ChatGPT

# SenseQuant Development Plan Integrating BMAD-METHOD (v4 Stable)

## Overview and Goals

SenseQuant is an AI-powered Indian stock trading assistant designed for both intraday and swing trading. It incorporates advanced AI components like a teacher–student learning architecture, sentiment analysis, multi-agent strategy execution, and integrates with ICICI Direct's Breeze API for market data and trade execution. The goal is to **integrate the BMAD-METHOD framework** (v4 stable) as the orchestration layer for development, using it to structure the multi-agent workflow while leveraging powerful LLMs (Claude or ChatGPT) for each agent role. BMAD will provide the agile workflow (roles, story files, planning → development cycle) but **Claude/ChatGPT will act as the brains** for each role instead of BMAD's default stub agents. This integration aims to preserve all of SenseQuant's domain-specific logic (the custom AI modules and trading strategies) while imposing an organized, context-rich development process.

**Why BMAD?** The BMAD-METHOD (Breakthrough Method for Agile AI-Driven Development) introduces a clear separation of roles (Analyst, Architect, Developer, QA, Project Manager, etc.) for AI agents and a story-centric workflow to prevent context loss [1] [2]. In a normal single-chat development, an AI might forget details or mix roles when the conversation gets long. BMAD solves this by maintaining **dedicated personas with persistent instructions** and by passing detailed "story" documents between phases [1] [3]. We will exploit this structured approach to ensure each phase of SenseQuant's development is handled by the right AI agent with full context and minimal confusion. The end result will be a step-by-step development plan that follows BMAD's two-phase workflow (Planning, then Context-Engineered Development) [4], with **Claude/ChatGPT filling in as Analyst, Architect, Developer, etc., within VS Code or a web chat interface** as preferred.

## Project Setup and BMAD Configuration

**1. Install and Configure BMAD:** Begin by installing BMAD-METHOD v4 into the SenseQuant project repository. This can be done via the npm package, e.g. `npx bmad-method install`, which will set up the necessary BMAD files and folder structure [5] [6]. After installation, the project will contain a `.bmad-core/` directory with agent definitions and utilities, as well as a `bmad/web-bundles/` directory containing ready-to-use agent and team persona files for web usage [7] [8].

**2. Documentation Structure:** Create a docs folder for BMAD to use in planning and development. BMAD expects certain documents where agents will read/write outputs [9]. Set up the following: - `docs/prd.md` – **Product Requirements Document** where the product vision and requirements (features, use cases) will be documented. - `docs/architecture.md` – **System Architecture Document** for high-level design, module breakdown, and technical decisions. - `docs/stories/` – a folder to hold story files (each user story or feature implementation plan will be a separate file here) [10]. - (Optional) `docs/backlog.md` or similar – could be used to list the product backlog (user story list), unless the backlog is directly broken into individual story files by the PM agent.

By structuring these upfront, we give the BMAD agents a place to record their outputs. For example, the Analyst/PM agents will write to the PRD or story files, the Architect to the architecture doc, etc., which subsequent agents (and the human) can read from.

**3. VS Code and Agent Environment:** In VS Code, set up integration for Claude or ChatGPT agents: - If using **GitHub Copilot Chat** (with GPT-4): Leverage the `.github/chatmodes/` or similar mechanism to create custom chat profiles for each agent role [11]. BMAD's install can generate `.chatmode.md` files for each agent (Analyst, Architect, Dev, QA, etc.), which include the role instructions and available tools [12] [11]. In VS Code, these can appear in the Copilot chat sidebar, allowing one to switch personas easily. - If using **Claude Code** (Anthropic's VS Code plugin or JetBrains plugin): Define slash-command triggers for each agent. For example, create commands like `/pm`, `/sm`, `/dev`, `/qa` that load the corresponding BMAD agent prompt. The BMAD + Claude guide suggests making command files wrapping each core agent under a `commands/bmad/` directory, then using them in the Claude chat interface [13] [14]. This allows you to type `/dev` to activate the Developer agent persona, etc. - If using **Web UI (Claude.ai or ChatGPT web)**: Use BMAD's **team bundle** approach. BMAD provides team definition files (e.g., `bmad/web-bundles/teams/team-fullstack.txt` for a full team) which contain all role definitions in one. You can copy-paste this entire file into a new chat with Claude or ChatGPT to initialize the personas [15]. After that, use the special directives to switch context to a specific agent. For example, you might paste the team file, then start issuing commands like `*agent analyst` or `*agent architect` to let the model know which role to assume for the next instruction [16] [17]. This method allows a single chat to handle multiple roles by explicitly signaling role switches. (Ensure to begin each command with the `*agent ...` prefix as specified by BMAD's syntax.)

**4. Configure Agent Context Sharing:** To maximize context sharing in VS Code, update BMAD's config so that key documents are always loaded for relevant agents. For instance, in `.bmad-core/core-config.yaml`, you can specify that the Developer agent always loads the PRD and architecture docs [18]. This guarantees that when Claude/ChatGPT is acting as the Developer, it has the latest requirements and design context on hand (in addition to the specific story file). Similarly, the QA agent could always load the PRD or the relevant story/test plan. This step ensures *Claude/ChatGPT integration is context-rich*, leveraging BMAD's ability to inject files into the prompt so the model isn't flying blind.

With the environment ready, we proceed to the two main phases of BMAD: **Planning** and **Development**.

## Phase 1: Planning with BMAD (Analyst, PM, Architect roles)

This phase focuses on defining *what* to build and *how* to build it, without writing actual code yet. In BMAD, dedicated planning agents work with the human to produce a **Product Requirements Document (PRD)** and an **Architecture design**, which together guide the development. We will use Claude/ChatGPT to fill the shoes of BMAD's **Analyst, Product Manager, and Architect** roles here. The output of this phase will be a detailed plan (stories, specs, design) that feeds into the development phase.

### 1. Requirements Analysis and PRD Creation (Analyst Agent)

**Role:** The Analyst agent (Claude/ChatGPT in an Analyst persona) will gather and formalize the requirements for SenseQuant. The user (you) provides the initial vision and goals: an AI trading assistant supporting intraday and swing trading for Indian markets, leveraging sentiment analysis and a teacher–student AI strategy model, integrated with ICICI's trading API. The Analyst's job is to ask clarifying questions if needed and produce a comprehensive **Product Requirements Document**.

**Process:** Initiate a chat with the Analyst agent. Provide all relevant background: trading objectives, constraints (e.g., free cloud hosting environment), and domain context (ICICI Breeze API availability, etc.). Then request a structured PRD. For example, you might prompt:

> "*As the Analyst, please draft a Product Requirements Document for the SenseQuant trading assistant. Include an overview, key features (intraday trading, swing trading, sentiment analysis integration, teacher-student AI loop), user scenarios (e.g., how a user might use this assistant), and any assumptions (like having an ICICI Direct account).*"

The Analyst agent, guided by its persona definition, will produce a detailed PRD. It should capture: - **Intraday Trading Features:** Define what intraday trading means for this system (e.g., buying and selling within the same day's market session). It can mention that positions must be closed by end of day to avoid overnight risk [19], and that the assistant should perhaps optimize for quick profits and manage multiple trades in a day. - **Swing Trading Features:** Define swing trading support (holding positions for multiple days to weeks). Clarify that the system can carry positions overnight and aim for larger price swings [19]. This might involve different strategy parameters (longer-term indicators, lower frequency of trades). - **Strategy and AI Components:** Explain the **teacher–student learning architecture** in simple terms – e.g., a "Teacher" AI module that evaluates performance or provides high-level strategy guidance, and a "Student" AI module that makes day-to-day trading decisions. The PRD should state why this is used (perhaps the teacher helps improve the student model over time by learning from mistakes). - **Sentiment Analysis Integration:** Outline that the assistant will incorporate sentiment data (from news, social media, etc.) to improve decision-making. For instance, it might use sentiment to avoid trades on bad news or capitalize on positive sentiment trends. The PRD doesn't need to detail the NLP model, but it should note this feature and its value (e.g., "incorporates market sentiment analysis to filter or validate trade signals"). - **Data Sources & APIs:** Note that real-time and historical market data will come from the **ICICI Direct Breeze API**, and that the system can place trades through this API. Include any known constraints (like API rate limits or the need for API keys). The PRD might specify basic requirements like *"The system shall fetch live stock prices and place orders through Breeze API"*. (ICICI's Breeze API allows real-time order execution and provides up to 10 years of historical data for backtesting [20] – this is valuable context to include). - **User Interface & Alerts:** If the user expects any interface (perhaps a simple CLI, or email/Telegram alerts for recommended trades), capture that. Since free hosting is desired, maybe the UI is minimal, but the PRD can say how the user interacts (e.g., configuration via config files, output logs, or a simple dashboard). - **Risk Management:** Requirements around risk limits (stop-loss, profit targets, position sizing) for both intraday and swing trades. - **Hosting & Deployment Constraints:** Mention that the solution will be deployed on a free cloud service, which might impose limits (memory, compute), so the design should be lightweight (for example, use efficient Python libraries, possibly avoid heavy databases unless necessary). - **Success Criteria:** Define what a successful trading assistant should achieve (not necessarily profit guarantees, but e.g. able to execute trades reliably, provide useful insights, respond in real-time during market hours, etc.).

