

Social Swarm + NVIDIA

a full-stack Python RAG and swarm system integrating Astra DB, Langsmith RAG, X AI, and Hume, along with a high-performance NVIDIA NIM agent leveraging BrewLaunchable and your Orin Nano.


This system will include:

1. **RAG Architecture:** Retrieval-Augmented Generation with structured blockchain data, multi-modal AI, and intelligent knowledge retrieval.
2. **Swarm Agents:** Social AI-driven agents with deep integration into Web3 infrastructures like Cheshire Terminal, Solana/Base, and zk-proof verification.
3. **NVIDIA NIM Agent:** Optimized AI compute leveraging your Orin Nano and BrewLaunchable for high-speed inference.
4. **Virtuals Plugin:** A Python and TypeScript plugin for easy interaction between the system and on-chain action.
5. **On-Chain Execution:** Enabling transactions and decentralized AI processing within blockchain environments.
6. **Game SDK Integration:** AI-enhanced decision-making and real-time interactions within games.
7. **Branding & AI Personality:** Keeping the 'based' theme while ensuring zk-proof verification and high-performance execution.

I will get started on designing the architecture, implementation details, and necessary components. I'll update you once the research and implementation details are ready.

Full-Stack RAG & Swarm System Architecture

1. Structured Data, Multi-Modal AI & Knowledge Retrieval

Knowledge Base Integration: At the core is a **Retrieval-Augmented Generation (RAG)** pipeline that leverages structured data from blockchains and other sources. We use **DataStax Astra DB** as a unified database for both structured storage and vector search. Astra DB (built on Apache Cassandra) supports storing embeddings and performing similarity queries natively ([Astra DB Vector Store](#) |  [LangChain](#)). This lets us store **structured blockchain data** (e.g. transaction logs, contract events) alongside embeddings for unstructured data (like social posts or documentation). A **knowledge base** in Astra will hold historical on-chain events, tweets, and relevant documents, indexed by embeddings for semantic retrieval ([Web 3 Social Swarm](#) | [Default Page](#)) ([Web 3 Social Swarm](#) | [Default Page](#)).

Retrieval-Augmented Generation: When an AI agent needs information, the system performs a vector similarity search on Astra DB to fetch relevant context (e.g. past transactions, social sentiments, or wiki knowledge). This context is fed into the language model prompt to **augment the model's knowledge**. RAG enhances the performance of the LLM by grounding its answers in up-to-date external data ([Evaluate a RAG application](#) | [LangSmith](#)). We can implement this using **LangChain** or a similar framework, with LangSmith for tracing and evaluation of the RAG pipeline. LangSmith provides tooling to test that the retrieved context indeed improves answer accuracy ([Evaluate a RAG application](#) | [LangSmith](#)).

Multi-Modal AI Interaction: The system supports **multi-modal inputs and outputs**. Textual data (on-chain records, chat messages) is handled via the LLM, while other modalities are processed by specialized models. For example, we integrate **Hume AI's Empathic Voice Interface (EVI)** to handle voice interactions. Hume's platform provides real-time voice intelligence – it can interpret a user's tone and generate expressive speech in various styles ([Home • Hume AI](#)) ([Home • Hume AI](#)). This means a user could speak a query; the audio is converted to text (via ASR like Whisper), passed through the RAG+LLM, and the response is spoken back in a chosen persona's voice via Hume's TTS. For images, an agent can use a computer vision model – e.g. if we incorporate an AI art module, the agent might retrieve an image embedding from Astra or generate an image (using a model like Stable Diffusion) based on the query. All these modes (text, voice, image) are unified in the knowledge retrieval layer so that the AI can intelligently cross-reference between modalities. For instance, an agent could recognize an NFT image, look up its metadata on-chain, and explain its context verbally.

Intelligent Retrieval Logic: Intelligent orchestration decides which modality to use for a given task. If a question requires on-chain facts, the agent queries the **blockchain data index**; if it involves an image (e.g. “what does this chart indicate?”), the agent uses a vision model. This flexible retrieval ensures the AI always uses the right tool for the job. The knowledge base can include a **knowledge graph or relational component** for highly structured queries (like relationship between wallet addresses), and a **vector index** for semantic searches on documents or code. By combining these, the agent can answer complex queries that involve factual accuracy (from structured DB data) and reasoning or explanation (via the LLM). The result is a system that **grounds AI responses in verified data** while allowing rich multimodal interaction.

2. Social Swarm Agents & Web3 Integration

Swarm of Agents: The platform employs a **social swarm** of AI agents that collaborate and communicate to monitor data and interact with users. This is inspired by the “Web3 Social Swarm” concept – a multi-agent ecosystem where agents aggregate social media trends and on-chain events in real time ([Web 3 Social Swarm | Default Page](#)). In our design, each agent has a specialized role but they share information via the common knowledge base (“agentic memory”). For example, one agent could be focused on **social data** (listening to Twitter/X for crypto trends or alpha), while another focuses on **blockchain analytics** (monitoring Solana and Base for notable transactions). A lead agent (call it **AlphaAgent**) could curate insights and post updates to a dashboard or social feed, while a contextual agent (**CheshireGPT**) replies with deeper analysis, historical context, or code insights ([Web 3 Social Swarm | Default Page](#)). This echoes how one agent might announce a prediction and another enriches it with context and caveats. The swarm architecture ensures **decentralized intelligence** – no single agent has to do everything, they can swarm on a problem with each contributing from its domain of expertise.

Cheshire Terminal (User Interface): All agent interactions funnel through the **Cheshire Terminal**, a unified front-end for users. This could be a web dashboard that shows real-time insights, trending topics, and agent outputs in a user-friendly way. The Terminal might feature a “swarm trending board” that displays metrics from both social media and on-chain data side by side (e.g. trending token mentions vs. on-chain volume spikes) as the agents report them. Users can chat

with the swarm (text or voice) via this interface, asking questions or requesting analyses; the query is dispatched to the appropriate agent or triggers a collaborative response from multiple agents.

Web3 Integration: The system deeply integrates with Web3 infrastructure to both **read and write** blockchain data:

- **On-Chain Data Monitoring:** Agents connect to blockchain nodes or APIs (e.g. via Infura/Alchemy for Base (Ethereum L2) and a Solana RPC for Solana). They subscribe to relevant events such as large token transfers, new contract deployments, or NFT mints. For example, one component might track **whale transactions** and feed that info into the knowledge base ([Web 3 Social Swarm | Default Page](#)). Significant events (like liquidity movements or governance proposals) become triggers for the swarm to discuss or alert on. This structured blockchain data is parsed and stored (both raw and as semantic embeddings) for later retrieval.
- **Transactions & On-Chain Actions:** Beyond reading data, agents can also **initiate on-chain transactions** when appropriate. For instance, an agent might automatically execute a trade on Solana or an interaction on an Ethereum smart contract if it's part of its role (with safeguards). This requires secure key management – likely each agent has an associated Web3 wallet. We incorporate **Privy authentication** for user and agent key management. Privy is an authentication/key management platform that allows linking traditional accounts with Web3 wallets securely ([Privy – All your users, onchain.](#)). Using Privy, the Cheshire Terminal can onboard users via email or OAuth and transparently create or link a non-custodial wallet for them. This means users log in easily while still getting an on-chain identity for interacting with the agents or receiving rewards.
- **User Authentication & Identity:** With **Privy** (and possibly augmented by **Dynamic.xyz** for wallet orchestration), users of the platform are verified and can grant the system permission to act on their behalf on-chain (if needed). For example, a user might ask an agent to execute a swap for them; the agent, through Privy, can prompt the user to sign the transaction securely. All sensitive user data and keys remain end-to-end encrypted and secure via Privy's vault, aligning with the need for privacy and safety in Web3 apps ([Privy – All your users, onchain.](#)).

Social Media Integration: The swarm's social agents connect to platforms like Twitter (X) via API to both consume and post content. One agent might live-stream tweet analysis: using the Twitter API's filtered stream or Account Activity API to catch specific keywords or influencer tweets in real time ([Web 3 Social Swarm | Default Page](#)). Another agent can perform **sentiment analysis** on these tweets (using an NLP model) to gauge market mood instantly ([Web 3 Social Swarm | Default Page](#)). The agents can then post aggregated insights on the Terminal or even tweet out through a designated account (like how [@aixbt_agent](#) and [@cheshiregpt](#) coordinate in the social swarm example ([Web 3 Social Swarm | Default Page](#))). All posts made by agents are also stored in the knowledge base so they can refer back to what was said (enabling accountability and memory of promises or predictions).

