signals), depending on configuration. The LLM makes it possible to have a *reasoned* approach to each trade rather than just a hard-coded algorithm, which is a unique aspect of Atlas's loop.

Python Code Skeletons

To illustrate the implementation, below are simplified Python code skeletons for key components of Atlas. These snippets highlight the structure and interactions without full detail, serving as a starting template for development.

Configuration & Settings

The config holds all tunable parameters and API keys. Using a dataclass or dictionary for structure:

```
# config.py
from dataclasses import dataclass

@dataclass
class AtlasConfig:
    # API Keys
    openai_api_key: str
    broker: str                 # e.g., "alpaca" or "zerodha"
    broker_api_key: str
    broker_api_secret: str
    # Google Sheets
```

```
    gsheet_creds_file: str
    gsheet_doc_name: str
    # Risk Parameters
    max_risk_per_trade: float   # e.g., 0.02 (2%)
    min_reward_ratio: float     # e.g., 3.0 (3:1 RR)
    max_positions: int          # max concurrent open trades
    # Market/Trading Settings
    base_currency: str          # "USD" or "INR"
    paper_trading: bool         # True for paper trading mode
    market: str                 # "US" or "IN"
    trading_hours: tuple        # (start_hour, end_hour) as per market timezone

# Example instantiation
config = AtlasConfig(
    openai_api_key="sk-***",
    broker="alpaca",
    broker_api_key="AK***", broker_api_secret="SK***",
    gsheet_creds_file="credentials.json",
    gsheet_doc_name="Atlas Trade Log",
    max_risk_per_trade=0.02, min_reward_ratio=3.0,
    max_positions=5,
    base_currency="USD", paper_trading=True,
    market="US", trading_hours=(9.5, 16)  # 9:30-16:00 for NYSE
)
```

This configuration is passed to various components so they know the operating parameters. For example, the risk agent will read `max_risk_per_trade` and `min_reward_ratio` from here.

Google Sheets Logger

A simple logger that appends trades to a Google Sheet using `gspread`:

```
# logger.py
import gspread
from datetime import datetime

class SheetsLogger:
    def __init__(self, creds_file: str, doc_name: str):
        self.gc = gspread.service_account(filename=creds_file)
        self.sheet = self.gc.open(doc_name).sheet1

    def log_trade(self, symbol, side, quantity, price, stop, target, reason):
        row = [
            datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
            symbol, side, quantity, price, stop, target, reason
        ]
```

```
        try:
            self.sheet.append_row(row)
        except Exception as e:
            print(f"[Logger] Failed to log to Google Sheets: {e}")
```

Usage example: `logger.log_trade("AAPL", "BUY", 100, 150.0, 145.0, 160.0, "Breakout signal")` would append a row with the current timestamp and these details. The logger catches exceptions to ensure a logging failure doesn't stop the trading loop (it would just print an error).

Risk Management Agent

This agent ensures each trade adheres to risk limits and computes position sizing:

```
# risk_agent.py
class RiskAgent:
    def __init__(self, config):
        self.max_risk = config.max_risk_per_trade  # e.g. 0.02 (2%)
        self.min_rr = config.min_reward_ratio     # e.g. 3.0 (3:1)

    def assess_trade(self, equity, symbol, side, entry_price, stop_price,
target_price):
        """Return adjusted quantity if trade is allowed, or 0 if rejected."""
        # Calculate risk per share/contract
        risk_per_unit = abs(entry_price - stop_price)
        if risk_per_unit <= 0:
            return 0  # invalid stop (e.g., stop not set below entry for long)
        # Position size for max risk
        max_loss_amount = equity * self.max_risk
        quantity = max_loss_amount // risk_per_unit  # integer division to get
units
        if quantity <= 0:
            return 0  # equity too low for even 1 unit at this risk
        # Check risk-reward ratio
        if target_price and stop_price:
            expected_rr = abs(target_price - entry_price) / abs(entry_price -
stop_price)
            if expected_rr < self.min_rr:
                return 0  # reject trade, RR not favorable
        # Additional checks (could include portfolio exposure limits, etc.)
        return int(quantity)
```

This `assess_trade` method takes current equity and the trade details, then computes how many shares/contracts to trade such that losing to the stop loss equals `max_risk` of equity. It also verifies the expected RR ratio. It returns `0` quantity if the trade should be rejected (e.g., if even 1 share exceeds risk, or RR is too low). The orchestrator or execution logic will interpret `0` as "don't execute".