The PRD should be stored in `docs/prd.md`. This document will serve as the single source of truth for what the system must do, and will be referenced throughout development. Human review is important here: ensure the PRD reflects the user's intentions accurately and comprehensively. You can iterate with the Analyst agent – BMAD encourages human-in-the-loop refinement of specifications [4]. For example, if something is missing (say, handling of trading holidays or specific stock segments), you can prompt the Analyst to add it. The result is a high-quality PRD that "goes far beyond generic AI task generation" [4], capturing domain-specific needs.

## 2. System Architecture Design (Architect Agent)

**Role:** The Architect agent (Claude/ChatGPT in an Architect persona) will devise the technical solution for the requirements laid out in the PRD. This means identifying the components, their interactions, and the technologies to be used, ensuring that the custom AI modules (teacher-student, sentiment) are included as specified (not replaced by something else).

**Process:** Once the PRD is ready, engage the Architect agent. Provide it the final `docs/prd.md` (you can copy relevant sections or the entire document if the context window allows) and ask it to produce an **Architecture Document (`docs/architecture.md`)**. Prompt example:

> "*As the Architect, please design the system architecture for the SenseQuant assistant based on the PRD. Identify all major components/modules, how they interact, and any design patterns or tech stack decisions. Ensure to include the AI components (teacher-student strategy module, sentiment analysis) and the Breeze API integration in the design.*"

The Architect will output a structured design. Key elements to expect and refine: - **Module Breakdown:** We should see modules like: - **Data Ingestion Module:** handles connection to Breeze API – fetching live data (during market hours) and historical data (for backtesting or overnight analysis). The design might mention using the Breeze **Python SDK** for easier integration [20] (since it's the official client library). - **Strategy Engine:** the core of trading decisions. Within this, specify sub-components for *intraday strategy* and *swing strategy*. Possibly, each could be an agent or process with its own logic, since their operation differs in timeframe. - **Teacher–Student Learner:** This could be modeled as two components: - *Student Trader*: makes trade decisions (could be an AI model or rule-based agent that looks at indicators, signals, sentiment, etc.). - *Teacher Critic*: evaluates the student's decisions and outcomes. The architecture should describe how the teacher provides feedback – e.g., after market close, the teacher reviews trades made by the student, calculates performance metrics, and adjusts the student model or parameters for next time. This forms a feedback loop (maybe implemented via periodic training sessions, or a rule-tuning mechanism). - The architecture should clarify whether this loop runs in real-time (likely not; more so after each trade or end of day) or in simulation for training. - It's crucial the Architect does **not** eliminate this loop. If the AI architect agent, for instance, tries to simplify by using only one agent, intervene and stress that maintaining the teacher-student design is a requirement. The BMAD Architect persona should normally honor what's in PRD, but it can be guided with reminders like "keep the learning loop architecture". - **Sentiment Analyzer:** a component that retrieves news or social media data and produces a sentiment score for relevant stocks. The architecture might say this module runs in parallel with the strategy engine – e.g., at market open it fetches latest news sentiment for each stock and the strategy engine uses those scores as inputs. It should define what tech to use (maybe a Python NLP library or a pre-trained model for sentiment – possibly the user can fine-tune later, but initial version could use something off-the-shelf). - **Execution Module:** places orders via Breeze API. It should encapsulate functions like `place_order(stock, quantity, action)` and handle order confirmations or errors. Also possibly handle **portfolio management** (checking current holdings, etc., which Breeze API supports [20]). - **Scheduler/Orchestrator:** since we have different activities (intraday trading needs a loop during market hours; swing trading maybe runs on daily signals; teacher feedback might run after market close; sentiment fetch could run daily or whenever), the architecture should include a scheduling mechanism. This could be as simple as a Python schedule (cron jobs on cloud) or a loop that runs during market hours for intraday signals and another that triggers once a day for swing decisions and teacher evaluation. Given free hosting, a lightweight scheduler (like APScheduler or cron on a free Linux instance) could be used. - **Data Storage** (if any): Decide if the system will store data, e.g., saving historical prices fetched, saving model parameters for the student model, logging trades, etc. Possibly use a small database or just flat files/CSV for simplicity due to free hosting constraints. - **Interface**: If an interface/alert system is in scope, mention how it's delivered (e.g., an email alert via

SMTP, or just console logs). Many trading bots use Telegram or email for notifications of trades; the architecture can mention this if desired by user. - **Interactions:** Draw out how data flows. For example: 1. Market data comes in via Breeze API (streaming websockets for live ticks or polling REST endpoints). 2. The intraday module consumes this data continuously, decides trades (perhaps with sub-second or minutely frequency, which might be challenging on free hosting – but architecture can propose either a real-time streaming approach or periodic checks). 3. The swing module might run on daily candle data (end-of-day prices). 4. Both modules consult sentiment data (which could be updated daily or intraday if news breaks). 5. When a trade signal is generated, it goes to the Execution module to place the order via Breeze. 6. All trades and outcomes are logged. After each day, the Teacher module reviews the intraday trades (and perhaps weekly the swing trades) to adjust strategy. - **Tech Stack Decisions:** The Architect should recommend using **Python** (likely, given Breeze SDK and AI libraries). It might specify using libraries/frameworks for: - Data: `breeze-connect` for Breeze API [21], possibly `pandas` for data handling, technical analysis libraries (like TA-Lib or pandas-ta) for indicators. - AI/ML: maybe TensorFlow/PyTorch if the student is a learning model, or simpler if rule-based. If sentiment analysis uses an NLP model, could use HuggingFace Transformers (if not too heavy) or a lightweight sentiment API. - The design might also discuss how to manage API keys and config (likely through config files or environment variables). - Testing: mention using a testing framework (unittest or pytest) for the modules, which QA agent will later utilize. - **Scalability & Constraints:** Acknowledge that because it's free hosting, components need to be efficient. For instance, avoid extremely data heavy operations or ensure the code can run within memory/CPU limits of a free tier. Perhaps the intraday loop might not handle extremely high-frequency trading (e.g., probably it will check every few seconds or minute rather than tick-by-tick if on a free tier, unless using Breeze's streaming which might be feasible with a persistent connection). - **Security:** Since this involves financial API, ensure the architecture notes how sensitive info (API secret, token) is stored (probably not in code – maybe in a secure config file not in repo, or as environment variables on the host).

The Architect agent will document these in `docs/architecture.md`, possibly including diagrams (which you can have it describe in text or simple ASCII if needed). Review this carefully as well; it's the blueprint the Developer will follow. If something is unclear or you want to enforce a particular approach (e.g., use reinforcement learning for the teacher-student loop), edit the architecture or prompt the agent to adjust. The architecture doc should align with the PRD – every requirement in PRD should be addressed by some part of the design. This ensures BMAD's next steps (story generation) are consistent with the intended system.

## 3. Backlog and Story Planning (Project Manager/PM Agent)

**Role:** The Product Manager (or Project Manager) agent will translate the PRD and architecture into an actionable **project backlog**. This means creating user stories or epics, prioritizing them, and preparing for the iterative development cycle. In BMAD's flow, the PM agent (sometimes called PO – Product Owner – agent) works with the requirements to produce a list of stories, and then the Scrum Master (next step) will take the top stories into development. We'll use the PM agent with Claude/ChatGPT to ensure we have a clear work breakdown aligned with BMAD's workflow.

**Process:** Now that PRD and architecture are in place, invoke the PM agent. Provide it with access to `docs/prd.md` (and possibly `docs/architecture.md` for technical context). Ask it to generate a **prioritized backlog of features**, broken down into epics and stories. For example:

> "*As the Product Manager, please create a product backlog for the SenseQuant project. Break the project into epics and user stories based on the PRD (and architecture), prioritize them (considering we likely want a working trading core first), and list acceptance criteria for each story where applicable.*"

The PM agent will likely create something like: - **Epic 1: Core Trading Engine** – might encompass basic trading logic. - Story 1.1: Implement Breeze API integration (connect, authenticate, simple data fetch and order placement) – **[Highest Priority]** because without this, no trading can occur. - Story 1.2: Intraday Trading Strategy module (basic version) – generating buy/sell signals on intraday data. - Story 1.3: Swing Trading Strategy module (basic version). - Story 1.4: Trade Execution and Order Management (using the signals from 1.2/1.3 and the Breeze API). - Story 1.5: Risk Management features (stop-loss, position sizing logic). - **Epic 2: AI Enhancement** – incorporate AI/ML elements. - Story 2.1: Integrate Sentiment Analysis into decisions (fetch news, compute sentiment score). - Story 2.2: Teacher-Student Feedback Loop – implement the mechanism to adjust strategy based on performance (could be split into: 2.2a: tracking performance metrics, 2.2b: adjusting strategy rules or model). - Story 2.3: Backtesting Utility (optional but useful for testing strategies on historical data). - **Epic 3: User Interface & Alerts** – how the user interacts/monitors. - Story 3.1: Logging and Alert System (e.g., console logs, file logs of trades, email alerts for each trade). - Story 3.2: Simple UI or CLI commands to control the bot (start/stop, switch between paper trading and live). - **Epic 4: Testing & Deployment** – final stage tasks. - Story 4.1: Write end-to-end tests for strategies (simulate various scenarios). - Story 4.2: Deployment setup on free cloud (Dockerfile or scripts for deployment). - Story 4.3: Documentation & User Guide (so a user knows how to run this, configure API keys, etc.).