Coordination: These social and on-chain agents communicate through the shared database or an internal pub-sub event bus. For example, when the on-chain agent detects a spike in a token's volume, it can emit an event that the social agent picks up to see if the token is trending on Twitter as well. The result might be a combined insight like "Token XYZ is spiking in trading volume on Solana and trending on Twitter – possible announcement incoming." The **intelligent retrieval** ensures each agent can query the full context (recent tweets, historical events, code from GitHub, etc.) to enrich its output ([Web 3 Social Swarm | Default Page](#)). This way, even though agents are specialized, they are not siloed – the **swarm intelligence** emerges from their collaboration and shared memory.

3. NVIDIA NIM Agent for High-Speed AI Compute

Edge Compute Node: We deploy a specialized agent on the user's **NVIDIA Jetson Orin Nano** device to leverage local GPU compute for AI tasks. This **NVIDIA NIM (Neural Inference Module) agent** is optimized to run heavy models at high speed, offloading work from the cloud and reducing latency. The Orin Nano, equipped with an NVIDIA GPU and CUDA cores, is well-suited for running optimized neural networks (e.g. running a medium-sized LLM or image generator locally). By utilizing the Orin for on-device inference, the system can perform certain tasks **offline or in real-time** without always calling out to cloud APIs. For instance, the NIM agent could run a local instance of a transformer model for quick replies or execute stable diffusion to generate game graphics or NFT art on demand.

BrewLaunchable Deployment: To integrate the Orin Nano into the overall system seamlessly, we package the NIM agent as a **containerized service** that can be launched via a one-command deployment (the notion of “BrewLaunchable” implies an easy launch, akin to Homebrew formulas or Docker images). Essentially, the user’s Orin Nano will run a lightweight server (possibly a FastAPI or gRPC server in Python/C++) that listens for tasks from the main system. When the swarm needs a high-compute task done, it dispatches it to the NIM agent. For example, if the user requests an AI-generated image or a lengthy piece of text analysis, the request is queued to the Orin service, which then processes it using its GPU and returns the result. Containerization ensures that all necessary libraries (CUDA, TensorRT, model weights, etc.) are pre-installed, making the agent **immediately launchable** on the Orin device without complex setup.

Optimizations for Speed: We optimize models for the Orin Nano using NVIDIA’s tooling. Heavy models are converted with **TensorRT**, which accelerates inference by optimizing the neural network graph for the GPU ([Maximizing Deep Learning Performance on NVIDIA Jetson Orin with ...](#)). We may quantize models (e.g. 8-bit or 4-bit quantization) to fit within the Orin’s memory while still maintaining acceptable accuracy. The NIM agent is essentially a mini AI server: for instance, it could host a local LLM (like a fine-tuned LLaMA-2) to handle user queries locally, avoiding external API calls. This is especially useful for privacy (user data can be processed on their own device) and for reducing latency in interactive applications (like a game, where quick AI response is needed).

Integration with Swarm: The Orin-based agent registers its capabilities to the swarm network. Other agents know that “if task == image_generation or large_LLM, send to NIM agent”. The main orchestrator can use an async job queue to delegate to Orin and wait for results. We ensure robust fallback: if the Orin is offline, tasks can default to cloud services (e.g. use a cloud GPU or API). Conversely, when online, the local agent can take over certain duties completely – for example, it might continuously run a computer vision model analyzing game frames or user facial expressions (if the use case involves AR/VR), providing another stream of multi-modal input to the system. By **leveraging edge computing**, the system attains high throughput for AI tasks and reduces cloud dependency, which is cost-effective and empowers the user’s hardware (especially relevant since the user specifically has an Orin Nano available for this purpose).

4. Virtuals Plugin for On-Chain Actions & Decentralized Execution

Virtuals GAME Framework: We integrate the **Virtuals Protocol's GAME SDK** to handle on-chain actions and to structure the agents' decision-making in a modular way. Virtuals GAME (General Autonomous Modular Environment) provides a framework where an agent can be given a goal, a set of available actions (functions), and it will plan and decide which action to execute based on current context ([GAME Framework | Virtuals Protocol Whitepaper](#)) ([GAME Framework | Virtuals Protocol Whitepaper](#)). This fits perfectly for enabling **on-chain operations**: we can define custom plugin "actions" that correspond to Web3 behaviors (e.g. *transfer tokens, execute swap on DEX, mint NFT, post to blockchain feed*). The agent's cognitive core (LLM-driven reasoning) will choose among these actions when appropriate, effectively allowing the AI to **act in the blockchain world** autonomously.

Python & TypeScript SDK: The Virtuals GAME SDK is available in both Python and TypeScript ([GAME SDK | GAME OS](#)). We can leverage the Python SDK on the backend to integrate the agent logic with our existing Python-based RAG system. For instance, using the GAME Python SDK, we configure an agent with certain **abilities** (plugins) and a persona. One plugin might be "SolanaTransaction" which we implement to interface with a Solana RPC (using `solana.py` under the hood), another could be "BaseContractCall" using `web3.py` or ethers. The Virtuals framework allows these actions to be added seamlessly as first-class options the agent can consider ([GAME Framework | Virtuals Protocol Whitepaper](#)). Meanwhile, if the front-end or a Node service needs to trigger agent behavior (say, to run in a browser or a Discord bot), the TypeScript SDK can be used to invoke the same agent logic or communicate with it. This cross-language availability ensures **seamless interaction between on-chain components and the AI logic** in whichever environment.

On-Chain Plugins: Using Virtuals' plugin system, we incorporate **out-of-the-box on-chain functions** and also develop custom ones ([GAME Framework | Virtuals Protocol Whitepaper](#)). For example:

- An **on-chain trading plugin** that allows the agent to execute trades or liquidity provision on Uniswap (Base) or Orca (Solana) when certain market conditions are met. The agent would decide to call this function if its analysis (from the

knowledge base and price feed) indicates an arbitrage or a rebalancing opportunity.

- A **wallet interaction plugin** that lets the agent manage a crypto wallet (for an NPC or the user). It could send tokens, check balances, or interact with NFT contracts. Through Privy's SDK, the plugin can request the user's signature when needed, maintaining security.
- A **cross-chain bridge plugin** if the agent needs to move assets between Solana and an EVM chain as part of its logic (this could use a bridging API or protocol).
- The plugin architecture is modular, so future capabilities (like image generation or external API calls) can be added as needed, and even community-contributed, per Virtuals design ([GAME Framework](#) | [Virtuals Protocol Whitepaper](#)).

Decentralized AI Execution: By integrating Virtuals, we also open the door to **decentralized execution** of the AI agents. The Virtuals protocol envisions that agents can be deployed in a way that multiple nodes or community members can contribute to their functioning (for example, an agent's decision process could be verified or even run by a network of token holders). In our system, while the core might run on our servers and user's devices, critical actions could be verified on-chain or by a decentralized oracle network to ensure trust. For instance, if the agent decides to execute a high-value transaction, we might require a smart contract on Base to verify a certain condition (like agent's token stake or a multisig approval) before execution – essentially a **governance or verification step** that aligns with decentralized principles. Virtuals also allows launching an "Agent Token" which could be used to distribute ownership or rewards related to an agent's performance ([AI Agents with Crypto Superpowers](#) | [by Henri.M - district0x Updates](#)), adding another layer of decentralization (though this is an optional feature of the ecosystem).

Connecting to Cheshire Ecosystem: The Virtuals plugin is how our AI swarm connects to the **Cheshire (Based) ecosystem** on-chain. The Cheshire smart contracts or identity system might require certain proofs or calls which the agent can carry out via these plugins. For example, if Cheshire Terminal has a contract for logging "verified AI insights" on-chain (as NFTs or events), the agent can trigger that through a plugin, imprinting important analyses immutably on the

blockchain for transparency. This way, the **knowledge retrieval and generation is not a black-box** – key outcomes get recorded on-chain, anchored to the Cheshire ecosystem's contracts.

5. AI Decision-Making in Gaming Environments (Game SDK Integration)

Game Environment Integration: The system is designed to inject AI-driven decision-making into gaming environments using the provided Game SDK (which in our case is the Virtuals GAME SDK, but it could also extend to other game engines). This means our AI agents can function as NPCs or game masters within a video game or metaverse, making real-time decisions that affect gameplay. Using the Game SDK, we can bind the agent's **perception** to the game state and its **actions** to game APIs:

- The agent's **state input** can include the game world status (positions of players, events happening in-game, inventory, etc.) similar to how it would take social or blockchain context. We translate game state into a context that the agent understands (this might involve prompting the LLM with a description of the current scene or numeric state fed as part of the agent's observation).
- The agent's **available actions** are defined by the game's SDK. For example, in a MMORPG, actions might be "move", "speak dialog", "attack", "trade item". In a strategy game, it could be higher-level decisions like "allocate resources to build X". We register these functions in the agent via the Game SDK so the decision engine knows it can choose them.