Broker Interface

Abstract base class and two example implementations (pseudo-code):

```
# broker.py
from abc import ABC, abstractmethod

class BrokerBase(ABC):
    @abstractmethod
    def get_price(self, symbol):
        pass

    @abstractmethod
    def place_order(self, symbol, side, quantity, order_type, price=None,
stop_loss=None, take_profit=None):
        pass

# Alpaca implementation example
import requests

class AlpacaBroker(BrokerBase):
    def __init__(self, api_key, api_secret, paper=True):
        base_url = "https://paper-api.alpaca.markets" if paper else "https://
api.alpaca.markets"
        self.base_url = base_url + "/v2"
        self.session = requests.Session()
        self.session.headers.update({
            "APCA-API-KEY-ID": api_key,
            "APCA-API-SECRET-KEY": api_secret
        })

    def get_price(self, symbol):
        # call Alpaca's market data API (v2) for last quote
        resp = requests.get(f"{self.base_url}/stocks/{symbol}/quotes/latest")
        data = resp.json()
        return data["quote"]["ap"]  # ask price as example

    def place_order(self, symbol, side, quantity, order_type="market",
price=None, stop_loss=None, take_profit=None):
        order = {
            "symbol": symbol,
            "qty": quantity,
            "side": side.lower(),        # "buy" or "sell"
            "type": order_type,         # "market" or "limit"
            "time_in_force": "day"
        }
        if order_type == "limit":
```

```
            order["limit_price"] = price
        # If stop-loss/take-profit provided, use bracket order
        if stop_loss or take_profit:
            order["order_class"] = "bracket"
            if take_profit:
                order["take_profit"] = {"limit_price": take_profit}
            if stop_loss:
                order["stop_loss"] = {"stop_price": stop_loss}
        resp = self.session.post(f"{self.base_url}/orders", json=order)
        if resp.status_code == 200:
            return resp.json().get("id")  # return order ID
        else:
            raise Exception(f"Order failed: {resp.text}")

# Zerodha implementation example (using hypothetical KiteConnect library)
class ZerodhaBroker(BrokerBase):
    def __init__(self, api_key, access_token):
        from kiteconnect import KiteConnect
        self.kite = KiteConnect(api_key=api_key)
        self.kite.set_access_token(access_token)

    def get_price(self, symbol):
        # symbol format might be different in Zerodha (exchange:token)
        quote = self.kite.quote(symbol)
        return quote[symbol]["last_price"]

    def place_order(self, symbol, side, quantity, order_type="MARKET",
price=None, stop_loss=None, take_profit=None):
        transaction_type = self.kite.TRANSACTION_TYPE_BUY if side.lower()=="buy"
else self.kite.TRANSACTION_TYPE_SELL
        order_params = {
            "tradingsymbol": symbol,
            "exchange": "NSE",
            "transaction_type": transaction_type,
            "quantity": quantity,
            "order_type": order_type,
            "product": "MIS",  # intraday product code
        }
        if order_type == "LIMIT":
            order_params["price"] = price
        # Zerodha doesn't support server-side bracket orders in 2025 for MIS;
handle SL/TP manually
        order_id = self.kite.place_order(**order_params)

# If stop_loss is set, immediately place a separate stop order (for MIS, use SL
order)
        if stop_loss:
            sl_params = order_params.copy()
```

```
            sl_params.update({
                "order_type": "SL",
                "price": stop_loss, "trigger_price": stop_loss,
                "transaction_type": self.kite.TRANSACTION_TYPE_SELL if
side.lower()=="buy" else self.kite.TRANSACTION_TYPE_BUY
            })
            self.kite.place_order(**sl_params)
        return order_id
```

These classes show how different the implementations can be. `AlpacaBroker` uses REST calls and can place bracket orders in one go [19], whereas `ZerodhaBroker` uses the Kite SDK and must place a separate stop-loss order because of the broker's limitations. The Atlas orchestrator would choose which broker class to instantiate based on config (e.g., if `config.broker == "alpaca"` then use AlpacaBroker).

Agent Orchestrator Loop

The main loop coordinating agents might look like this:

```
# orchestrator.py
import openai
from time import sleep

class Orchestrator:
    def __init__(self, config, broker, risk_agent, logger, memory):
        self.config = config
        self.broker = broker
        self.risk = risk_agent
        self.logger = logger
        self.memory = memory  # vector DB or similar for context
        openai.api_key = config.openai_api_key

    def run(self):
        equity = self.broker.get_account_equity()  # hypothetical method to get
balance
        while True:
            # Pseudocode: fetch signals from agents (could be event-driven in
practice)
            signals = self.get_signals()
            for sig in signals:
                symbol, side, entry_price, stop_guess, target_guess, reason =
sig
                # Retrieve context from memory for this symbol/sector
                context = self.memory.search(f"{symbol} trade rationale")
                # Compose LLM prompt
                prompt = (f"Trade signal: {side} {symbol} at {entry_price}. "
                          f"Stop ~{stop_guess}, Target ~{target_guess}. Reason:
```

```
{reason}.\n"
                        f"Context: {context}\n"
                        "Assess this trade and suggest position size, precise
stop and target. Respond in JSON.")
                try:
                    response = openai.ChatCompletion.create(
                        model="gpt-4",
                        messages=[{"role": "user", "content": prompt}]
                    )
                except Exception as e:
                    # Fallback to local LLM or GPT-3 if GPT-4 fails
                    response = fallback_local_model(prompt)
                plan = self.parse_response(response)  # extract dict:
{"symbol":..., "side":..., "stop":..., "target":..., "confidence":...}
                # Risk check
                qty = self.risk.assess_trade(equity, plan["symbol"],
plan["side"], entry_price, plan["stop"], plan["target"])
                if qty and qty > 0:
                    order_id = self.broker.place_order(plan["symbol"],
plan["side"], qty, order_type="MARKET",

stop_loss=plan.get("stop"), take_profit=plan.get("target"))
                    self.logger.log_trade(plan["symbol"], plan["side"], qty,
entry_price,
                                          plan.get("stop"), plan.get("target"),
plan.get("reason", "LLM trade"))
                    # Save rationale to memory
                    rationale = (f"{plan['side']} {plan['symbol']} at
{entry_price}, SL={plan['stop']}, TP={plan['target']} -> {plan.get('reason')}")
                    self.memory.add_document(rationale)
            sleep(1)  # wait or sync to next data update interval

    def get_signals(self):
        # In a real system, this would collect outputs from signal agents.
        # Here we just stub a list or pull from some signal queue.
        return []
```