The above is an illustrative breakdown. The PM agent will produce something in this vein, and importantly, it should **assign priorities**. Likely, core integration (Epic 1) will be highest priority, because without a working trading pipeline, other features can't be demonstrated. The PM might plan a first iteration (say a 1-week sprint) focusing on the most critical stories: Breeze API integration, a basic intraday strategy, and execution flow – enough to place a trade. Then subsequent iterations add sophistication (swing trading, sentiment, learning loop, etc.). BMAD often has the PM agent create a **prioritized list of user stories with IDs** and maybe labels like `us-001`, `us-002`, etc., stored either in individual files or one backlog file [22]. For example, BMAD's example command for PM is to read the PRD and *"produce a prioritized backlog (epics → stories) for a 1-week iteration. Save stories in docs/stories/."* [22]. This means each story might directly get its own file in `docs/stories/` with a unique name (like `docs/stories/us-001.md` for Breeze API integration, etc.), containing the story title and acceptance criteria.

Ensure the **acceptance criteria (AC)** for each story are noted, as these will be used by QA later. For instance, AC for "Implement Breeze API integration" could include: *"Able to fetch current market price for a given stock symbol using Breeze API,"* and *"Able to place a test order (in a paper trading mode or small quantity) successfully through Breeze API."* Acceptance criteria for "Intraday Strategy module" might include conditions like *"Generates buy/sell signal at least once a day under certain volatility conditions," "Closes all open positions by market close (3:30 PM IST) on the same day"*, etc. These AC will later guide testing.

After this step, we should have: - A set of story files in `docs/stories/` (or a single backlog file listing them) that represent the **work breakdown**. - Each story with a unique ID, a description, and acceptance criteria.

**Preserve Custom Modules in Plan:** It's crucial that these stories reflect the custom AI aspects. For example, ensure there is a story explicitly for the teacher-student loop (not something generic like "implement machine learning improvement" which might get misinterpreted). Name it clearly and include the detail from PRD: e.g., *"Story: Implement Teacher-Student Learning Loop – Create a mechanism where a Teacher agent evaluates the trades made by the Student (trading algorithm) and adjusts its parameters or model weights to improve performance over time."* This way, when that story is picked up, the dev agent knows exactly what to build. Similarly, the sentiment integration story should specify what sources or API to use for sentiment (if decided in architecture, e.g., "use Twitter API or a news feed" or a placeholder). If these domain-specific features are well-represented in the backlog, BMAD's structure will **not override them** – instead, each agent will work to implement them as written.

At the end of Phase 1, you (and the AI agents) have produced a *specification and plan* that includes: - **PRD** – what the system should do (features and requirements) [4] . - **Architecture** – how the system will be structured to meet those requirements. - **Backlog/Stories** – the implementation plan broken into tasks with criteria.

This satisfies BMAD's planning phase outcome: *"detailed, consistent PRDs and architecture documents" (via Analyst/Architect), and a structured plan to feed the development cycle* [4] . Now, we move to Phase 2 to actually implement the system using the Dev and QA agents, following BMAD's story-centric development workflow.

## Phase 2: Iterative Development with BMAD (Scrum Master, Developer, QA roles)

Phase 2 is where code gets written and tested. BMAD's core development cycle uses a **Scrum Master (SM) agent** to orchestrate each user story, a **Developer agent** to write code, and a **QA agent** to verify the outputs [2] . This happens in iterative loops, typically story by story (or in small batches). Claude/ChatGPT will take on these roles in our process, guided by BMAD's story files. We aim to implement each story from the backlog while preserving the context (via story files) and maintaining quality through testing.

### 4. Orchestration & Task Handoff (Scrum Master Agent)

**Role:** The Scrum Master (SM) agent acts as the facilitator of the development sprint. In BMAD, the SM's key responsibility is to take a planned user story and **generate a detailed development plan for it**, including breaking it into tasks, providing the context from PRD/architecture, and listing acceptance criteria – all consolidated into a "story file" for the Developer agent [2] . Essentially, the SM agent is the glue between planning and coding: it ensures the Developer has everything needed and knows exactly what to do.

**Process:** For each top-priority story (from the backlog prepared earlier), do the following: - Open or identify the story file (e.g., `docs/stories/us-001.md`). It might already contain a short description from the PM step. Now use the SM agent to enrich it. - Prompt example: *"As the Scrum Master, take story us-001 (Breeze API integration) and prepare it for development. Include relevant details from the PRD and architecture that the Developer will need, enumerate the development tasks step-by-step, and list the acceptance criteria clearly. Once ready, mark it as ready for Dev."* - The SM agent will read the story description and fetch relevant context (the BMAD SM agent has access to the PRD and architecture docs in the configuration, or you may copy-paste key points). It will then output within that story file: - **Context Recap:** A section that summarizes the user story and any context. For Breeze API example, it might copy the PRD line "System must connect to Breeze API to fetch data and send orders" and architecture lines about using the BreezeConnect SDK [20] . - **Development Tasks:** A checklist or numbered list of what needs to be done. E.g.: 1. *Set up Breeze API client authentication (using API key, secret, session as per API docs).* 2. *Implement a function to fetch current market price for a given symbol (use Breeze API endpoint).* 3. *Implement a function to place a buy/sell order (with necessary parameters). If possible, use a sandbox or small quantity for safety.* 4. *Test the above functions with a sample symbol and record the output (print or log the result).* 5. *Ensure error handling for network issues or API errors (e.g., invalid token).* - **Acceptance Criteria:** Already defined in story, but SM will ensure they are clearly listed. For example: - *AC1: Able to retrieve the latest price of a specified stock via Breeze API.* - *AC2: Able to execute a* *trade (buy/sell) through Breeze API and receive confirmation. - AC3: API credentials and session management handled securely (no plaintext exposure in code logs).* - **Handoff Note:** The SM agent might add a note like "**Hand-off to Developer:** All set. Developer can proceed to implement above tasks. Reference: Breeze API documentation if needed [20]."

This story file now becomes a self-contained work order for the Developer agent. BMAD emphasizes that by embedding **full context and instructions in the story file, we eliminate context loss** between planning and coding [2]. The Developer doesn't need to scroll through chat history or recall earlier conversations – they open `us-001.md` and see everything relevant.

Repeat this SM agent preparation for each story in sequence. In practice, you might do one story at a time, especially if you're testing and integrating as you go. The SM can generate one story's tasks, then once that story is done (Dev + QA), move to the next top priority story.

## 5. Development Implementation (Developer Agent)

**Role:** The Developer agent is the AI that writes the actual code, configurations, and possibly documentation according to the task specifications. Claude or ChatGPT in Developer persona will use the story file content plus any always-loaded context (PRD, architecture) to produce code that fulfills the requirements and passes the acceptance criteria. This is where we ensure the AI does not deviate into something off-track – by constraining it with the well-defined story and architecture guidelines.

**Process:** Invoke Claude/ChatGPT as the Developer agent in your IDE or chat, and provide the prepared story file content. For example, you might open `docs/stories/us-001.md` and copy its content into the Developer agent's prompt (if not auto-loaded). Then say:

> "*As the Developer, please implement the following story.*" (and paste the story content, or if in VS Code with chatmode, it may have it loaded already).

The Developer agent will then proceed to write code and respond with the proposed changes. According to BMAD's guidance, the Developer should: - **Write the Code**: Create or modify files under the `src/` (or whatever directory) to implement the functionality. If using VS Code with a connected AI, it might directly open a new file or suggest file content. If using just chat, it will output code blocks for each file or section. For Breeze API integration, it might output a new file like `breeze_client.py` with a class or functions for `BreezeClient` handling auth and data retrieval, for example. - **Follow Architecture**: E.g., if the architecture said to use the official SDK, ensure the code uses `from breeze_connect import BreezeConnect` and not some unofficial approach. The agent might recall or you may remind it of such details (the architecture.md content is available to it if configured). - **Include Comments/Docs**: The Developer agent persona might be instructed to comment the code for clarity or write docstrings. This is useful for maintainability. - **Testing Stubs**: Sometimes, the Developer agent might also generate unit tests or usage examples if instructed by tasks. For instance, it could write a quick test in the code or a separate `test_breeze_client.py` in a tests folder to verify it can fetch data (depending on how BMAD's Developer persona prompt is set, it often encourages writing tests or at least ensuring criteria). - **Not commit secrets**: It should avoid hardcoding API keys; perhaps it will expect keys to be provided via environment or config. (Ensure you handle that with either a config file to be read or have the dev create a placeholder like `API_KEY = "<your_api_key>"` to be filled by user.) - **Output Format**: In chat, the code might be provided in markdown fences. You will need to copy these into your project files manually if not directly integrated. The BMAD approach keeps you in control: you review the code output, paste it into actual source files in your IDE, and can run it to test.