Real-time Decision Loop: Within the game loop, the AI agent functions as an **autonomous character**. Each tick or at key events, the game calls the agent (or the agent polls the game state) to decide what to do next. The Virtuals GAME framework's decision engine (backed by an LLM) will take into account the agent's goal/personality and the current context to output an action ([GAME Framework | Virtuals Protocol Whitepaper](#)). For example, if we have a *DeFi trading card game*, an AI opponent agent could analyze both the on-chain data (maybe card NFTs on chain) and the player's moves to decide its next move, all while role-playing a certain persona.

Use of Knowledge Base in Games: Because our knowledge retrieval is robust, the agent in the game can tap into external knowledge to enrich gameplay. Imagine a puzzle game where the AI NPC gives hints: the NPC could actually pull from a database of riddles or use the LLM to generate hints based on a vast trove of info. Or in a **crypto treasure hunt** game, the AI could verify a clue on-chain (using its blockchain access) and then guide the player. The **multi-modal capability** also shines here – the agent can speak to the player with a unique voice (via Hume’s voice models), show images or maps when needed, or even present on-chain evidence for a clue. This creates a highly immersive experience where the game’s AI is not scripted with fixed dialogs, but is **dynamic and data-driven**.

Example – Virtual Guide NPC: Consider an in-game guide character in a metaverse that helps new users. This NPC agent can access the player’s on-chain profile (what NFTs they hold, etc.), social media trends (to see what the community is up to), and then tailor its guidance. It might say via voice, *“I see you own an NFT sword that was forged on Solana last week – it’s currently trending in the arena battles. You should try it out in the Coliseum event happening now!”*. The agent arrives at this by retrieving on-chain data (NFT metadata), checking social feed (trending arena battles), and game state (player’s inventory), all through our RAG system, then deciding on an action (dialogue) through the Game SDK. This showcases the **power of AI-driven decision-making in games**: the game world responds to real-world data and the player’s context in a believable, intelligent manner.

Provided Game SDK: Since the question mentions a “provided Game SDK,” it implies we have a specific integration point. In our solution, that is the Virtuals GAME SDK (open-sourced for Python/TS). If the gaming platform has its own SDK (for example, a Unity plugin or Unreal mod), we would create a bridge between that and our AI system. Typically, a thin integration layer could expose game events to a Python service (using websockets or RPC) and in return execute AI-suggested actions in the game engine (Unity could call a REST API that our agent service provides for decision-making, for instance). The implementation plan would cover adapting our agent outputs to the game’s API. The key point is that our architecture is flexible: **the same AI brain that operates in Web3 and social domains can also live inside a game world**, thanks to modular design and the Game SDK abstraction.

6. "Based" ZK-Proof Verification and Cheshire Branding

ZK-Proof Verification: Security and trust are paramount in the Cheshire ecosystem. We incorporate **zero-knowledge proof (ZKP) techniques** to verify critical aspects of agent interactions without exposing sensitive information. In practice, this could mean using zk-SNARKs or similar to prove statements like "Agent X's recommendation is based on authentic data" or "User completed quest Y legitimately" without revealing the underlying private data. For example, if a reward is given when a user achieves some on-chain milestone, the system can verify that milestone with a ZKP rather than requiring the user to divulge all their wallet details. *Zero-knowledge proofs allow one party to prove to another that an on-chain statement is true without revealing anything beyond the truth of the statement ([Zero-Knowledge Proofs: The Magic Key to Identity Privacy - Galaxy](#)).* In our architecture, an agent generating a crucial piece of analysis could attach a ZK-proof that it followed the approved algorithm (preventing any rogue behavior from a compromised agent). Similarly, when the user authenticates via Privy and completes an on-chain action, a ZK proof could attest to their eligibility for a reward in a privacy-preserving way. These proofs reinforce trust in the AI outputs and the system's integrity, aligning with the ethos of decentralization and privacy.

"Based" Cheshire Branding: All AI agents will maintain the **Cheshire ecosystem's branding and identity**, notably the "'based' zk-proof verified" style that has been cultivated. In practical terms, this involves both **visual branding** and **behavioral consistency**:

- Agents present themselves with the Cheshire persona (e.g. using the Cheshire cat or grin logo in the Terminal UI, and perhaps a witty, community-favorite tone in communications). Any user-facing messages can include a small tagline or watermark indicating it's "Based & Verified by Cheshire" to assure users the content is authentic and approved.
- Before an agent publishes any significant insight or executes an on-chain transaction on behalf of the platform, the action is stamped with a form of verification. This could be a digital signature using the Cheshire DAO's private key or a cryptographic proof logged on-chain. Essentially, the ecosystem's **zero-knowledge verification badge** is attached, signaling that the

action/result has been verified according to Cheshire's standards (either via formal proofs or multi-sig validation).

- The **“based” terminology** likely alludes to the platform's use of Coinbase's Base chain as well as meaning “grounded”. We ensure that wherever relevant, interactions with the Base L2 are prominent and efficient (since Base is part of the branding). For instance, the Cheshire Terminal might primarily operate on Base for any L2 transactions, taking advantage of lower fees, and using its finality to store important records. The zk-proof approach complements this by potentially using L2-friendly ZK-rollup tech for verification of off-chain computations.

Trust and Community: By baking in ZK verification and maintaining a strong brand persona, we foster user trust and community engagement. Users can trust that an insight labeled as “Cheshire verified” has indeed been cross-checked (maybe the agent's reasoning was verified by another agent or by a cryptographic routine). The branding also helps in marketing – the **Cheshire AI** comes across as a reliably “based” (grounded in facts) assistant in the Web3 and gaming space. Technically, this means we might integrate libraries like zk-SNARK circuits (e.g. using SnarkJS or Halo2 library in Rust, invoked via Python bindings) for generating proofs, and smart contracts on Base to verify them. Those contracts could manage a registry of verified agents/actions – only content with a valid proof is considered “official”. This creates a **feedback loop of trust**: the more the agents act and prove their fidelity, the more users trust and engage with the system.

Implementation Plan

1. **Data Ingestion & Storage:** Set up pipelines to ingest data into Astra DB. This includes blockchain data (using providers like Alchemy for Base, and Solana RPC or The Graph for Solana) and social data (Twitter API streams). Define Cassandra tables for structured data (e.g. transactions with fields like hash, from, to, amount, timestamp) and for unstructured data (e.g. a table for documents or tweets with text and metadata). Enable vector search on relevant tables (Astra allows adding a vector column and indexing it for similarity search ([Astra DB Vector Store](#) | 🦜 LangChain)). Ingest historical data and periodically update with new events.

2. **Embedding & Indexing:** Implement a service to generate embeddings for incoming data. For text, use a pre-trained model (like SentenceTransformer or OpenAI Embeddings) to vectorize tweets, descriptions, and even code snippets. For any images (if storing NFT images or others), use a vision model (like CLIP) to get image embeddings. Store these vectors in Astra alongside the identifiers. This service runs continuously so that the knowledge base stays up-to-date for retrieval.
3. **LLM Integration (X AI or Equivalent):** Choose and integrate a core large language model that will power the agents' reasoning and generation. If X AI (Elon Musk's AI) provides an API or model (e.g. the rumored "Grok" model), integrate that; otherwise use an open model (GPT-4 via OpenAI API, or Llama-2 locally on Orin). Wrap the LLM with LangChain to facilitate passing retrieved context. Test the RAG loop: given a sample query, ensure the system properly retrieves from Astra DB and the LLM incorporates it into the answer. Use LangSmith to trace these calls and verify that relevant data is being used (improving the prompt or retrieval parameters as needed).
4. **Backend Services & Orchestration:** Develop the Python backend as a collection of services:
 - An API service (FastAPI or Flask) that serves the Cheshire Terminal frontend and provides endpoints for queries, agent status, etc.
 - Agent daemons: processes or async tasks for each agent (social agent, on-chain agent, etc.). For example, a Twitter listener agent that pushes new tweets into the DB and alerts the main system, and an analysis agent that periodically scans for notable events and generates summaries.
 - The **Orin NIM agent service:** running on the Jetson device, but accessible over the network. You can use MQTT or gRPC for lightweight messaging between the cloud and the Orin. For instance, when the main server needs an image generated, it sends a message to Orin's queue; the Orin does it and returns a URL or binary.
 - A scheduler (could be Celery beat or APScheduler) to schedule regular tasks like "daily market summary" generation or routine cleanup.
5. **Web3 Connectivity:** Integrate Web3 libraries:

- For Base (EVM chain): set up `web3.py` or use ethers.js via a small Node microservice. Write functions to fetch on-chain info (balances, events) and to send transactions (e.g. using a provided private key or via user signature with MetaMask integration on the front-end).
 - For Solana: use the `solana-py` SDK or RPC calls. Implement listeners for specific account addresses or parse transaction feeds (Solana has webhooks services or just poll the RPC for certain signatures).
 - Test reading from and writing to both chains in a dev environment. For writing (transactions), integrate Privy: use Privy's SDK to get the user's wallet keys or to trigger signature requests. Ensure that for any automated agent wallet, keys are stored securely (in an HSM or Vault, or using a multi-sig scheme if high risk).
6. **User Authentication & Portal:** Implement the authentication flow with Privy. On the Cheshire Terminal front-end (likely a React/Next.js app), install Privy's SDK to handle login. When users log in, retrieve their decentralized identity and any profile info. Use this identity to customize their view (e.g. show their linked wallets, let them select which wallet an agent should use for actions). The back-end verifies JWTs or tokens from Privy to establish sessions. Also integrate additional OAuth if needed (for Discord or Twitter linking, if users want to authorize the agents to post on their behalf, etc.).
7. **Front-End Development:** Build the Cheshire Terminal UI in TypeScript (Next.js or similar). This UI will have components like:
- **Dashboard:** showing real-time data (market charts, trending topics from the swarm). Use webSockets or Server-Sent Events from the backend to push updates (for low-latency updates when agents find new info).
 - **Chat/Console:** where the user can converse with the AI agents (text input or microphone input for voice). This interacts with the backend API to get responses (possibly streaming responses for immediacy).
 - **Control Panel:** allowing the user to deploy or configure swarm agents (maybe an interface to turn on/off certain agents, or to feed in custom data).

- Make it responsive and intuitive, using modern UI frameworks. Emphasize the Cheshire branding (colors, logos) and show when content is verified (perhaps a shield icon indicating zk-proof verified).
8. **Virtuals GAME Integration:** Now implement the higher-level agent logic using the GAME SDK:
- Write an agent config (could be in YAML or code as per Virtuals SDK) for each agent. Define their goals and descriptions (e.g. SocialAgent: “monitor crypto Twitter and provide timely alpha”), and list the actions they can take. Many actions will map to functions we have already implemented (like `post_tweet(content)`, `execute_transaction(tx)`, `speak_to_user(message)`).
 - If using the Virtuals Python SDK, this might involve creating a subclass or registering functions. If the logic proves easier in TypeScript (for example, if the game environment is easier to hook in TS), consider using the TS SDK for those parts or even running a Node service for the agent reasoning.
 - Test the agent decision-making in isolation: simulate a scenario (e.g., a spike in price) and see if the agent chooses the correct actions (perhaps posting an alert and doing a trade). Virtuals SDK likely has a way to run the agent in a loop where it assesses state and picks an action; use that to verify the flow.
 - Integrate the GAME agent loop with our event system: when new data comes in or a user query arrives, feed it into the GAME agent as updated state so it can decide if an action is needed.
9. **Gaming Use-Case Integration:** If there is a target game environment:
- Connect the game’s SDK or API with our system. For example, if the game is in Unity, use Unity’s Web Request to call our backend’s agent decision endpoint every few seconds or on certain triggers (like player enters area → ask AI for a response). Or if the game is web-based, use the Virtuals TS SDK directly in the game client to embed the agent.
 - Create a **mock game environment** for testing, if needed (some simplified environment data) to ensure the AI behaves as expected. Tune the agent’s

prompt/persona to fit the game theme (for instance, make the agent more role-playful in a fantasy game, or more formal in a sci-fi game).

- Ensure performance is adequate – in a game, we may need to limit LLM response time or use a smaller model to meet real-time constraints. Possibly leverage the Orin agent for any local computation in an AR/VR device scenario.

10. **ZK-Proof and Security:** Implement the zk-proof mechanisms for critical flows:

- If using a known library, set up the circuits and proving environment. For example, use snarkJS for a simple demo: prove that a number is prime without revealing the number (just conceptually to test). Then move to actual use cases, e.g., a circuit that takes a user's wallet and an achievement and proves the user has that achievement NFT without revealing which user (for anonymous leaderboards).
- Alternatively, for a simpler approach initially, use digital signatures: each agent could have an Ethereum key and sign its messages. The public key tied to Cheshire is known, so users can verify the signature off-chain. This can later be complemented with true ZK proofs.
- Harden the security of key management: integrate an HSM or at least ensure Privy's custody of keys is used so the backend never exposes private keys in plaintext.
- Run penetration tests or audits, especially on the on-chain transaction flow and the zk-proof verification contracts.


11. **Testing & Iteration:** Test each component thoroughly:

- Unit test the data retrieval (does the query to Astra bring the right context?).
- Simulate user queries to the RAG pipeline and verify correctness of answers (perhaps comparing against known truths).
- Test multi-modal: speak a query via microphone and ensure the pipeline from speech-to-text to answer to speech output works end-to-end.
- For the swarm agents, simulate various scenarios (bull market tweets vs. bear market, or different game scenarios) and observe if agents

coordinate properly. Adjust their prompts or the shared memory logic if needed to avoid redundant or conflicting actions.

- Use LangSmith's evaluation capabilities to gather feedback on LLM outputs (measure relevance/accuracy of answers ([Evaluate a RAG application](#) | [LangSmith](#))).
 - Involve beta users or team members to try the Cheshire Terminal and report issues or confusion. Use their feedback to refine the UI/UX and agent behaviors.
12. **Deployment:** Deploy the backend on a cloud platform (could be AWS EC2 or a container on Kubernetes, etc. – Astra DB is serverless so just ensure connectivity and proper Astra credentials in env). Deploy the front-end (perhaps on Vercel if Next.js). Deploy the Orin agent on the physical device (it might auto-start on device boot and connect back to the cloud). Set up monitoring for all agents (logs, errors) and use a tool like LangSmith or custom dashboards to monitor agent decisions and outcomes in production.
13. **Maintenance and Future Enhancements:** Once live, maintain the system by updating models (e.g. incorporate X AI's latest model if available), adding new data sources (maybe other social platforms or more chains like Arbitrum, etc.), and improving the ZK verification (aim to make more and more of the process trustless with time – possibly even move some agent logic on-chain in simplified form). Also, consider community features: since it's a swarm, perhaps allow power users to run their own agent nodes that contribute data or analysis (similar to how folks might run their own instance of the agent connected via an API, crowd-sourcing intelligence). The architecture is designed to be modular and extensible, so new plugins (say a **DAO governance plugin** for the agent to propose/vote in DAOs, or a **DeFi lending plugin** to integrate Aave) can be added with minimal changes to the core.

Recommended Frameworks and Tools

- **Database and Vector Store:** **DataStax Astra DB** for a scalable serverless DB with Cassandra reliability and built-in vector search ([Astra DB Vector Store](#) |  [LangChain](#)). This avoids managing separate databases for structured vs.

semantic data. Alternatives: Weaviate or Pinecone for pure vector search (if not using Astra's), and PostgreSQL for structured data if needed.

- **LLM and RAG Orchestration: LangChain** (Python) for chaining the retrieval and generation steps, and managing prompts. **LangSmith** for observability (tracking queries, responses, and evaluating success metrics in the RAG pipeline). The LLM can be OpenAI's GPT-4 (for best quality) or an open model via Hugging Face Transformers (like Llama-2 13B for local running). If Elon Musk's **xAI** releases a model or API, that could be integrated here as an LLM choice. We will also use smaller models for support tasks: e.g. sentiment analysis (perhaps a HuggingFace model fine-tuned for finance sentiment) and embedding models (e.g. **all-MiniLM** for quick embeddings if OpenAI embedding API is too slow or costly).
- **Multi-modal AI: Hume AI** platform for voice - it provides APIs/SDK for TTS and emotion recognition which we'll use for the empathic voice feature ([Home • Hume AI](#)). For speech-to-text, OpenAI Whisper or Google Cloud Speech can be used. For image generation, **Stable Diffusion** (with a custom LoRA model if we train on specific art style, as hinted by the Cheshire art engine) running on the Orin Nano with optimizations. For image analysis (like reading blockchain QR codes or analyzing charts), OpenCV combined with a lightweight CNN could be used.
- **Web3 and Blockchain: Privy SDK** for authentication and wallet management in the dApp ([Privy – All your users, onchain.](#)). **Web3.py** for Ethereum/Base interactions (or ethers.js if using JS on that part). **Solana Python SDK (solana.py)** for Solana, or **Anchor** framework if needed to interact with Solana programs. We might also incorporate **Covalent** or **The Graph** for easier blockchain data querying (Covalent API can fetch historical transactions conveniently, TheGraph can get structured data from specific protocols). For on-chain monitoring, services like **Alchemy Notify** or **QuickNode webhooks** can simplify catching events without constant polling.
- **Agent Framework: Virtuals GAME SDK** (Python & TypeScript) for defining agent capabilities and autonomous decision logic ([GAME SDK](#) | [GAME OS](#)). This gives a structured approach to building agents that can plan actions (critical for the multi-action swarm and game NPC behaviors). We will also use Virtuals' **plugin library** if available for existing integrations (they might already

have a plugin for “tweeting” or “sending a transaction” which we can use or adapt ([GAME Framework](#) | [Virtuals Protocol Whitepaper](#))). If not using Virtuals for some reason, an alternative could be **LangChain's agents** (which allow an LLM to choose tools/actions in a similar manner) or **Haystack** for a custom agent loop, but Virtuals is tailored to Web3 use cases.