In this skeleton, the orchestrator loop fetches signals (the implementation of `get_signals` is abstracted – it might pull from agent threads or a message queue). For each signal, it prepares a prompt including the raw suggestion and retrieved context from memory. It then calls the OpenAI API (with error handling to use a fallback model if needed). The response is parsed into a structured plan. The plan is passed to `RiskAgent.assess_trade` to get an allowed quantity. If a positive quantity is returned, it means the trade is approved – the orchestrator calls the broker's `place_order` and logs the trade. It also adds the trade rationale to memory for learning. The loop runs continuously with a short sleep, or could be event-driven (e.g., wake up on new data or signals).

This orchestrator design ensures each decision goes through the full cycle: raw signal → LLM reasoning → risk filter → execution → logging. Many improvements can be made (like more sophisticated parsing, handling partial fills, etc.), but this gives the high-level structure.

(Note: In actual implementation, concurrency and asynchronous processing might be used. Signal agents could run on separate threads or processes and feed into the orchestrator. The loop might also have to manage time (only trade during market hours from config) and implement a graceful shutdown or daily reset. Those details are omitted here for clarity.)

Backtesting Strategy

Before deploying Atlas with real money, robust backtesting is essential. This section outlines how to backtest strategies and the recommended approach to evolve towards reinforcement learning.

- **Choosing a Backtesting Framework:** Two popular Python backtesting frameworks are **Backtrader** and **vectorbt** (VectorBT). Backtrader is known for its user-friendly, event-driven approach and rich feature set, whereas vectorbt offers high-performance, vectorized simulations. Backtrader is often favored for its simplicity and reliability – it "just works" and provides a powerful, coherent API for strategy development [20]. You can define your strategy in an `next()` method reacting to each new bar, making it intuitive for many traders. VectorBT, on the other hand, is designed for speed at scale, leveraging NumPy and Pandas under the hood to eliminate Python loops. It can run **fully vectorized backtests**, processing entire time series operations at once, which makes it **blazing fast** for large datasets [21]. For example, vectorbt can simulate thousands of parameter combinations across years of minute data extremely quickly, whereas Backtrader might take significantly longer due to iterative processing.

- **Custom vs Library Backtesting:** For Atlas's architecture, a custom backtesting approach might be used initially. This could mean simulating the trade loop logic on historical data: feeding historical prices to the signal agents, having the orchestrator make decisions (potentially with the LLM in the loop for realistic behavior, or a stubbed rule set for repeatability), and recording trades. The advantage of a custom backtester is that you can incorporate Atlas's full logic (including the LLM reasoning if desired) to see how it would have performed in the past. However, it's more work to build. Using a library like Backtrader or vectorbt can speed up strategy iteration – you might encode a simplified version of Atlas's strategy in those frameworks to get quick performance metrics. For example, Backtrader could be used to test a purely technical version of the strategy (since integrating an LLM into Backtrader's loop is non-trivial). **Recommendation:** Start with simpler custom backtests to validate individual components and rules, then gradually integrate with more sophisticated frameworks. This means first ensure that, say, the 2% risk rule and 3:1 RR filter actually improve outcomes by testing on historical data. Then, one can use FinRL or others for more advanced analysis.

- **FinRL vs Traditional Backtesting: FinRL** (Financial Reinforcement Learning library) is an advanced framework that applies deep reinforcement learning to trading strategies [22]. It provides a gym-like environment to train an agent (like a deep neural network) to make trading decisions through trial and error. While powerful, FinRL introduces complexity: defining state/action space, lengthy training times, and less interpretability. We suggest a phased approach:

- **Phase 1 – Rule-based Backtesting:** Use Backtrader or vectorbt to simulate the rule-based strategy (the core logic Atlas uses, sans LLM or with LLM decisions approximated by rules). Evaluate performance metrics like win rate, average profit per trade, max drawdown, Sharpe ratio, etc. Ensure risk controls are effective (e.g., no single trade loss exceeds 2% in the backtest) and that the strategy shows a positive expectancy.
- **Phase 2 – Iterative Refinement:** Tweak the strategy rules or parameters based on backtest results. Perhaps the backtest reveals that a 2.5:1 RR still yields good performance with higher win rate; or maybe certain technical signals fare better in certain market regimes. Apply those insights and re-test.

- **Phase 3 – Reinforcement Learning (FinRL):** Once a solid baseline is established, consider using FinRL to potentially discover improved policies. For example, define the state (technical indicators, account equity, positions) and reward (e.g., incremental profit with penalties for risk rule violations) and let an RL agent train on historical data. FinRL provides market environments and has shown promise in developing trading agents [23]. Initially, use FinRL in a constrained way (maybe allow it to adjust some parameters of your strategy). Over time, one could allow the RL to take more control (like deciding position sizing dynamically). Keep in mind RL will require many episodes of training and careful tuning to avoid overfitting.