For example, for **Story us-001 (Breeze API)**, the Developer might output:

```
# file: src/breeze_client.py
import requests
from breeze_connect import BreezeConnect

class BreezeClient:
    def __init__(self, api_key, api_secret, session_token):
        self.breeze = BreezeConnect(api_key=api_key)
        # Generate session
        self.breeze.generate_session(api_secret=api_secret,
session_token=session_token)

    def get_ltp(self, symbol: str):
        # Fetch latest price for given symbol
        response = self.breeze.get_quotes(stock_code=symbol,
exchange_code="NSE", product_type="cash")
        # assume response contains price in a field
        return response['Success'][0]['ltp'] if 'Success' in response else
None

    def place_order(self, symbol: str, action: str, quantity: int, price:
float=None):
        # Place a market or limit order via Breeze
        order_params = {
            "stock_code": symbol,
            "exchange_code": "NSE",
            "product": "cash",
            "action": action,  # "BUY" or "SELL"
            "order_type": "market" if price is None else "limit",
            "quantity": quantity
        }
        if price:
            order_params["price"] = price
        result = self.breeze.place_order(**order_params)
        return result
```

*(This is an illustrative snippet; the actual API methods might differ, but the idea is the Developer agent will attempt something like this, guided by Breeze API docs provided.)*

The Developer might also suggest a quick test usage:

```
# Example usage (for testing, not part of production code):
if __name__ == "__main__":
    client = BreezeClient(api_key="YOUR_API_KEY",
api_secret="YOUR_API_SECRET", session_token="YOUR_SESSION")
    print("LTP of RELIANCE:", client.get_ltp("RELIANCE"))
    print("Placing test order for RELIANCE:", client.place_order("RELIANCE",
action="BUY", quantity=1))
```

and comment that the user should replace credentials and be cautious with actual orders.

**Human oversight:** After the Developer outputs code, you should review it. Does it align with the acceptance criteria and architecture? If something is missing or wrong (e.g., wrong API usage), you can correct it or even ask the Developer agent to refine: "*The code seems to use an endpoint* `get_quotes` *with certain params; ensure it matches Breeze API correctly, and handle if the response has errors.*" The AI can then adjust the code. This interactive refinement is part of the development process – you are effectively the senior engineer guiding the AI developer.

Once satisfied, you integrate the code into the project, run it (maybe with test keys or in a test environment) to verify basic functionality. Keep in mind, for something like API integration, you might need to actually use valid credentials to test. If you have them, you can try a quick run of `get_ltp` to see if it returns a price. This is outside BMAD's scope (it doesn't run code for you), but as a developer you'd do it manually or with a small test script.

With code in place, commit it (with a message referencing the story, e.g., "feat(us-001): Breeze API integration complete"). BMAD suggests keeping commits small and tagged by story ID [23] – a good practice for clarity.

## 6. Quality Assurance and Testing (QA Agent)

**Role:** The QA agent is responsible for validating that the implementation meets the acceptance criteria and is free of major defects. Claude/ChatGPT in QA persona will examine the code and possibly run logical tests in its head or propose automated tests. The QA agent essentially performs code review and testing within the context provided. Since the user will ultimately test the system with real data as well, the QA agent's job is to catch issues early and ensure that the story's definition of done is met.

**Process:** After the Developer phase for a story: - Invoke the QA agent with the context of the story file and a summary of what was implemented (and possibly the code, if small enough or critical sections). In an IDE integration, the QA agent might have access to the changed files; if not, you might copy portions of the code or describe them. - For example, prompt: *"As the QA, verify that story us-001 (Breeze API integration) is implemented correctly. The acceptance criteria are X, Y, Z (list them). The developer has provided the BreezeClient class with methods get_ltp and place_order (share code snippet or path). Check if these fulfill the criteria and suggest tests."*

The QA agent will then: - Cross-check code vs criteria: It might say *"AC1: fetch latest price – The* `get_ltp` *method calls* `get_quotes` *and returns* `ltp`*. This seems correct if the API returns* `ltp` *in the response. We should test it with a known symbol. AC2: place order – The* `place_order` *method uses the provided parameters correctly (market if no price, etc.). It appears to satisfy the requirement of placing an order. AC3: secure credentials – The code requires passing credentials into BreezeClient; as long as those are not hardcoded in the code (which they are not, they're parameters), this is okay. It prints them only if the user does so in main (which should be removed in production)."* - It might identify edge cases or bugs: For instance, *"The code does not handle the case when the API response for get_ltp is an error or empty – it returns None. Is that acceptable or should it raise an exception? Possibly fine, but worth noting."* Or *"In place_order, the code doesn't confirm if the order was successful except returning result. Perhaps log the result or ensure it's as expected."* - Suggest tests: The QA might propose writing a unit test for BreezeClient. Without actual API access in QA context, it might suggest mocking BreezeConnect. But if we have credentials, maybe QA will say *“Run BreezeClient in a test mode with your credentials to ensure it returns a price and can place an order (maybe a very small quantity or a dummy order type if available).”* If the QA agent is integrated in an environment that allows code execution, it could attempt to run it (some advanced setups do that), but here likely it will be a conceptual check. - Mark the story as **Pass or Fail**: If everything looks good (no criteria missed), QA can say the story is verified. If not, it will list the issues to fix.

Based on QA output, if issues were found, go back to Developer agent to fix them: - For example, if QA noted *“No error handling for get_ltp when API fails,”* then update the code to handle exceptions or error responses. This may be a quick manual fix or ask Developer agent to implement a try/except around the API call and perhaps retry or propagate error properly. - Then run QA agent again on the updated story/code.

Only once QA agent (and you, the human) are satisfied that the acceptance criteria are met and no major bug is present, consider the story **done**. The QA agent might update the story file to mark it as verified or note test results (e.g., *“Tested with symbol RELIANCE, got price X. Test order placed in paper mode, received order ID, indicating success.”*). Keeping such notes is useful for future reference.

## 7. Iteration: Continue with Next Stories

Follow the SM → Dev → QA cycle for each user story in priority order. This iterative process is essentially an **Agile sprint** managed by AI agents: - Take the next story (say “Intraday strategy module”), have SM prepare it with context (e.g., include any formula or strategy rules specified in PRD like “use moving average crossover as initial strategy”), then Dev implements it (writing code for perhaps a class `IntradayStrategy` that reads live data and decides trades), then QA tests it (maybe by simulating some data or checking the logic against scenarios). - Then the next story (e.g., “Swing strategy module”) and so on.

As you progress, integrate each new module with the previous ones: - E.g., after building intraday and swing strategies separately, there might be integration steps to ensure they don’t conflict (perhaps the system should not enter both an intraday and swing trade on the same stock simultaneously unless intended). The architecture might have a coordinator for that. Ensure to test such interactions as well (maybe a QA story dedicated to integration testing). - The teacher-student loop likely will come after the basic strategies are in place (since the teacher needs something to evaluate). That story might involve creating a mechanism to log all trades and outcomes, then some offline process to adjust strategy. The Developer might implement it as a function that runs after market hours to tweak parameters (like adjust thresholds or retrain a model if using ML). QA would then have to verify that adjustments are happening logically (maybe by forcing a scenario where the student consistently loses money and seeing if teacher reduces risk or something).

Throughout these iterations, maintain discipline of **keeping context in the story files** and switching roles as needed, rather than using one giant chat. This prevents context bleed and confusion. It also creates a documentation trail: each story file serves as a mini-spec + implementation log, which is great for collaboration and future reference [3] .

Additionally, leveraging **multi-agent parallelism** can speed things up if you’re comfortable: BMAD’s methodology and others have noted that splitting work among specialized agents can accelerate development [24] . For example, while the Developer agent is coding Story 1, you (or another instance) could have the Architect agent refine the architecture for a later feature, or have the PM agent flesh out more stories ahead. Just be careful to integrate changes sequentially. The structured approach yields benefits like *parallel development, built-in quality checks (different agents catching issues), clear separation* *of concerns, and better documentation* [25]. However, if you prefer a linear approach (especially if you're the only one operating the agents), that's perfectly fine too.

By the end of Phase 2, you should have the full codebase of SenseQuant implemented, with all features from the PRD addressed. The BMAD process would have ensured that at each step, the AI had the necessary context to do its job well, and that nothing critical was forgotten between steps. This addresses the classic issue of an AI losing track of earlier instructions by providing fresh, targeted context at each stage [26]. In other words, each agent instance started with exactly the info it needed and nothing else, improving focus and productivity.

## BMAD Agent Roles and Replacing them with Claude/ChatGPT

Now, let's clarify the **role of each BMAD agent** in this project and how we'll use Claude or ChatGPT in place of BMAD's default agents. BMAD defines several persona agents, each with a specific mandate and style [1]. We will utilize these roles as follows:

- **Analyst Agent (Claude/ChatGPT as Analyst):** The Analyst acts as a **business analyst**, focusing on understanding user needs and formulating requirements. In our plan, the Analyst (Claude/ChatGPT) produced the PRD from the user's objectives. It remains in character by asking clarification questions and ensuring nothing is assumed. For SenseQuant, the Analyst's output was the structured list of features (intraday, swing, sentiment, etc.) with detailed descriptions. By using an LLM with the Analyst persona, we avoid having to remind the AI of basic context repeatedly – the persona file tells it to think and communicate like a requirements analyst [1]. In practice, you might only use the Analyst agent at the project's start (and maybe later if new features or changes in requirements come in).