- **Backend & Infra: FastAPI** for the Python API server (fast and easy to define REST and WebSocket endpoints). **Docker** for containerization of services (especially the Orin agent – likely using NVIDIA's base images for CUDA support). **Kubernetes** or Docker Compose to manage multi-container deployments (we might have a container for the main API, one for each agent type if separating concerns, etc.). Use **Redis or NATS** as a message broker for internal pub-sub (for the swarm agents to talk to each other or to queue tasks for Orin). **Celery** could be used with Redis if we need a task queue (though for real-time, event-driven might suffice).
- **Frontend: Next.js (React)** for the web interface, as it supports TypeScript and is great for building interactive dashboards. We'll employ UI libraries like **Chakra UI** or **Tailwind CSS** for a clean, reactive UI. For charts/visualizations (displaying data trends), use **Plotly.js** or **D3.js** (since image embedding is off, we assume charts would be rendered on client side). The front-end will also utilize **Privy's React components** if available to streamline login, and perhaps **Web3Modal** or similar for wallet connections if users want to connect external wallets.
- **Communication: WebSockets (Socket.IO)** for real-time updates from backend to front (e.g., push new swarm findings to the dashboard immediately). **Discord bot framework** (like discord.py) if extending the social swarm to Discord communities as well. Possibly **Twilio** integration if we allow SMS or phone interactions (the Hume AI voice could even be used over a phone call to give verbal updates, which could be a neat feature for alerts).
- **NVIDIA Optimization: TensorRT and PyTorch with CUDA** on the Orin Nano for running models efficiently. NVIDIA's **JetPack SDK** will be used on the Orin to ensure we have all necessary drivers and libraries. We may also use **NVIDIA Triton Inference Server** on the Orin if we want to serve multiple models (it can optimize serving of multiple ML models behind gRPC/HTTP).

- **Security & ZK: Circom/SnarkJS** for generating zero-knowledge proofs in the browser or Node (if simple circuits for demo), and **Halo2 or ZoKrates** for more complex circuits. On the blockchain side, **Polygon ID or Mina** could be looked at for managed ZK identity solutions. To manage cryptographic keys and signatures, use **ethereumjs** libraries for signing data (for the agent signatures) and standard **ed25519** libs for Solana.

All these components together create a robust full-stack system. The combination of a Cassandra-based vector store, a swarm of specialized agents, high-performance local inference, and deeply integrated Web3 hooks (with modern authentication and ZK security) will fulfill the requirements. Each framework was chosen for being well-suited and proven: for example, Astra DB's vector search and LangChain make implementing RAG easier ([Evaluate a RAG application | LangSmith](#)), and Virtuals/Game SDK is explicitly designed for agent actions in Web3/gaming contexts. The end result is a **cutting-edge RAG-powered swarm intelligence platform** that bridges AI and blockchain, capable of driving rich user experiences from DeFi dashboards to immersive games – all under the **Cheshire "Based & Verified" banner**.

Great! I'll design a full-stack Python RAG and swarm system integrating Astra DB, Langsmith RAG, X AI, and Hume, along with a high-performance NVIDIA NIM agent leveraging BrewLaunchable and your Orin Nano.

This system will include:


1. **RAG Architecture:** Retrieval-Augmented Generation with structured blockchain data, multi-modal AI, and intelligent knowledge retrieval.
2. **Swarm Agents:** Social AI-driven agents with deep integration into Web3 infrastructures like Cheshire Terminal, Solana/Base, and zk-proof verification.
3. **NVIDIA NIM Agent:** Optimized AI compute leveraging your Orin Nano and BrewLaunchable for high-speed inference.
4. **Virtuals Plugin:** A Python and TypeScript plugin for easy interaction between the system and on-chain action.
5. **On-Chain Execution:** Enabling transactions and decentralized AI processing within blockchain environments.

6. **Game SDK Integration:** AI-enhanced decision-making and real-time interactions within games.
7. **Branding & AI Personality:** Keeping the 'based' theme while ensuring zk-proof verification and high-performance execution.

I will get started on designing the architecture, implementation details, and necessary components. I'll update you once the research and implementation details are ready.

Full-Stack RAG & Swarm System Architecture

1. Structured Data, Multi-Modal AI & Knowledge Retrieval

Knowledge Base Integration: At the core is a **Retrieval-Augmented Generation (RAG)** pipeline that leverages structured data from blockchains and other sources. We use **DataStax Astra DB** as a unified database for both structured storage and vector search. Astra DB (built on Apache Cassandra) supports storing embeddings and performing similarity queries natively ([Astra DB Vector Store | LangChain](#) ). This lets us store **structured blockchain data** (e.g. transaction logs, contract events) alongside embeddings for unstructured data (like social posts or documentation). A **knowledge base** in Astra will hold historical on-chain events, tweets, and relevant documents, indexed by embeddings for semantic retrieval ([Web 3 Social Swarm | Default Page](#)) ([Web 3 Social Swarm | Default Page](#)).

Retrieval-Augmented Generation: When an AI agent needs information, the system performs a vector similarity search on Astra DB to fetch relevant context (e.g. past transactions, social sentiments, or wiki knowledge). This context is fed into the language model prompt to **augment the model's knowledge**. RAG enhances the performance of the LLM by grounding its answers in up-to-date external data ([Evaluate a RAG application | LangSmith](#)). We can implement this using **LangChain** or a similar framework, with LangSmith for tracing and evaluation of the RAG pipeline. LangSmith provides tooling to test that the

retrieved context indeed improves answer accuracy ([Evaluate a RAG application](#) | [LangSmith](#)).

Multi-Modal AI Interaction: The system supports **multi-modal inputs and outputs**. Textual data (on-chain records, chat messages) is handled via the LLM, while other modalities are processed by specialized models. For example, we integrate **Hume AI's Empathic Voice Interface (EVI)** to handle voice interactions. Hume's platform provides real-time voice intelligence – it can interpret a user's tone and generate expressive speech in various styles ([Home • Hume AI](#)) ([Home • Hume AI](#)). This means a user could speak a query; the audio is converted to text (via ASR like Whisper), passed through the RAG+LLM, and the response is spoken back in a chosen persona's voice via Hume's TTS. For images, an agent can use a computer vision model – e.g. if we incorporate an AI art module, the agent might retrieve an image embedding from Astra or generate an image (using a model like Stable Diffusion) based on the query. All these modes (text, voice, image) are unified in the knowledge retrieval layer so that the AI can intelligently cross-reference between modalities. For instance, an agent could recognize an NFT image, look up its metadata on-chain, and explain its context verbally.

Intelligent Retrieval Logic: Intelligent orchestration decides which modality to use for a given task. If a question requires on-chain facts, the agent queries the **blockchain data index**; if it involves an image (e.g. "what does this chart indicate?"), the agent uses a vision model. This flexible retrieval ensures the AI always uses the right tool for the job. The knowledge base can include a **knowledge graph or relational component** for highly structured queries (like relationship between wallet addresses), and a **vector index** for semantic searches on documents or code. By combining these, the agent can answer complex queries that involve factual accuracy (from structured DB data) and reasoning or explanation (via the LLM). The result is a system that **grounds AI responses in verified data** while allowing rich multimodal interaction.

2. Social Swarm Agents & Web3 Integration

Swarm of Agents: The platform employs a **social swarm** of AI agents that collaborate and communicate to monitor data and interact with users. This is inspired by the "Web3 Social Swarm" concept – a multi-agent ecosystem where agents aggregate social media trends and on-chain events in real time ([Web 3](#)

[Social Swarm | Default Page](#)). In our design, each agent has a specialized role but they share information via the common knowledge base (“agentic memory”). For example, one agent could be focused on **social data** (listening to Twitter/X for crypto trends or alpha), while another focuses on **blockchain analytics** (monitoring Solana and Base for notable transactions). A lead agent (call it **AlphaAgent**) could curate insights and post updates to a dashboard or social feed, while a contextual agent (**CheshireGPT**) replies with deeper analysis, historical context, or code insights ([Web 3 Social Swarm | Default Page](#)). This echoes how one agent might announce a prediction and another enriches it with context and caveats. The swarm architecture ensures **decentralized intelligence** – no single agent has to do everything, they can swarm on a problem with each contributing from its domain of expertise.