- **Stop-loss & Take-profit Simulation:** In backtests, especially with vectorized frameworks, handling stop-loss and take-profit conditions is important. Backtrader supports specifying stop orders or one-cancels-other logic, but if using vectorbt or custom, you'd manually simulate this: e.g., if a trade is active, check each subsequent bar if low price hit stop or high hit target first, and exit accordingly. It's crucial to simulate realistic order execution: assume stop orders fill at stop price (or maybe with some slippage), and targets likewise. This will give a more accurate reflection of performance, especially for strategies relying on tight stop-losses.

- **Performance Metrics & Evaluation:** Analyze the backtest results comprehensively:

- **Win Rate and Payoff:** With a 3:1 RR strategy, you might expect a win rate somewhere around 30-50%. Verify that in backtests. A low win rate is fine if RR is high, but if you notice win rate dropping too low, the drawdowns may be large.
- **Max Drawdown:** Ensure the maximum equity drawdown in backtests is within tolerances (maybe <20% for a moderate risk strategy). If a single bad period causes a 50% drop, reconsider risk controls or strategy parameters.
- **Sharpe and Sortino Ratios:** These give insight into risk-adjusted returns. A Sharpe > 1 is decent; > 2 is very good for trading strategies. Check these to gauge consistency.
- **Exposure and Turnover:** How much of the time was the strategy in the market? How frequently does it trade (turnover)? A very high turnover might incur costs or be impractical live, so ensure the assumed transaction costs (commissions, slippage) are accounted for and that performance remains positive after them.

- **Regime analysis:** It can help to break the backtest by periods (bull vs bear markets, high-volatility vs low-volatility regimes) to see where it performs well or poorly. Atlas might perform best in trending markets if it's momentum-driven, for instance, and struggle in choppy sideways markets – the backtest will reveal that.

- **Iterating Safely:** Backtesting can lead to overfitting if one is not careful (especially with many parameters or the flexibility of an LLM making decisions). Use techniques like cross-validation on different time periods, walk-forward analysis, and keep some data as out-of-sample test. The goal is to ensure the strategy generalizes and the risk controls truly protect against unforeseen scenarios.

In conclusion, **backtesting is a critical step** to validate Atlas's strategy before live deployment [24]. Start simple (rule-based, known frameworks), get a baseline, then gradually incorporate the full Atlas complexity (LLM decisions, multi-agent interplay). Only once the strategy proves itself on historical (and ideally live paper) performance should one allocate real capital. This disciplined approach will provide confidence in the system's robustness.

LLM + Memory Integration

One of Atlas's distinguishing features is the integration of a Large Language Model with a memory system to inform and improve trading decisions. This **LLM + Memory** design follows the Retrieval-Augmented Generation (RAG) paradigm, where the LLM's outputs are augmented by relevant retrieved information. Let's break down how Atlas uses this and what data it stores:

- **RAG Workflow:** The LLM doesn't operate in isolation; whenever it needs to make a decision or produce an analysis, it first retrieves relevant data from the memory stores. *Retrieval-Augmented Generation* combines the generative power of LLMs with an external knowledge base [12]. In Atlas, before the LLM orchestrator writes a trade rationale or decides on an action, it queries the vector database for related content (past trades, market events, notes) and includes those in the prompt. This effectively extends the LLM's context window with domain-specific knowledge on-the-fly. For example, if considering a trade on a pharmaceutical stock, Atlas might pull in notes of "FDA approval news significantly moves Pharma stocks" from its memory, guiding the LLM to factor that in. The result is outputs that are more grounded and tailored to Atlas's prior experiences.

- **Memory Content (What is stored):** Atlas's memory can be thought of in **layers**:

- **Trade Rationales:** After each trade (win or loss), the self-critique process generates a short summary of why the trade succeeded or failed. These summaries are stored in the vector DB. Over time, this builds a knowledge base of *lessons*. For instance, *"Buying breakouts in energy stocks failed during period of high oil volatility – manage risk accordingly"*. When a similar scenario arises, the LLM can recall these lessons.
- **Outcomes and Metrics:** The system can store key metrics or outcomes related to certain strategies. E.g., *"Last 10 breakout trades: 6 wins, 4 losses, avg RR 2.8:1"*. This might be stored in a structured DB but could also be embedded as a piece of text for the LLM to recall trend-level info.
- **Playbooks/Rules:** Atlas might have heuristic "playbooks" (possibly written or refined by humans) that describe how to handle certain events – e.g., *"If major central bank announcement today, do not trade 30 min before/after"* or *"During earnings season, prefer option strategies to limit risk"*. These can be stored as documents in the vector DB. The LLM, when formulating a plan, can fetch these playbook items when context matches (thanks to semantic search) and then essentially follow that guidance. This approach gives Atlas a form of *institutional memory* or policy that persists.
- **External Knowledge Snippets:** Relevant news or research insights can be embedded. For example, if the Fed changes interest rates, a summary of that event and its implications can be added to memory, tagged with date and category. Then, for any trade involving bank stocks, the LLM might

retrieve *"Fed raised rates by 0.25% on 2025-08-01, financial sector historically dips after rate hikes"*. This ensures the LLM is not stuck with only static training data (which might be outdated) but is updated with current facts.