- **Product Manager (PM) Agent (Claude/ChatGPT as PM):** The PM (or Product Owner) is like the **planner/strategist** in the team. This agent takes the high-level requirements and turns them into an actionable plan (stories, priorities). We used the PM agent to generate the backlog and will use it whenever we need to reprioritize or add new stories. Claude/ChatGPT in this role follows the methodology to produce a "prioritized backlog (epics → stories)" [22]. This ensures our development is incremental and value-driven (tackling highest value pieces first). If mid-project the scope changes or we realize we need an extra feature (for example, perhaps adding a new strategy or a GUI), we could call on the PM agent again to update the backlog accordingly.

- **Architect Agent (Claude/ChatGPT as Architect):** The Architect is responsible for the **system's technical blueprint**. Claude/ChatGPT as the Architect agent provided the design for how all parts of SenseQuant fit together. This role was crucial to ensure the custom AI modules (teacher-student, sentiment) were **preserved and integrated** in the design. The persona likely has instructions to produce clear, consistent architecture docs, and possibly check for consistency with requirements. If any later technical design questions arise (e.g., "How should we modify the system to scale to more stocks or to include futures trading?"), you can re-engage the Architect agent. It will always refer to the PRD and existing architecture to extend or modify the design, keeping the big picture coherent.

- **Scrum Master (SM) Agent (Claude/ChatGPT as Orchestrator):** The Scrum Master is essentially the **orchestrator** of the development workflow. In BMAD v4, the SM agent generates the detailed story plans and guides information flow between planning and coding [2]. We utilized the SM to create story files with embedded context for each feature. Claude or ChatGPT as SM strictly followed the framework's patterns – e.g., writing out tasks and criteria, then handing off to Dev. The SM agent ensures that *nothing gets lost in translation*: it grabs the acceptance criteria from the PM's output and the relevant parts of architecture, and writes them into the story file that the Dev will read [2]. Using an LLM for this is straightforward since it's more about organizing and rephrasing existing info than creating new code or requirements. The SM role might be less "creative" and more procedural; nonetheless, having Claude/ChatGPT do it saves us manual work and avoids human error in forgetting context. If using the web UI with the team file, you would invoke this by `*agent sm` (if "sm" is the Scrum Master identifier) and instruct it per story.

- **Developer (Dev) Agent (Claude/ChatGPT as Developer):** The Developer agent is the **coder**. It writes application code, test code, and updates documentation according to the story's instructions [27]. We replaced the default BMAD Dev agent with Claude or ChatGPT (GPT-4 ideally for coding prowess). In VS Code, this could be the Copilot chat in Developer mode; in web, `*agent dev`. The Developer persona is configured to focus on implementing rather than exploring requirements – it trusts the provided story file and architecture notes. This is great because it means the LLM won't wander off thinking up new features; it will stick to *"implement X with method Y as design says"*. In our context, Claude/GPT-4 as Dev wrote the Python modules for trading strategies, API integration, etc. Each time we fed it a story file, it responded with the code solutions. It's like having an AI pair-programmer who has the design doc open next to them at all times. We also expect the Developer agent to incorporate any coding standards or checklists provided by BMAD (there are templates for certain tasks), resulting in consistent code style and structure across the project. Notably, since we have some **AI logic** (like perhaps training a model), the Developer agent is the one to implement those algorithms too – for example, coding how the Teacher updates the Student's parameters. We must ensure it follows the logic described in our design (the LLM might suggest alternate ML approaches, but we should align it with our intended method unless we approve the change).

- **QA (Tester) Agent (Claude/ChatGPT as QA):** The QA agent is the **test engineer and quality analyst**. Using Claude/ChatGPT here provides a second pair of eyes on the code. The QA persona in BMAD is likely instructed to be thorough, skeptical, and detail-oriented – it will compare the implementation against requirements and try to find any discrepancy or edge case [27]. We used the QA agent after each story's development. In practice, the QA agent can also generate test cases or even test code. For instance, it could write a pytest function to simulate a scenario or verify output. We, as humans, might then run these tests to see results. By replacing the QA agent with a strong LLM, we also get the benefit of its analytical ability to reason about logic. For example, ChatGPT might catch a logical bug that we missed, or suggest a better way to handle a failure mode. The QA agent's role is also to ensure **the custom criteria are met**: e.g., it will remember that *"all intraday positions must be closed by end of day"* and look in the intraday module code to see if that rule is enforced. If not, it flags it. This protects our domain requirements from being overlooked in code. After QA sign-off on a story, we have a high level of confidence that the feature works as intended (at least in theory – real integration testing still to come).

- **Other Roles (Optional):** BMAD also mentions roles like *Project Manager, Product Owner, UX Designer, etc.* For our context:

- *Project Manager vs Product Manager*: We essentially used the PM agent to cover both product/ backlog planning. We could consider that done.
- *Product Owner (PO) Agent*: This might overlap with PM/Analyst – since you as the user are effectively the product owner providing requirements, we didn't explicitly use a separate PO agent. The Analyst and PM together fulfilled that.

- *UX Designer Agent*: If SenseQuant were to have a significant user interface (say a web dashboard to monitor trades), we might involve a UX agent to propose wireframes or user interactions. The user hasn't requested a UI beyond maybe notifications, so we likely skip UX agent. If needed, that agent could design a simple interface or output format (like a nice format for logs or a small GUI using Streamlit for example). But given "assistant" nature, not required unless the user specifically wants it.
- *BMAD Orchestrator*: There is a **bmad-orchestrator** agent that can answer questions about the process itself. We might not need to explicitly use it unless we have confusion about commands. But it's there (for example, in a chat you could do `*agent bmad-orchestrator` to ask "How do I do X?"). We mention it just as a resource: if at any point you are unsure how to proceed with BMAD flow, the orchestrator agent (which is part of the team file) can help clarify the next steps [11] [15].

In summary, **Claude or ChatGPT will impersonate each of these roles** depending on the task at hand: - Use separate chats or role-switch commands so that the AI always has the appropriate persona context loaded (preventing confusion between roles). - Each role's prompt (via BMAD persona definitions) ensures consistency in style and knowledge for that role [1]. For instance, the Developer role knows it should produce code and not long-winded analysis, the Analyst role knows it should not output code but only requirements, etc. - This structured role-play significantly improves project organization. It's akin to having a full agile team of AIs: one designs, one codes, one tests, coordinated by the framework of BMAD.

## Story File Handoff and Agent Prompting Strategies

A critical aspect of this multi-agent orchestration is how information is handed off between agents. BMAD uses **story files** as the medium of context transfer [2]. Let's highlight how we structure these and suggest some concrete prompting and context management tips for each handoff:

- **Story File Content:** Each story file (in `docs/stories/`) contains the details needed for development of that feature. We ensure it has:
- *Title/ID:* e.g., `us-002: Implement Intraday Trading Strategy`.
- *Description:* A brief narrative of the feature from a user perspective (e.g., "As a trader, the system should automatically execute intraday trades on volatile stocks and square off all positions by day's end.").
- *Contextual Details:* Relevant excerpts from PRD and architecture. For intraday strategy, include from PRD: what intraday trading entails (maybe mention typical trade frequency, risk constraints) and from architecture: which module/class handles it, what inputs (market data, maybe sentiment, etc.) and outputs (trade signals) it has.
- *Technical Guidance:* If architecture decided a certain approach (e.g., "use a moving average crossover strategy for initial implementation"), include that. If specific formulas or thresholds were suggested (say, "Buy if 50MA > 200MA and RSI < 70"), list them. This ensures the Developer doesn't have to invent strategy logic; it's coming from either domain knowledge or your instructions.
- *Tasks:* A breakdown of what needs doing. For intraday strategy example, tasks might be:
    1. *Implement a class* `IntradayStrategy` *with a method* `generate_signals(data)` *that returns trade signals (buy/sell) for given intraday data.*
    2. *Use a simple strategy: e.g., if 5-minute momentum > threshold and no open position, buy; if profit target or end of day nearing, sell.* (This could be more rule-based initially; we can refine or allow the AI to incorporate a basic algorithm here.)
    3. *Integrate sentiment: if a negative news sentiment is detected for a stock, maybe filter out buy signals for that stock.* (If that's part of requirements.)

4. *Ensure the strategy marks all open positions to close by a cutoff time (e.g., 3:25 PM) to meet intraday closure requirement.*
5. *Output: signals should include stock, action, quantity (could be determined by some position sizing logic), etc. For now, maybe fixed quantity for simplicity.*

- *Acceptance Criteria:* Clearly itemize:
    - *AC1: The IntradayStrategy produces at least one trade signal given a sufficiently volatile price series (test on historical intraday data).*
    - *AC2: It always issues a "close" signal for any open position before the market close time.*
    - *AC3: The strategy uses the sentiment input to avoid going against strongly negative sentiment.* (If we enforce that.)
    - *AC4: The strategy's logic can be toggled or adjusted easily (modular design) for future refinement.* (Maybe not a formal AC, but a design consideration.)

The story file thus reads like a mini spec + task list. The SM agent in our plan assembled much of this from existing docs, but as a human, you might also add any details you want to ensure (especially domain logic that an AI might not know by itself, like specifics of trading rules).