Cheshire Terminal (User Interface): All agent interactions funnel through the **Cheshire Terminal**, a unified front-end for users. This could be a web dashboard that shows real-time insights, trending topics, and agent outputs in a user-friendly way. The Terminal might feature a “swarm trending board” that displays metrics from both social media and on-chain data side by side (e.g. trending token mentions vs. on-chain volume spikes) as the agents report them. Users can chat with the swarm (text or voice) via this interface, asking questions or requesting analyses; the query is dispatched to the appropriate agent or triggers a collaborative response from multiple agents.

Web3 Integration: The system deeply integrates with Web3 infrastructure to both **read and write** blockchain data:

- **On-Chain Data Monitoring:** Agents connect to blockchain nodes or APIs (e.g. via Infura/Alchemy for Base (Ethereum L2) and a Solana RPC for Solana). They subscribe to relevant events such as large token transfers, new contract deployments, or NFT mints. For example, one component might track **whale transactions** and feed that info into the knowledge base ([Web 3 Social Swarm | Default Page](#)). Significant events (like liquidity movements or governance proposals) become triggers for the swarm to discuss or alert on. This structured blockchain data is parsed and stored (both raw and as semantic embeddings) for later retrieval.
- **Transactions & On-Chain Actions:** Beyond reading data, agents can also **initiate on-chain transactions** when appropriate. For instance, an agent might

automatically execute a trade on Solana or an interaction on an Ethereum smart contract if it's part of its role (with safeguards). This requires secure key management – likely each agent has an associated Web3 wallet. We incorporate **Privy authentication** for user and agent key management. Privy is an authentication/key management platform that allows linking traditional accounts with Web3 wallets securely ([Privy – All your users, onchain.](#)). Using Privy, the Cheshire Terminal can onboard users via email or OAuth and transparently create or link a non-custodial wallet for them. This means users log in easily while still getting an on-chain identity for interacting with the agents or receiving rewards.

- **User Authentication & Identity:** With **Privy** (and possibly augmented by **Dynamic.xyz** for wallet orchestration), users of the platform are verified and can grant the system permission to act on their behalf on-chain (if needed). For example, a user might ask an agent to execute a swap for them; the agent, through Privy, can prompt the user to sign the transaction securely. All sensitive user data and keys remain end-to-end encrypted and secure via Privy's vault, aligning with the need for privacy and safety in Web3 apps ([Privy – All your users, onchain.](#)).

Social Media Integration: The swarm's social agents connect to platforms like Twitter (X) via API to both consume and post content. One agent might live-stream tweet analysis: using the Twitter API's filtered stream or Account Activity API to catch specific keywords or influencer tweets in real time ([Web 3 Social Swarm | Default Page](#)). Another agent can perform **sentiment analysis** on these tweets (using an NLP model) to gauge market mood instantly ([Web 3 Social Swarm | Default Page](#)). The agents can then post aggregated insights on the Terminal or even tweet out through a designated account (like how [@aixbt_agent](#) and [@cheshiregpt](#) coordinate in the social swarm example ([Web 3 Social Swarm | Default Page](#))). All posts made by agents are also stored in the knowledge base so they can refer back to what was said (enabling accountability and memory of promises or predictions).

Coordination: These social and on-chain agents communicate through the shared database or an internal pub-sub event bus. For example, when the on-chain agent detects a spike in a token's volume, it can emit an event that the social agent picks up to see if the token is trending on Twitter as well. The result might be a

combined insight like “Token XYZ is spiking in trading volume on Solana and trending on Twitter – possible announcement incoming.” The **intelligent retrieval** ensures each agent can query the full context (recent tweets, historical events, code from GitHub, etc.) to enrich its output ([Web 3 Social Swarm | Default Page](#)). This way, even though agents are specialized, they are not siloed – the **swarm intelligence** emerges from their collaboration and shared memory.

3. NVIDIA NIM Agent for High-Speed AI Compute

Edge Compute Node: We deploy a specialized agent on the user’s **NVIDIA Jetson Orin Nano** device to leverage local GPU compute for AI tasks. This **NVIDIA NIM (Neural Inference Module) agent** is optimized to run heavy models at high speed, offloading work from the cloud and reducing latency. The Orin Nano, equipped with an NVIDIA GPU and CUDA cores, is well-suited for running optimized neural networks (e.g. running a medium-sized LLM or image generator locally). By utilizing the Orin for on-device inference, the system can perform certain tasks **offline or in real-time** without always calling out to cloud APIs. For instance, the NIM agent could run a local instance of a transformer model for quick replies or execute stable diffusion to generate game graphics or NFT art on demand.

BrewLaunchable Deployment: To integrate the Orin Nano into the overall system seamlessly, we package the NIM agent as a **containerized service** that can be launched via a one-command deployment (the notion of “BrewLaunchable” implies an easy launch, akin to Homebrew formulas or Docker images).

Essentially, the user’s Orin Nano will run a lightweight server (possibly a FastAPI or gRPC server in Python/C++) that listens for tasks from the main system. When the swarm needs a high-compute task done, it dispatches it to the NIM agent. For example, if the user requests an AI-generated image or a lengthy piece of text analysis, the request is queued to the Orin service, which then processes it using its GPU and returns the result. Containerization ensures that all necessary libraries (CUDA, TensorRT, model weights, etc.) are pre-installed, making the agent **immediately launchable** on the Orin device without complex setup.

Optimizations for Speed: We optimize models for the Orin Nano using NVIDIA’s tooling. Heavy models are converted with **TensorRT**, which accelerates inference by optimizing the neural network graph for the GPU ([Maximizing Deep Learning Performance on NVIDIA Jetson Orin with ...](#)). We may quantize models (e.g. 8-bit

or 4-bit quantization) to fit within the Orin's memory while still maintaining acceptable accuracy. The NIM agent is essentially a mini AI server: for instance, it could host a local LLM (like a fine-tuned LLaMA-2) to handle user queries locally, avoiding external API calls. This is especially useful for privacy (user data can be processed on their own device) and for reducing latency in interactive applications (like a game, where quick AI response is needed).

Integration with Swarm: The Orin-based agent registers its capabilities to the swarm network. Other agents know that "if task == image_generation or large_LLM, send to NIM agent". The main orchestrator can use an async job queue to delegate to Orin and wait for results. We ensure robust fallback: if the Orin is offline, tasks can default to cloud services (e.g. use a cloud GPU or API). Conversely, when online, the local agent can take over certain duties completely – for example, it might continuously run a computer vision model analyzing game frames or user facial expressions (if the use case involves AR/VR), providing another stream of multi-modal input to the system. By **leveraging edge computing**, the system attains high throughput for AI tasks and reduces cloud dependency, which is cost-effective and empowers the user's hardware (especially relevant since the user specifically has an Orin Nano available for this purpose).

4. Virtuals Plugin for On-Chain Actions & Decentralized Execution

Virtuals GAME Framework: We integrate the **Virtuals Protocol's GAME SDK** to handle on-chain actions and to structure the agents' decision-making in a modular way. Virtuals GAME (General Autonomous Modular Environment) provides a framework where an agent can be given a goal, a set of available actions (functions), and it will plan and decide which action to execute based on current context ([GAME Framework | Virtuals Protocol Whitepaper](#)) ([GAME Framework | Virtuals Protocol Whitepaper](#)). This fits perfectly for enabling **on-chain operations**: we can define custom plugin "actions" that correspond to Web3 behaviors (e.g. *transfer tokens, execute swap on DEX, mint NFT, post to blockchain feed*). The agent's cognitive core (LLM-driven reasoning) will choose among these actions when appropriate, effectively allowing the AI to **act in the blockchain world** autonomously.

Python & TypeScript SDK: The Virtuals GAME SDK is available in both Python and TypeScript ([GAME SDK](#) | [GAME OS](#)). We can leverage the Python SDK on the backend to integrate the agent logic with our existing Python-based RAG system. For instance, using the GAME Python SDK, we configure an agent with certain **abilities** (plugins) and a persona. One plugin might be “SolanaTransaction” which we implement to interface with a Solana RPC (using `solana.py` under the hood), another could be “BaseContractCall” using `web3.py` or ethers. The Virtuals framework allows these actions to be added seamlessly as first-class options the agent can consider ([GAME Framework](#) | [Virtuals Protocol Whitepaper](#)). Meanwhile, if the front-end or a Node service needs to trigger agent behavior (say, to run in a browser or a Discord bot), the TypeScript SDK can be used to invoke the same agent logic or communicate with it. This cross-language availability ensures **seamless interaction between on-chain components and the AI logic** in whichever environment.