- **Conversation / Internal Dialogue:** If Atlas engages in multi-step reasoning via the LLM (like chain-of-thought), some of that dialogue could be stored transiently. However, this is usually not kept long-term unless it contains a nugget of insight, to avoid clutter.

- **Vector Database Usage:** The vector DB (like Pinecone, Qdrant, or Chroma) stores embeddings of the above pieces of text. Each entry might include metadata (date, tags like symbol or sector, type of info). The advantages of using a vector store here are semantic search and scalability – Atlas can quickly find *conceptually* related info, not just exact keyword matches [25]. This means even if the current trade is about "AAPL earnings breakout" and a past memory talks about "Microsoft earnings volatility", the semantic similarity (tech sector, earnings) could flag that memory, whereas a normal DB search might miss it if keywords differ. The vector DB effectively acts as **long-term memory** for the LLM agent, giving it context and facts to recall [26]. This makes Atlas's decision-making more informed and less likely to repeat past mistakes.

- **Embedding & Caching:** When new data is stored (a new trade rationale, or a new playbook entry), it is converted to an embedding via an encoder model (like OpenAI's text-embedding-ada or a local SentenceTransformer). To optimize, Atlas will cache embeddings for any frequently used text. For example, if it repeatedly queries similar prompts, caching prevents recomputation and speeds up retrieval. Additionally, if using an external service for embeddings, caching saves on API calls. There might also be a periodic job to re-embed or fine-tune the embeddings if the model is updated (ensuring consistency across the vector space). Each memory item could also be versioned or timestamped.

- **Memory Consolidation & Version Control:** Over time, the memory will grow. Atlas should periodically prune or consolidate it. For instance, if 10 trade lessons all say similar things about breakout trades, an offline process might compress them into one summary and remove redundancies. This keeps the memory relevant and not too noisy. Version control wise, one can keep an archive of old memories (or mark them as deprecated) especially if the strategy changes. For instance, if Atlas abandons a certain approach, memories tied to that might be less relevant; they can be retained but given a lower priority in searches or filtered out. Tools like a simple metadata flag (active/inactive) or even moving old vectors to a secondary index can achieve this.

- **LLM Prompting with Memory:** The way memory is fed to the LLM is important. Typically, Atlas will do a semantic search in the vector DB with a query derived from the current context (like `f"{symbol} {strategy} past outcomes"`), retrieve the top K results (say 3-5), and then construct a prompt that includes them. For example:

Prompt: "You are a trading assistant. Here are relevant past notes: [Memory 1: …] [Memory 2: …]. Using these, analyze the new signal: XYZ… Provide your plan."

The LLM then has those notes as if someone reminded it of them, leading to a more informed output. The citations or references in the output can even be tied back to memory entries for transparency (though in practice, we might not need the model to output citations, just to use the info).

- **Benefits of LLM+Memory:** This combination allows Atlas to adapt. If the market regime changes or a new type of event occurs, you can *teach* Atlas by adding new memory (rather than retraining the entire LLM). It's like giving it new chapters in a continually growing playbook. The LLM brings the ability to generalize and reason, while the memory brings specificity and recency. Together, Atlas's AI component becomes **context-aware** and **evolving**. Essentially, the system "remembers" previous trades and advice, giving a sense of cumulative learning [26].

- **Challenges:** It's worth noting memory isn't a silver bullet. The vector search might pull slightly irrelevant info at times (semantic similarity isn't perfect), so Atlas might sometimes retrieve a memory that isn't actually useful [27]. We mitigate that by storing metadata (so we can filter by same symbol or sector to increase relevance, using hybrid search). Also, too much memory could overwhelm the prompt token limit, so we keep only top relevant snippets. Another aspect is that the LLM might incorporate a memory snippet incorrectly or over-generalize from it. To handle this, Atlas might give the LLM instructions like "Use the following notes if relevant, otherwise ignore." In critical scenarios, one could even have the LLM justify how the memory applies, as a check.

In summary, the LLM+Memory integration turns Atlas into a system that **learns and adapts**. It's not stuck with a static strategy – it can improve with experience, remembering what it learned yesterday to make a better decision today. By leveraging retrieval-augmented generation, Atlas's AI reasoning stays up-to-date and becomes richer over time, which is crucial in the ever-changing landscape of trading.

Deployment Roadmap

Deploying Atlas from a local prototype to a reliable cloud-based service requires careful planning. Below is a roadmap for migrating and hardening the system for production:

1. **Local Development & Testing:** Begin by running Atlas on a local machine or a development server. Use a paper trading account and sandbox APIs (e.g., Alpaca's paper trading, or Zerodha's test environment) to validate end-to-end functionality. Ensure logs are written (to console or local files) and monitor for any crashes or exceptions. At this stage, fast iterations are possible – refine the code, add more error handling, and write unit tests for critical components (e.g., risk calculations, broker order formatting).