- **Prompting the Developer with Story Files:** When handing off to the Developer agent, you essentially feed it the story file content as the prompt (plus any codebase context if using an IDE). You can prepend an instruction like *"Implement the following story. Be sure to follow the tasks and meet the acceptance criteria. If something is unclear, assume a reasonable approach and note any assumptions."* This lets the AI know it should proceed to coding. Since the story contains all needed info, ideally the Developer just says "Understood, implementing now…" and then outputs code. In a VS Code environment, the Developer might directly create/modify files per tasks. In ChatGPT web, it will show code in the chat, which you then manually apply.

- **Handling Large Context:** If a story file grows large (some can be very detailed), ensure to use an LLM with sufficient context window. Claude 2 with 100k context can easily handle very large story files or even multiple files at once [26]. GPT-4 with 8k might require being a bit more concise or splitting into parts (but 8k is usually enough for a story plus some code). If needed, you can break a story into sub-stories to keep context smaller.

- **QA Agent Handoff:** For the QA agent, the story file plus the resulting code (or a summary of code changes) should be provided. If using an IDE that can diff or show file changes, you could give QA the git diff or list of files modified for that story. If using chat, you might copy the relevant function implementations into the prompt for QA to see. Prompt QA like: *"Here is the story with AC, and here is the code that was written. Please verify compliance and correctness."* The QA persona should then respond with a point-by-point analysis.

- **Agent Handoff Example:** Consider the *sentiment analysis integration* story.

- Story file might include: PRD snippet "The assistant shall incorporate sentiment from news on stocks to filter trade decisions," architecture snippet "SentimentAnalyzer uses external API or model to score news articles daily," tasks like *"Implement a function fetch_news_headlines(api) and get_sentiment_score(headlines) using a simple sentiment model. Integrate this by adjusting trade signal generation: e.g., do not take long trades on stocks with strongly negative sentiment."*, and AC such as *"If a stock's sentiment score is below -0.5 (very negative), the system does not open new buy positions for that stock that day."*
- Developer reads this and writes code: perhaps a module `sentiment.py` that calls some news API (or reads a static RSS feed for simplicity) and a sentiment model (maybe TextBlob or a pretrained model) to produce scores. Then modifies the strategy module to use these scores.

- QA then checks: Did we indeed prevent trades on negative sentiment? It might simulate by calling the sentiment function with a mocked negative headline and see if the strategy skips a trade.

- **Maintaining Context across Handoffs:** One advantage of BMAD's structured approach is that context is always intentionally provided, not left to chance in a single chat thread [2]. This means you should **avoid reusing the same chat session for different roles without resetting context**. For example, if you used ChatGPT for the Developer role, do a fresh prompt for QA role (or better, a different chat or use the `*agent qa` directive) so that the QA agent isn't influenced by how the Developer "thought" while coding. It should approach with a fresh perspective, only seeing the final outputs and the original criteria (just like a real QA tester who wouldn't typically sit in on the coding). This mimics a separation of concerns and can lead to catching things the Developer (AI) might have glossed over. The BMAD guidelines explicitly highlight using separate persona contexts to avoid mixing roles [28].

- **Prompt Engineering for Agents:** While BMAD provides a lot of the role definitions, you as the user still control how you *ask* each agent to do its job:

- For generative tasks (Analyst writing PRD, Architect design), prompts should be open-ended but within scope, as shown in previous sections.
- For the Developer, be direct: e.g., "*Write the code for the above tasks.*" If you expect code in certain structure (like "create these files..."), mention it.
- For QA, you can explicitly instruct "*If any acceptance criterion is not met, list it as a defect. If all are met, confirm the story is verified.*" This ensures the QA agent outputs something actionable (either a list of defects or a pass).
- Also consider instructing the agents to keep outputs **concise and structured** (since you prefer structured, context-rich outputs). BMAD's templates usually handle this (e.g., Developer outputs just code, QA outputs a list of checks). But you can reiterate: "*Architect: use bullet points or numbered sections for clarity in the architecture doc.*", "*Analyst: structure the PRD into sections (overview, features, etc.).*"

By carefully crafting story files and prompts for each role, you maintain a high level of control over development. The story-centric approach of BMAD essentially means *each user story is a mini-project with its own specification and lifecycle*, which makes the whole development very organized. And because all these story files, PRD, arch, etc., remain in your repository, you have thorough documentation of how and why things were built – which is useful for future contributors or for debugging down the line [3].

## Preserving Custom AI Logic (Teacher–Student, Sentiment, Multi-Strategy)

One of the user's key concerns is that their existing **custom AI modules and domain logic should not be overwritten or lost** when using BMAD. We address this by explicitly incorporating those elements at every stage of the BMAD workflow, rather than letting the AI agents take generic shortcuts.

Here's how we preserve each major custom component:

- **Teacher–Student Training Loop:** This is a unique aspect of SenseQuant's design. We made sure it's front-and-center in the requirements (Analyst/PRD) and architecture. For example, the PRD described the need for a feedback mechanism to improve the trading strategy, and the architecture allocated a Teacher and Student component. When the Architect agent was generating the design, we confirmed it included, say, *"A reinforcement learning loop: the Student module (policy) makes trades, the Teacher module (critic) evaluates performance daily and updates the policy."* This ensures the Developer agent later knows to implement something for it. We will not allow the AI to replace this with a standard one-pass algorithm. If the Developer agent in story tries to, say, propose a totally different approach ("I'll use a random forest to predict trades" which doesn't incorporate teacher feedback), we will steer it back: remind it of the teacher-student requirement. **In practice**, likely we'll have a story like "Implement learning feedback loop" where tasks could be:

- *Collect trading performance data (P&L, win rate, etc.) from past N trades or days.*
- *Implement a Teacher module that analyses this performance vs benchmarks.* (Perhaps it could compute if the strategy beat buy-and-hold or track drawdowns, etc.)
- *Adjust the Student strategy parameters accordingly.* (If it's a rule-based student, maybe tweak thresholds; if it's a model, maybe retrain on new data with a higher weight for recent performance.)
- *Ensure this loop can run periodically (like end of each week or overnight).*
- The acceptance could be something like *"After poor performance (e.g., large losses), the system reduces risk in subsequent trades (e.g., smaller position size or more conservative thresholds), indicating learning."* We might simulate that to test it.

By structuring it this way, we **preserve the concept**. The actual effectiveness of the teacher-student loop might need fine-tuning and could be a complex ML task, but at least the scaffolding will be implemented, and you can later plug in more sophisticated logic if needed (e.g., train a model offline with reinforcement learning algorithms – that could be beyond the initial scope but at least the architecture won't preclude it).

- **Sentiment Analysis Module:** We keep this separate and intact by giving it a dedicated story and describing exactly how it influences trading. For example, we instructed the architecture to have a `SentimentAnalyzer` component. The Developer then implements it (maybe using an API like NewsAPI to fetch news and a simple sentiment model to rate them). The key is we told the AI agents *why* it's there – to improve trading decisions. Therefore, when coding, the Developer agent integrated it into the strategy logic rather than ignoring it. If we didn't specify, an AI might think sentiment is an optional fancy and skip it, but because it's in the requirements and acceptance criteria (e.g., *"system must factor in news sentiment when deciding trades"*), the AI must include it to consider the story done. We as humans also watch for this during QA – if the QA agent or we notice the strategy doesn't actually use the sentiment data, we flag it and get it fixed. This way, the sentiment module remains a first-class citizen in the final product, not dropped due to AI convenience.

- **Multi-Agent Strategy Execution:** The system is supposed to handle multiple strategies (intraday and swing, possibly more). We have preserved this by making **intraday and swing trading separate modules** in the design, each implemented in separate stories. The architecture might outline that in the future more strategies can plug in (for example, a structure where each strategy module registers with a main engine and outputs signals, which the Execution module then handles). We should ensure the final integration supports both concurrently:

- For instance, if intraday and swing could give conflicting signals on the same stock, decide how to handle that (maybe intraday has priority for short-term moves, swing only triggers on different conditions). These rules can be documented in architecture and enforced in code.
- Multi-agent in runtime could also mean using different AI models for different tasks (one for sentiment, one for technical patterns). If that was the case, ensure the design supports it (like a

separate thread or process for each). However, likely here “multi-agent strategy” meant the combination of intraday & swing plus the teacher maybe acting as another agent in logic.

- The key is that **we don’t funnel everything into one monolithic algorithm**; we maintain the separation. The BMAD approach actually helps here: because we treat each as a separate story, the code naturally ends up modular (intraday module, swing module). This is opposed to if one asked a single AI in one go to “build a trading bot with everything” – it might merge them or drop one. With BMAD, we did one at a time and integrated them.