On-Chain Plugins: Using Virtuals’ plugin system, we incorporate **out-of-the-box on-chain functions** and also develop custom ones ([GAME Framework](#) | [Virtuals Protocol Whitepaper](#)). For example:

- An **on-chain trading plugin** that allows the agent to execute trades or liquidity provision on Uniswap (Base) or Orca (Solana) when certain market conditions are met. The agent would decide to call this function if its analysis (from the knowledge base and price feed) indicates an arbitrage or a rebalancing opportunity.
- A **wallet interaction plugin** that lets the agent manage a crypto wallet (for an NPC or the user). It could send tokens, check balances, or interact with NFT contracts. Through Privy’s SDK, the plugin can request the user’s signature when needed, maintaining security.
- A **cross-chain bridge plugin** if the agent needs to move assets between Solana and an EVM chain as part of its logic (this could use a bridging API or protocol).
- The plugin architecture is modular, so future capabilities (like image generation or external API calls) can be added as needed, and even community-contributed, per Virtuals design ([GAME Framework](#) | [Virtuals Protocol Whitepaper](#)).

Decentralized AI Execution: By integrating Virtuals, we also open the door to **decentralized execution** of the AI agents. The Virtuals protocol envisions that agents can be deployed in a way that multiple nodes or community members can contribute to their functioning (for example, an agent's decision process could be verified or even run by a network of token holders). In our system, while the core might run on our servers and user's devices, critical actions could be verified on-chain or by a decentralized oracle network to ensure trust. For instance, if the agent decides to execute a high-value transaction, we might require a smart contract on Base to verify a certain condition (like agent's token stake or a multisig approval) before execution – essentially a **governance or verification step** that aligns with decentralized principles. Virtuals also allows launching an "Agent Token" which could be used to distribute ownership or rewards related to an agent's performance ([AI Agents with Crypto Superpowers | by Henri.M - district0x Updates](#)), adding another layer of decentralization (though this is an optional feature of the ecosystem).

Connecting to Cheshire Ecosystem: The Virtuals plugin is how our AI swarm connects to the **Cheshire (Based) ecosystem** on-chain. The Cheshire smart contracts or identity system might require certain proofs or calls which the agent can carry out via these plugins. For example, if Cheshire Terminal has a contract for logging "verified AI insights" on-chain (as NFTs or events), the agent can trigger that through a plugin, imprinting important analyses immutably on the blockchain for transparency. This way, the **knowledge retrieval and generation is not a black-box** – key outcomes get recorded on-chain, anchored to the Cheshire ecosystem's contracts.

5. AI Decision-Making in Gaming Environments (Game SDK Integration)

Game Environment Integration: The system is designed to inject AI-driven decision-making into gaming environments using the provided Game SDK (which in our case is the Virtuals GAME SDK, but it could also extend to other game engines). This means our AI agents can function as NPCs or game masters within a video game or metaverse, making real-time decisions that affect gameplay. Using the Game SDK, we can bind the agent's **perception** to the game state and its **actions** to game APIs:

- The agent's **state input** can include the game world status (positions of players, events happening in-game, inventory, etc.) similar to how it would take social or blockchain context. We translate game state into a context that the agent understands (this might involve prompting the LLM with a description of the current scene or numeric state fed as part of the agent's observation).
- The agent's **available actions** are defined by the game's SDK. For example, in a MMORPG, actions might be "move", "speak dialog", "attack", "trade item". In a strategy game, it could be higher-level decisions like "allocate resources to build X". We register these functions in the agent via the Game SDK so the decision engine knows it can choose them.

Real-time Decision Loop: Within the game loop, the AI agent functions as an **autonomous character**. Each tick or at key events, the game calls the agent (or the agent polls the game state) to decide what to do next. The Virtuals GAME framework's decision engine (backed by an LLM) will take into account the agent's goal/personality and the current context to output an action ([GAME Framework | Virtuals Protocol Whitepaper](#)). For example, if we have a *DeFi trading card game*, an AI opponent agent could analyze both the on-chain data (maybe card NFTs on chain) and the player's moves to decide its next move, all while role-playing a certain persona.

Use of Knowledge Base in Games: Because our knowledge retrieval is robust, the agent in the game can tap into external knowledge to enrich gameplay. Imagine a puzzle game where the AI NPC gives hints: the NPC could actually pull from a database of riddles or use the LLM to generate hints based on a vast trove of info. Or in a **crypto treasure hunt** game, the AI could verify a clue on-chain (using its blockchain access) and then guide the player. The **multi-modal capability** also shines here – the agent can speak to the player with a unique voice (via Hume's voice models), show images or maps when needed, or even present on-chain evidence for a clue. This creates a highly immersive experience where the game's AI is not scripted with fixed dialogs, but is **dynamic and data-driven**.

Example – Virtual Guide NPC: Consider an in-game guide character in a metaverse that helps new users. This NPC agent can access the player's on-chain profile (what NFTs they hold, etc.), social media trends (to see what the community is up to), and then tailor its guidance. It might say via voice, "*I see you*

own an NFT sword that was forged on Solana last week – it's currently trending in the arena battles. You should try it out in the Coliseum event happening now!".

The agent arrives at this by retrieving on-chain data (NFT metadata), checking social feed (trending arena battles), and game state (player's inventory), all through our RAG system, then deciding on an action (dialogue) through the Game SDK. This showcases the **power of AI-driven decision-making in games**: the game world responds to real-world data and the player's context in a believable, intelligent manner.

Provided Game SDK: Since the question mentions a "provided Game SDK," it implies we have a specific integration point. In our solution, that is the Virtuals GAME SDK (open-sourced for Python/TS). If the gaming platform has its own SDK (for example, a Unity plugin or Unreal mod), we would create a bridge between that and our AI system. Typically, a thin integration layer could expose game events to a Python service (using websockets or RPC) and in return execute AI-suggested actions in the game engine (Unity could call a REST API that our agent service provides for decision-making, for instance). The implementation plan would cover adapting our agent outputs to the game's API. The key point is that our architecture is flexible: **the same AI brain that operates in Web3 and social domains can also live inside a game world**, thanks to modular design and the Game SDK abstraction.

6. "Based" ZK-Proof Verification and Cheshire Branding

ZK-Proof Verification: Security and trust are paramount in the Cheshire ecosystem. We incorporate **zero-knowledge proof (ZKP) techniques** to verify critical aspects of agent interactions without exposing sensitive information. In practice, this could mean using zk-SNARKs or similar to prove statements like "Agent X's recommendation is based on authentic data" or "User completed quest Y legitimately" without revealing the underlying private data. For example, if a reward is given when a user achieves some on-chain milestone, the system can verify that milestone with a ZKP rather than requiring the user to divulge all their wallet details. *Zero-knowledge proofs allow one party to prove to another that an on-chain statement is true without revealing anything beyond the truth of the statement ([Zero-Knowledge Proofs: The Magic Key to Identity Privacy - Galaxy](#)).*

In our architecture, an agent generating a crucial piece of analysis could attach a ZK-proof that it followed the approved algorithm (preventing any rogue behavior from a compromised agent). Similarly, when the user authenticates via Privy and completes an on-chain action, a ZK proof could attest to their eligibility for a reward in a privacy-preserving way. These proofs reinforce trust in the AI outputs and the system's integrity, aligning with the ethos of decentralization and privacy.


"Based" Cheshire Branding: All AI agents will maintain the **Cheshire ecosystem's branding and identity**, notably the "'based' zk-proof verified" style that has been cultivated. In practical terms, this involves both **visual branding** and **behavioral consistency**:

- Agents present themselves with the Cheshire persona (e.g. using the Cheshire cat or grin logo in the Terminal UI, and perhaps a witty, community-favorite tone in communications). Any user-facing messages can include a small tagline or watermark indicating it's "Based & Verified by Cheshire" to assure users the content is authentic and approved.
- Before an agent publishes any significant insight or executes an on-chain transaction on behalf of the platform, the action is stamped with a form of verification. This could be a digital signature using the Cheshire DAO's private key or a cryptographic proof logged on-chain. Essentially, the ecosystem's **zero-knowledge verification badge** is attached, signaling that the action/result has been verified according to Cheshire's standards (either via formal proofs or multi-sig validation).
- The **"based" terminology** likely alludes to the platform's use of Coinbase's Base chain as well as meaning "grounded". We ensure that wherever relevant, interactions with the Base L2 are prominent and efficient (since Base is part of the branding). For instance, the Cheshire Terminal might primarily operate on Base for any L2 transactions, taking advantage of lower fees, and using its finality to store important records. The zk-proof approach complements this by potentially using L2-friendly ZK-rollup tech for verification of off-chain computations.