2. **Containerization:** Once stable locally, containerize the application using Docker. Write a Dockerfile that sets up Python, installs dependencies, and copies the Atlas code. This ensures consistency across environments. Inside the container, make sure to include any system-level dependencies (for example, if using TA-Lib or specific libraries that need C extensions). By containerizing, you also make it easier to deploy on cloud providers or Kubernetes. Test the Docker image locally (or in a staging environment) with the paper trading mode to confirm everything works the same inside a container.

3. **Cloud VM Deployment:** Choose a cloud provider (AWS, GCP, Azure, etc.) and spin up a VM or small instance. This can be as simple as an EC2 VM on AWS or a Droplet on DigitalOcean. Transfer the Docker image or use Docker Hub to pull it on the VM. Ensure environment variables for API keys and secrets are set securely on the VM (never store secrets in the image; instead use cloud secrets manager or environment injection). Run the Atlas container, initially pointing to paper trading endpoints. Monitor resource usage (CPU, memory) to size the VM appropriately. Also configure the timezone/region on the server consistent with the market if needed (or handle in code).

4. **Database & State Management:** If using a database (for memory or logs), decide on managed services vs local. For instance, you might use a managed PostgreSQL for structured data, or a hosted vector DB (like Pinecone cloud) for embeddings. Ensure connectivity from your VM to these services is secure (VPC, API keys). For Google Sheets, the API calls will work from cloud just like local as long as credentials are present. Set up proper **credentials on the cloud** (e.g., store the Google API service account JSON securely on the VM or use a secrets manager). At this point, also set up backups or persistence for any critical data (like trade logs in DB).

5. **Observability:** Implement logging and monitoring solutions. Use a structured logging framework to send logs to a file or stdout (which cloud platforms can aggregate). For serious deployments, integrate with monitoring tools: e.g., CloudWatch (AWS) or Stackdriver (GCP) to track metrics. Key metrics to monitor: uptime of the process, number of trades executed, P&L, as well as system metrics (CPU, memory, network). Set up alerts for exceptional situations: e.g., if the process stops or if P&L drops beyond a daily threshold (risk limit breach). Observability also means being able to see what the AI is "thinking" – consider logging the LLM's rationale outputs to a separate log for later analysis. This is invaluable for debugging decisions.

6. **Safety & Kill-Switches:** In live trading, you need the ability to halt the system quickly if something goes wrong. Implement a kill-switch mechanism: for example, Atlas could periodically poll a particular file or database value "trading_enabled". If set to false, Atlas will not initiate new trades (and possibly close out all positions). You could toggle this via a simple UI or command. Additionally, use broker-provided features like account stop-loss or max drawdown limits if available. For instance, some brokers allow setting "max loss" on an account level; if not, Atlas's monitoring should enforce it (e.g., if cumulative losses for the day > X, then set trading_enabled false). Make sure this logic is tested in paper mode so it indeed stops trading under those conditions.

7. **Scaling & Message Bus:** As Atlas grows, consider a more distributed architecture. For instance, running signal agents on separate processes or machines, and using a message bus (like Redis pub/sub, RabbitMQ, or Kafka) to send signals to the orchestrator. This decouples components and improves scalability and fault tolerance. The orchestrator can run on one machine, while data ingestion or heavy computations (like indicator calculations or NLP on news) run elsewhere, communicating via messages. Apache Kafka could even be used to handle real-time data streams and events [28], which Atlas agents subscribe to. While this may not be needed initially, designing with modularity in mind will ease this transition later.

8. **Cloud Migration & Keys:** As you move fully to cloud, ensure all API keys are handled securely. Use services like AWS Secrets Manager or GCP Secret Manager to store sensitive keys and retrieve them in your app at runtime. Rotate keys periodically as a best practice. Also, manage access control – if using Git or CI/CD, never expose keys in configs. Utilize IAM roles for access where possible (e.g., if

running on AWS and needing to access certain AWS services, use an IAM role attached to the instance rather than static keys).

9. **Docker Orchestration / Kubernetes:** For high availability, consider using Kubernetes or Docker Compose in the cloud to manage the Atlas components. Kubernetes can ensure the Atlas service is always running (by restarting pods on failure, etc.), and you can set resource limits. It also makes scaling out simpler (if you needed multiple instances of Atlas for different strategies, etc.). If that's overkill initially, a simpler approach is using a process manager (like systemd or PM2 for Python) on the VM to auto-restart the app if it crashes.

10. **Rollout to Live Trading:** After extensive paper trading validation and perhaps a small-scale live test (with minimal capital), you can transition to full live trading. This means switching the broker endpoints to live trading (make sure `paper_trading=False` or equivalent in config). Perhaps do this during a calm market period and monitor like a hawk initially. Keep a manual override ready (the kill-switch, or even the ability to hit the broker's big red button to close positions manually). Gradually increase capital allocation as confidence grows.

11. **Continuous Learning & RL (Future):** Looking beyond initial deployment, you might incorporate an **RL training loop** in parallel. For instance, collecting data from live trading to periodically retrain a model (maybe using FinRL or a custom RL approach) on recent history. This could be offline (not interfering with live trades) and once in a while propose updated strategy parameters or policies. The roadmap could include a phase where the RL agent's suggestions are paper-tested alongside the rule-based approach, and only merged into the live strategy upon proven improvement. Essentially, an *experimentation pipeline* in the cloud can be set up: live trading on main, and a shadow instance that paper-trades new strategies or parameters. With Atlas's modular design, you could even A/B test strategies by splitting capital or running two versions concurrently on small allocation, then choosing the best.