- **Preventing Overwrite of Domain Logic:** One risk when using AI to architect or code is that it might propose a completely different approach (like using some pre-built library or a strategy not asked for). To guard against that:

- We maintained a strong guiding hand in prompts. For example, when creating PRD and architecture, we clearly spelled out these custom choices (teacher-student, etc.) so that the AI treats them as given constraints, not problems to solve anew.
- BMAD’s human-in-loop philosophy [4] empowers us to reject or correct any AI suggestion that violates the intended logic. If the Architect agent had said “Instead of a teacher-student, we’ll use a single deep learning model,” we would step in and say “No, stick to teacher-student as specified.” The persona will then comply.
- During development, similarly, if Developer agent tries to simplify (maybe it doesn’t want to implement the teacher because it’s complex and instead just leaves a TODO), we catch that in QA or code review and insist on completing it.
- We may also incorporate any **existing code or prototypes**: If the user already has some code for these modules from earlier experimentation, we can feed that to the Developer agent as a starting point. For example, “Here’s a rough sentiment analysis script I wrote, integrate this.” The BMAD Developer can then adapt it to the new structure. Reusing proven pieces avoids any risk of them being left out and can save time.
- By the end, the delivered codebase should clearly contain the separate modules for teacher, student, sentiment, etc. The **teacher-student training code** might be a bit abstract at first (maybe just a stub that reduces risk after a loss, etc.), but it’s there and ready to be enhanced with more AI training later. The **sentiment integration code** will be fully functional (maybe not extremely advanced NLP, but at least something like keyword-based sentiment or an API call).

In short, **BMAD acts as an overlay**, not a replacement, for the custom AI logic. We’ve used it to structure the development process, but we always inserted the domain logic into that structure at the right points. This way, BMAD’s agents help implement *our* vision of SenseQuant, rather than their own generic trading bot. The output is the best of both: a system that meets the original custom requirements, built efficiently and systematically through multi-agent collaboration.

## Hosting, Collaboration, and Testing Workflow

With the code developed and reviewed through the BMAD-driven process, the final steps are to host the application, allow for collaboration (if multiple people or just you and AI maintaining it), and perform integration testing – all within the BMAD framework and VS Code environment.

### Deployment and Free Cloud Hosting

The user intends to deploy SenseQuant on a **free cloud hosting** solution. Given this constraint, the plan should favor a deployment approach that is cost-free and relatively easy: - **Containerize or Scripted Deploy:** We can containerize the app with Docker, which could then be run on services like Heroku (free tier), Railway (free for limited usage), or fly.io. Another approach is to use a serverless function for parts (though a trading bot likely needs a persistent process for market hours). - Since intraday trading needs the bot running continuously during market hours (6-7 hours a day), a small VM or container on a free tier is suitable. For example, Oracle Cloud's Always Free VM, or an AWS EC2 micro (if within free usage limits) could run it. - The plan should be to schedule the bot to start at market open and stop at close (to save resources, or it can run 24/7 if usage is low). - **Credential Management:** Because deployment involves secrets (API keys for Breeze), ensure environment variables or a secure config is used. Document this in a README (which the Developer agent or yourself should write as part of final docs). - The BMAD workflow itself doesn't handle deployment, but we can have a story "Deploy application to cloud" where tasks would be: 1. *Write a Dockerfile for the app.* 2. *Create a free account on XYZ cloud and configure pipeline.* 3. *Ensure the app can be started via a single command.* 4. *Document how to set environment variables for API keys on the host.* - The Developer agent can help write the Dockerfile and perhaps a simple GitHub Action for CI/CD if needed. - However, if this is too detailed, we can do it manually or partially automated. For a free solution, sometimes simpler is using something like Replit which can host continuously running Python apps for free (with always-on if you subscribe, but at least during dev, it's free).

**Recommendation:** Use the smallest, simplest free hosting first, e.g., deploy on **Heroku** (which used to have a free tier for small apps; if not available now, consider **Railway** or **fly.io** for a free container). The app being primarily Python, we can use their free Postgres or just file logging if needed (no heavy DB).

Document the hosting steps in the project README. The BMAD PM agent or Architect might help outline environment requirements in architecture doc (like "requires Python 3.x, packages X, Y, Z, runs continuously"). You may ask an agent to output a one-click deploy instruction if possible.

## Collaboration Using BMAD in VS Code

If you collaborate with others or simply manage the project over time, the BMAD structure in VS Code is beneficial: - All personas can be re-used at any time to add features or fix bugs. For example, if a bug is found later, you can create a story for it ("Bug: XYZ") and then use SM -> Dev -> QA cycle to fix it. The consistency of process remains, which is great for maintainability. - If multiple people are involved, they can also use the story and docs to understand context. E.g., a teammate could read `docs/architecture.md` to see how things work before modifying code. - Version control with story-based commits (as we practiced) means your Git history is nicely organized by feature. This is useful in code reviews or rollbacks if needed. - The VS Code integration with agents allows concurrent usage: you can literally have an Analyst chat open (to discuss a new feature idea), a Dev chat open (writing code), and a QA chat (writing tests) in parallel [11]. This mirrors a real team where different roles work in tandem. Just ensure they coordinate via the docs and code repository (which is the single source of truth).

In a scenario where, say, you and a colleague use BMAD, one could take on the role of orchestrating the AI agents while the other monitors or handles the code integration. Or both could alternately use the AI agents to speed up tasks. Because BMAD defines outputs (like PRD, tasks, etc.), it's clear what each person/agent should be doing at a time, reducing confusion.

## Testing Flow (Integration & User Acceptance Testing)

Even though we used QA agent for each story, we should do full **integration testing** of the system: - Combine all components and simulate a trading day. This can be done by using historical data (for intraday and swing) to replay a day's market and seeing if the system behaves correctly end-to-end (this is essentially a backtest). You could write a simple backtesting script or have the QA agent assist with this. - For example, have a script that reads historical minute-by-minute prices for a given day for a few stocks, feed them to the IntradayStrategy module as if they were live, and verify that trades are triggered and closed properly. The QA agent could draft this script if prompted: “simulate intraday on data.csv and verify positions count.” - Test scenarios: - A trending day vs a flat day (does the strategy trade too much or appropriately?). - A day with a big negative news event on a stock (simulate by injecting a very negative sentiment score) – check the system skips that trade. - Multi-day test for swing: feed data for a month and see if swing trades open and close over days as expected. - Test the teacher-student: perhaps deliberately feed poor performance (maybe the strategy loses money for 5 days) and see if the teacher module reacts (like it should modify something). This might be hard to verify automatically if the adjustments are subtle. But at least check that the teacher function executes without error and changes a parameter (e.g., lowers risk parameter). - If possible, do a **paper trading run** in live conditions for a short period. Because the user has a real ICICI account, they might try running the bot in market hours with either very small capital or in a “virtual” mode (maybe Breeze API has a paper trading mode or simply not confirm orders). This will reveal any issues with real API usage (latency, authentication refresh, etc.). Logging will be crucial here – ensure the system logs important events (the Developer can implement logging to console or file).

BMAD can be extended to help with testing as well: - Write an **End-to-End Test Story** as mentioned earlier, and have QA agent script out tests. Even if QA can’t run them itself, it can generate code for tests which you then execute. - For instance, QA could write a pytest that mocks Breeze API responses and tests that, given certain prices, the strategy yields certain orders. This kind of automated test secures the system’s core logic from regressions. Include such tests in the repository.

## Monitoring and Maintenance

Given this is a trading system, ongoing monitoring is needed even after deployment: - Set up notifications or logs to alert if something goes wrong (e.g., an exception, or if no trades happen when expected). - BMAD doesn’t directly handle runtime monitoring, but as developers we can incorporate this (maybe a simple health check log, or an email if the process crashes).

For maintenance, future changes can go through the same BMAD process: - If the user later wants to add a new feature (like options trading or a new indicator), they can create a new story, feed the PRD with new requirement (or update PRD), have the Architect update the design, and then implement via Dev/QA. The structured approach ensures even future changes remain organized.

**Collaboration with AI Agents Over Time:** As AI models improve or change (Claude, GPT-4, etc.), you can continue using BMAD with newer models. The story files and docs remain model-agnostic references of truth. If you switch from ChatGPT to Claude for the Dev role at some point (or vice versa), you just need to ensure the persona instructions are loaded, and the new model can pick up by reading the same docs. This is a robust way to not be locked to one AI or one session’s memory.

In conclusion, by deploying on a suitable free host, thoroughly testing the integrated system, and using BMAD’s methodology within VS Code for any iterative improvements or collaboration, we set up SenseQuant for a stable launch and easy future evolution.

## Maximizing Claude/ChatGPT Integration in BMAD's Story-Centric Workflow

To wrap up, here are some recommendations to get the most out of using Claude or ChatGPT as the engines behind BMAD's agents, ensuring the combination is efficient and effective:

- **Use Large Context Windows to Your Advantage:** One of the strengths of models like Claude 2 is the ability to handle very large inputs (up to 100k tokens) without losing coherence. BMAD's story-centric approach already chunks work into reasonably sized pieces, but you can push it further with Claude. For instance, you could feed Claude's Developer agent not just the story file, but the entire PRD and architecture as context every time (since 100k can easily include it) – this ensures it has *global context* and might catch if something it's doing in one story contradicts another requirement. ChatGPT's GPT-4 (currently 8k or 32k tokens) can also handle a lot, but maybe not every document at once. In practice, we saw that giving each agent only what it needs (story + relevant excerpts) works well to avoid confusion [2]. However, during complex integrations, you might open a Claude instance and paste in multiple files (PRD, architecture, maybe a couple of story files) and ask it to reason about the system as a whole. This can be useful for final reviews or debugging. For example, *"Claude, here's the architecture and here are the intraday and swing strategy code. Do these align? Any contradictions or improvements you see?"* – Claude might give a holistic analysis, something very powerful for quality assurance.

- **Role-specific Conditioning:** The BMAD persona files effectively act as system prompts that condition the model to behave in certain ways [1]. Make sure you always include those when starting a session. If using the web UI route, don't skip directly to tasks without first loading the team/agent file. The difference is night and day – with the persona, the AI will stick to the expected format (e.g., the Developer agent will output code directly rather than conversational fluff). This structured output is exactly what we want for clarity. If you find the persona's style not ideal, you can tweak it (since BMAD is a framework, those files are editable). For instance, you might edit the Developer persona file to say "Always include a brief comment at the top of each code file explaining its purpose." Then when ChatGPT/Claude generates code, you'll get that automatically. This is a way of customizing the AI's behavior globally for the project.

- **Iterative Refinement with Claude/ChatGPT:** Even with good prompts, the first output might not be perfect. One advantage of having Claude or ChatGPT is you can converse and refine in each role. BMAD encourages that human feedback loop [4]. For example, if the Architect's first draft is off, you can say *"Revise the architecture to include a module for X explicitly"* and it will update it. If the Developer's code has a bug, you can point it out and ask for correction. This saves time since you don't have to fix everything manually; let the AI fix its own mistakes when possible. However, always verify the fixes – sometimes an AI "fix" can introduce new issues, so treat it like a junior dev that needs oversight.

- **Parallelize Agents (if resources allow):** Claude particularly allows multiple instances (and some UIs like Anthropic's console or Slack integration support multiple conversations). You could theoretically run the Analyst agent on Claude and Developer on ChatGPT simultaneously, for example. Or more practically, use one model for all roles but in different chats concurrently. The Reddit example we found had 4 Claude instances (Architect, Builder=Dev, Validator=QA, Scribe=Doc) running in parallel, communicating via files [24]. We can emulate a lighter version: while one story is in Dev, you can start planning the next with Architect. Or have QA reviewing Story 1 while Dev starts Story 2. Just be careful to sync up outputs (don't let Dev get too far if architecture changes). This technique can dramatically speed up development when used carefully, essentially doing in hours what a single agent might take days to do sequentially [25]. In our project, since you're likely orchestrating alone, you might not fully parallelize at first – but know that the option is there if you face a time crunch.

- **Take Advantage of Claude's Code Execution (if available):** Claude has a "Claude Code" mode which sometimes can execute code or at least handle file system operations in an IDE. If you have access to that, it could be integrated with BMAD's test steps. For example, the QA agent could actually run the test script it wrote and report results, or the Dev agent could run a snippet to verify it works. This is still an evolving area, and with free services you might not get full code execution. ChatGPT has the Code Interpreter (now called Advanced Data Analysis) which can execute Python in a sandbox – one could imagine feeding it the project and running an end-to-end test. However, given SenseQuant likely needs real API keys, be cautious (you wouldn't put real keys into Code Interpreter environment as it's ephemeral and not secure). For offline backtesting though, it could be useful: you could upload historical data to Code Interpreter and have ChatGPT/Claude test the strategies on that data, giving a quick analysis of performance. This would be a neat use of AI to evaluate the AI-built system.

- **Use BMAD's Orchestrator for Guidance:** If uncertain about a BMAD command or best practice, don't forget the orchestrator agent (often accessible via a command like `#bmad-orchestrator` or similar in chat) – it's basically a help agent with knowledge of the framework. For example, *"How do I handle a situation where the Developer output is too large to fit in context?"* – the orchestrator might have tips (like instructing to use file references instead). This is a minor tip, but it leverages the fact that BMAD itself is partly about educating users in the method. Since you're integrating with LLMs, sometimes asking the LLM (in orchestrator mode) how to integrate with VS Code or how to optimize prompts can yield good advice.

- **Maintain the Story-Centric Discipline:** It might be tempting occasionally to just ask ChatGPT something in a single ad-hoc conversation (outside the BMAD format). For example, you might think "Let me quickly ask ChatGPT to write a snippet of code" outside the story context. While that's fine for brainstorming, try to channel everything through the structured flow when it comes to actual development changes. The reason is that the story-centric flow leaves an audit trail and ensures consistency [3]. If you start coding outside of it, you might forget to update docs or the AI won't remember those changes in the next cycle. With BMAD, each story is documented, so any code written is accounted for. Of course, you as a human can always code directly too (and you should, if something is quicker to do manually or too sensitive to trust to AI, like handling secure credentials). But then consider writing a brief note in the story or PRD about the manual change so that the context remains up-to-date for the AI.

- **Combining Claude and ChatGPT Strengths:** Since you have the option of Claude or ChatGPT, you might use both for what they're best at. For example:

- Claude is excellent at digesting lots of info and producing structured, thoughtful outputs. You might prefer it for the **Analyst and Architect roles** where context is heavy and the output is narrative/analytical. Claude also tends to be very good at QA reasoning due to its larger context (can hold the whole code and spec).
- ChatGPT (GPT-4) is extremely good at **coding** and adhering to specific formats. You might prefer GPT-4 for the **Developer role**, especially if you have it integrated in VS Code (Copilot Chat uses GPT-4), because it will produce clean code and often can do it more interactively (suggesting code completions as you type).

- There's no harm in this hybrid approach. You'd ensure that both share the same documents. For instance, after Analyst (Claude) writes PRD, you use GPT-4 as Developer to code; GPT-4 will see the PRD via the story file or loaded context, and it's fine.
- Be mindful that the style might differ slightly; but as long as requirements are clear, both should converge on the same functionality.

- If using two models, you essentially act as the go-between orchestrator (since BMAD doesn't natively coordinate two different LLMs automatically). This is manageable in small projects: e.g., copy Claude's arch output into your VS Code, then let ChatGPT see it to implement.

- **Regular Checks and Balances:** Use the multi-agent nature to your advantage by having agents double-check each other's work. We did this inherently with QA checking Dev. But you can also have Architect review code from a design perspective after it's written, or have Analyst review the PRD vs the final product at the end to ensure all original requirements were met. For instance, after all stories are done, you could prompt the Analyst or PM agent: *"Cross-check the implemented features (as described in these story files or code summary) against the PRD. Are all requirements fulfilled?"* The AI might spot if anything was dropped inadvertently. This is akin to a **final acceptance testing** from a requirements perspective.

In summary, the synergy of BMAD + Claude/ChatGPT gives you a powerful development workflow: - You have **structured roles and processes** ensuring nothing falls through the cracks, - and you have **state-of-the-art AI** in each role bringing intelligence, consistency, and speed [24] [26].

By following this detailed plan, SenseQuant will be developed in a controlled yet efficient manner. The BMAD method will orchestrate the multi-agent development, and Claude/ChatGPT will provide the heavy lifting in each role. The end result will be a robust AI trading assistant that meets the user's needs for intraday and swing trading, all achieved with a clear engineering framework and preserved custom AI logic.

**Sources:**

- BMAD Methodology and Roles – Vishal Mysore, *"BMAD-Method: From Zero To Hero"* [1] [15]
- BMAD Planning & Story Workflow – BMAD Method v4 Documentation [4] [2]
- BMAD IDE/Web Integration – Zichen.dev, *"BMAD + Claude Code Setup Guide"* [22] [27]
- ICICI Breeze API Features – *Breeze-Python-SDK README* [20]
- Intraday vs Swing Trading – Bajaj Finserv, *"Day Trading vs Swing Trading"* [19] [29]
- Multi-Agent Dev Benefits – Reddit post by ABGDreaming [24] [25], Biju Joseph on Medium [26]

---

[1] [11] [12] [15] [28] BMAD-Method : From Zero To Hero. The BMAD-METHOD is a way of working… | by Vishal Mysore | Sep, 2025 | Medium
https://medium.com/@visrow/bmad-method-from-zero-to-hero-1bf5203f2ecd

[2] [3] [4] GitHub - bmad-code-org/BMAD-METHOD: Breakthrough Method for Agile Ai Driven Development
https://github.com/bmad-code-org/BMAD-METHOD

[5] [6] [7] [8] [9] [10] [13] [14] [16] [17] [18] [22] [23] [27] BMAD + Claude Code Setup Guide (IDE & Web Chat) – ZICHEN.DEV
https://zichen.dev/bmad-claude-code-setup-guide-ide-web-chat/

[19] [29] Swing Trading vs. Day Trading: What's the Difference?
https://www.bajajfinserv.in/swing-trading-vs-day-trading

[20] [21] GitHub - Idirect-Tech/Breeze-Python-SDK: The official Python client library for the ICICI Securities trading APIs
https://github.com/Idirect-Tech/Breeze-Python-SDK

[24] [25] How I Built a Multi-Agent Orchestration System with Claude Code Complete Guide (from a nontechnical person don't mind me) : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1l11fo2/how_i_built_a_multiagent_orchestration_system/

[26] Building an Agile Multi Agent Squad: This is the future of software development with Claude Code multi-agent orchestration | by BIJU JOSEPH | Medium
https://medium.com/@themoonwalker/building-an-agile-multi-agent-squad-this-is-the-future-of-software-development-with-claude-code-ec73f6ff758d