12. **Security & Compliance:** As a final note on deployment – trading systems need to be secure (financial APIs and keys are sensitive). Use HTTPS everywhere, don't expose your services unnecessarily (e.g., if not needed, keep ports closed, or behind VPNs). Also, ensure compliance with any trading regulations (for example, some jurisdictions require certain disclosures or forbid fully automated trading on retail accounts). Keep an eye on broker terms of service as well, to ensure Atlas's operation (especially with AI decision-making) doesn't violate any usage policies.

By following this roadmap, you can transition Atlas from a development project to a **production-grade trading system** that is robust, observable, and controllable. Each step reduces risk: e.g., containerization removes "it works on my machine" issues, observability catches problems early, and kill-switches limit damage from unforeseen events. Running on the cloud with proper setup also allows Atlas to operate 24/7 (useful if you extend to crypto markets which never close, or if running strategies across timezones). With the deployment in place, Atlas becomes a live platform that can be continuously improved and scaled.

Strategic Add-ons

With the core system in place, there are several advanced features and extensions that can enhance Atlas's intelligence and adaptability. These "strategic add-ons" are designed to give Atlas a cutting-edge over basic algorithmic trading systems:

- **Self-Critique Agent:** This is essentially an introspective layer for Atlas. After each trade (or each day), a dedicated agent (which can be an LLM prompt or a separate routine) reviews the trade decisions and outcomes. It uses a **reflection technique** – prompting the LLM to analyze its own actions and suggest improvements [18]. For example, it might ask, "Why did the last trade hit the stop-loss? Was the entry timing wrong, or was it just bad luck?" The LLM, given the trade data, might respond with something like, "The entry was just before a major news announcement, which increased volatility. Next time, avoid trading before big news." This feedback can then be turned into updated rules or memory notes. Over time, the self-critique agent helps refine Atlas's strategy and even the LLM's future responses (since these critiques become part of the memory). It's like having an independent risk manager or coach that constantly evaluates the AI's performance. Research in AI agents shows that reflecting on past actions can significantly improve future decisions [18]. Implementing this could be as simple as an end-of-day job that goes through trades and asks the LLM for each, "What could have been done better?"

- **Dynamic Long/Short Allocation:** Atlas can be extended to manage not just individual trades, but also **portfolio-level strategy** in terms of long-term vs short-term allocation. For instance, Atlas could allocate a portion of capital to longer-term investments (LT) and another portion to short-term trading (ST). The ratio of this allocation can be dynamically adjusted by an agent based on market conditions. In a trending bull market, the system might allocate more to long-term holds (to ride the trend), whereas in a sideways or mean-reverting market, it might allocate more to short-term trades to capitalize on swings. This could be driven by metrics or an LLM assessment of regime ("Current regime: volatile and range-bound – shift 20% more capital to short-term mean reversion strategies."). The dynamic allocation ensures that Atlas is balancing between "investment" and "trading" strategies optimally. It's essentially an internal hedge – the long-term component might be a steady ETF or a set of quality stocks, and the short-term is the active trading. This add-on requires portfolio oversight logic that monitors performance of each segment and rebalances. It transforms Atlas from just a trading bot into a quasi-fund manager that decides how much to deploy in different strategies on the fly.

- **News/Event Anomaly Filter:** Markets are often shaken by news events (earnings releases, economic data, geopolitical events). This module acts as a **guardian that watches for anomalies** and news, and can override or adapt Atlas's behavior. It could work in a few ways:

- *News Sentinel:* Integrate a news API (or even social media feed) to get alerts on significant events (e.g., central bank rate decision, war outbreak, company CEO resigning). The LLM or a rule-based system classifies the news as high, medium, low impact for Atlas's positions or watchlist. If high impact, Atlas might pause trading or tighten stops around that period. For example, if Apple's earnings call is in an hour, the system might avoid opening new tech stock positions or hedge existing ones.
- *Volatility Anomaly Detector:* Use statistical techniques to detect unusual price movements or volume spikes that were not preceded by signals. If an anomaly is detected (say a stock drops 5% in a minute

out of nowhere), the system can check news instantly for that symbol. If news confirms (e.g., a sudden SEC announcement on that company), Atlas can take appropriate action (maybe exit if it has a position, or avoid entering a new trade that a signal might otherwise falsely indicate as a "dip buy"). If no news, it might treat it as a possible data error or just a peculiar spike and tread carefully.

- *Adaptive Risk Modes:* During known event windows (like Fed announcement days, major economic reports), the risk agent could automatically reduce the per-trade risk from 2% to 1% or adjust other parameters. This can be triggered by a calendar or news feed. Many traders avoid trading during such events – Atlas could do the same or switch to a special strategy optimized for event volatility (like straddles options strategy if equipped).

Essentially, this add-on ensures Atlas is **event-aware** and can step out of the way when a freight train of news is coming, or capitalize on it if that's part of the strategy. It blends fundamental/event-driven thinking with the technical system. As noted, an advanced news filter helps bots account for news impact on decisions [29] – bridging the gap between purely technical models and real-world events.

- **Multi-Agent Collaboration:** In the future, Atlas could spawn additional specialized agents – for example, a **Regime Detection Agent** that analyzes macro data to determine if we are in a bull, bear, or sideways market (and then informs other agents to adjust their thresholds or which strategies to prioritize). Or a **Capital Allocation Agent** that decides how to distribute capital among multiple strategy agents (if Atlas runs, say, a trend-following strategy alongside a mean-reversion strategy). This moves towards a **hierarchical multi-agent system**, where a top-level agent orchestrates multiple strategy agents – a bit like having multiple Atlas instances focused on different approaches, and a meta-agent choosing between them.

- **Reinforcement Learning Extension:** Tied to the deployment roadmap, one strategic add-on is to integrate an **RL training loop** more directly. Atlas could run an RL agent in parallel that observes Atlas's trades and learns to suggest adjustments. Over time, this RL agent might replace certain rule-based components if it proves better. For instance, instead of a fixed 2% risk, an RL agent might learn to vary risk per trade slightly (maybe 1% on lower confidence trades vs 2.5% on very high confidence trades), within safe boundaries. This is speculative, but a path where Atlas becomes increasingly autonomous in learning from data, not just following preset rules.

- **User Interaction Interface:** While not exactly an agent, a strategic improvement is giving Atlas a UI or chatbot interface where a human trader can query it. For example, an operator could ask, "Hey Atlas, why did you not trade EURUSD this week?" and the LLM (with context from memory) can answer, "Because volatility was high due to ECB meeting, which didn't meet my risk criteria." This makes the system more transparent and user-friendly. One could even *ask* Atlas for advice on a trade outside its normal operation, leveraging its analysis capabilities.

By implementing these strategic add-ons, Atlas can evolve into an even smarter and safer trading system. The **self-critique** ensures continuous improvement (a form of meta-learning), the **dynamic allocation** ensures better capital efficiency across strategies, and the **news/anomaly filter** protects against regime shifts and black swan events. Combined, these features aim to make Atlas not just an automated trader, but a **holistic AI trading platform** that adapts like a seasoned trader with experience, caution, and strategic foresight.

---

1 Python quickstart | Google Sheets
https://developers.google.com/workspace/sheets/api/quickstart/python

2 Examples of gspread Usage — gspread 6.1.2 documentation
https://docs.gspread.org/en/latest/user-guide.html

3 4 The 2% Rule - CME Group
https://www.cmegroup.com/education/courses/trade-and-risk-management/the-2-percent-rule.html

5 The Truth About the 3:1 Risk-Reward Ratio Rule | EBC Financial Group
https://www.ebc.com/forex/the-truth-about-the--risk-reward-ratio-rule

6 17 19 24 How to build a ChatGPT-powered AI trading bot: A step-by-step guide
https://cointelegraph.com/news/how-to-build-a-chatgpt-powered-ai-trading-bot-a-step-by-step-guide

7 11 LLM Agent Orchestration: A Step by Step Guide | IBM
https://www.ibm.com/think/tutorials/llm-agent-orchestration-with-langchain-and-granite

8 10 14 AI-Powered Multi-Agent Trading Workflow | by Bijit Ghosh | Medium
https://medium.com/@bijit211987/ai-powered-multi-agent-trading-workflow-90722a2ada3b

9 25 Memory Buffer as Vector Database in Autonomous Agents | by Vic Genin | Medium | Medium
https://deeprnd.medium.com/memory-buffer-as-vector-database-in-autonomous-agents-021e5024a3eb

12 What is Retrieval-Augmented Generation (RAG)? | Google Cloud
https://cloud.google.com/use-cases/retrieval-augmented-generation

13 26 27 How AI Agents Remember Things: The Role of Vector Stores in LLM Memory
https://www.freecodecamp.org/news/how-ai-agents-remember-things-vector-stores-in-llm-memory/

15 16 The 23% Solution: Why Running Redundant LLMs Is Actually Smart in Production : r/OpenAI
https://www.reddit.com/r/OpenAI/comments/1l70htm/the_23_solution_why_running_redundant_llms_is/

18 Reflection Agents
https://blog.langchain.com/reflection-agents/

20 21 Battle-Tested Backtesters: Comparing VectorBT, Zipline, and Backtrader for Financial Strategy Development | by Trading Dude | Jun, 2025 | Medium
https://medium.com/@trading.dude/battle-tested-backtesters-comparing-vectorbt-zipline-and-backtrader-for-financial-strategy-dee33d33a9e0

22 Awesome Quant
https://wilsonfreitas.github.io/awesome-quant/

23 GitHub - AI4Finance-Foundation/FinRL: FinRL®: Financial Reinforcement Learning.
https://github.com/AI4Finance-Foundation/FinRL

28 Apache Kafka + Vector Database + LLM = Real-Time GenAI
https://www.kai-waehner.de/blog/2023/11/08/apache-kafka-flink-vector-database-llm-real-time-genai/

29 Advanced News Filter in trading bot - Forex insights
https://forexinsights.hashnode.dev/advanced-news-filter-in-trading-bot