Trust and Community: By baking in ZK verification and maintaining a strong brand persona, we foster user trust and community engagement. Users can trust that an insight labeled as "Cheshire verified" has indeed been cross-checked (maybe the agent's reasoning was verified by another agent or by a cryptographic

routine). The branding also helps in marketing – the **Cheshire AI** comes across as a reliably “based” (grounded in facts) assistant in the Web3 and gaming space. Technically, this means we might integrate libraries like zk-SNARK circuits (e.g. using SnarkJS or Halo2 library in Rust, invoked via Python bindings) for generating proofs, and smart contracts on Base to verify them. Those contracts could manage a registry of verified agents/actions – only content with a valid proof is considered “official”. This creates a **feedback loop of trust**: the more the agents act and prove their fidelity, the more users trust and engage with the system.

Implementation Plan

1. **Data Ingestion & Storage:** Set up pipelines to ingest data into Astra DB. This includes blockchain data (using providers like Alchemy for Base, and Solana RPC or The Graph for Solana) and social data (Twitter API streams). Define Cassandra tables for structured data (e.g. transactions with fields like hash, from, to, amount, timestamp) and for unstructured data (e.g. a table for documents or tweets with text and metadata). Enable vector search on relevant tables (Astra allows adding a vector column and indexing it for similarity search ([Astra DB Vector Store](#) |  LangChain)). Ingest historical data and periodically update with new events.
2. **Embedding & Indexing:** Implement a service to generate embeddings for incoming data. For text, use a pre-trained model (like SentenceTransformer or OpenAI Embeddings) to vectorize tweets, descriptions, and even code snippets. For any images (if storing NFT images or others), use a vision model (like CLIP) to get image embeddings. Store these vectors in Astra alongside the identifiers. This service runs continuously so that the knowledge base stays up-to-date for retrieval.
3. **LLM Integration (X AI or Equivalent):** Choose and integrate a core large language model that will power the agents’ reasoning and generation. If X AI (Elon Musk’s AI) provides an API or model (e.g. the rumored “Grok” model), integrate that; otherwise use an open model (GPT-4 via OpenAI API, or Llama-2 locally on Orin). Wrap the LLM with LangChain to facilitate passing retrieved context. Test the RAG loop: given a sample query, ensure the system properly retrieves from Astra DB and the LLM incorporates it into the answer. Use

LangSmith to trace these calls and verify that relevant data is being used (improving the prompt or retrieval parameters as needed).

4. **Backend Services & Orchestration:** Develop the Python backend as a collection of services:

- An API service (FastAPI or Flask) that serves the Cheshire Terminal frontend and provides endpoints for queries, agent status, etc.
- Agent daemons: processes or async tasks for each agent (social agent, on-chain agent, etc.). For example, a Twitter listener agent that pushes new tweets into the DB and alerts the main system, and an analysis agent that periodically scans for notable events and generates summaries.
- The **Orin NIM agent service**: running on the Jetson device, but accessible over the network. You can use MQTT or gRPC for lightweight messaging between the cloud and the Orin. For instance, when the main server needs an image generated, it sends a message to Orin's queue; the Orin does it and returns a URL or binary.
- A scheduler (could be Celery beat or APScheduler) to schedule regular tasks like "daily market summary" generation or routine cleanup.

5. **Web3 Connectivity:** Integrate Web3 libraries:

- For Base (EVM chain): set up `web3.py` or use ethers.js via a small Node microservice. Write functions to fetch on-chain info (balances, events) and to send transactions (e.g. using a provided private key or via user signature with MetaMask integration on the front-end).
- For Solana: use the `solana-py` SDK or RPC calls. Implement listeners for specific account addresses or parse transaction feeds (Solana has webhooks services or just poll the RPC for certain signatures).
- Test reading from and writing to both chains in a dev environment. For writing (transactions), integrate Privy: use Privy's SDK to get the user's wallet keys or to trigger signature requests. Ensure that for any automated agent wallet, keys are stored securely (in an HSM or Vault, or using a multi-sig scheme if high risk).

6. **User Authentication & Portal:** Implement the authentication flow with Privy. On the Cheshire Terminal front-end (likely a React/Next.js app), install Privy's

SDK to handle login. When users log in, retrieve their decentralized identity and any profile info. Use this identity to customize their view (e.g. show their linked wallets, let them select which wallet an agent should use for actions). The back-end verifies JWTs or tokens from Privy to establish sessions. Also integrate additional OAuth if needed (for Discord or Twitter linking, if users want to authorize the agents to post on their behalf, etc.).

7. **Front-End Development:** Build the Cheshire Terminal UI in TypeScript (Next.js or similar). This UI will have components like:

- **Dashboard:** showing real-time data (market charts, trending topics from the swarm). Use webSockets or Server-Sent Events from the backend to push updates (for low-latency updates when agents find new info).
- **Chat/Console:** where the user can converse with the AI agents (text input or microphone input for voice). This interacts with the backend API to get responses (possibly streaming responses for immediacy).
- **Control Panel:** allowing the user to deploy or configure swarm agents (maybe an interface to turn on/off certain agents, or to feed in custom data).
- Make it responsive and intuitive, using modern UI frameworks. Emphasize the Cheshire branding (colors, logos) and show when content is verified (perhaps a shield icon indicating zk-proof verified).

8. **Virtuals GAME Integration:** Now implement the higher-level agent logic using the GAME SDK:

- Write an agent config (could be in YAML or code as per Virtuals SDK) for each agent. Define their goals and descriptions (e.g. SocialAgent: “monitor crypto Twitter and provide timely alpha”), and list the actions they can take. Many actions will map to functions we have already implemented (like `post_tweet(content)`, `execute_transaction(tx)`, `speak_to_user(message)`).
- If using the Virtuals Python SDK, this might involve creating a subclass or registering functions. If the logic proves easier in TypeScript (for example, if the game environment is easier to hook in TS), consider using the TS SDK for those parts or even running a Node service for the agent reasoning.

- Test the agent decision-making in isolation: simulate a scenario (e.g., a spike in price) and see if the agent chooses the correct actions (perhaps posting an alert and doing a trade). Virtuals SDK likely has a way to run the agent in a loop where it assesses state and picks an action; use that to verify the flow.
- Integrate the GAME agent loop with our event system: when new data comes in or a user query arrives, feed it into the GAME agent as updated state so it can decide if an action is needed.

9. Gaming Use-Case Integration: If there is a target game environment:

- Connect the game's SDK or API with our system. For example, if the game is in Unity, use Unity's Web Request to call our backend's agent decision endpoint every few seconds or on certain triggers (like player enters area → ask AI for a response). Or if the game is web-based, use the Virtuals TS SDK directly in the game client to embed the agent.
- Create a **mock game environment** for testing, if needed (some simplified environment data) to ensure the AI behaves as expected. Tune the agent's prompt/persona to fit the game theme (for instance, make the agent more role-playful in a fantasy game, or more formal in a sci-fi game).
- Ensure performance is adequate – in a game, we may need to limit LLM response time or use a smaller model to meet real-time constraints. Possibly leverage the Orin agent for any local computation in an AR/VR device scenario.

10. ZK-Proof and Security: Implement the zk-proof mechanisms for critical flows:

- If using a known library, set up the circuits and proving environment. For example, use snarkJS for a simple demo: prove that a number is prime without revealing the number (just conceptually to test). Then move to actual use cases, e.g., a circuit that takes a user's wallet and an achievement and proves the user has that achievement NFT without revealing which user (for anonymous leaderboards).
- Alternatively, for a simpler approach initially, use digital signatures: each agent could have an Ethereum key and sign its messages. The public key

tied to Cheshire is known, so users can verify the signature off-chain. This can later be complemented with true ZK proofs.

- Harden the security of key management: integrate an HSM or at least ensure Privy's custody of keys is used so the backend never exposes private keys in plaintext.
- Run penetration tests or audits, especially on the on-chain transaction flow and the zk-proof verification contracts.


11. Testing & Iteration: Test each component thoroughly:

- Unit test the data retrieval (does the query to Astra bring the right context?).
- Simulate user queries to the RAG pipeline and verify correctness of answers (perhaps comparing against known truths).
- Test multi-modal: speak a query via microphone and ensure the pipeline from speech-to-text to answer to speech output works end-to-end.
- For the swarm agents, simulate various scenarios (bull market tweets vs. bear market, or different game scenarios) and observe if agents coordinate properly. Adjust their prompts or the shared memory logic if needed to avoid redundant or conflicting actions.
- Use LangSmith's evaluation capabilities to gather feedback on LLM outputs (measure relevance/accuracy of answers ([Evaluate a RAG application](#) | [LangSmith](#))).
- Involve beta users or team members to try the Cheshire Terminal and report issues or confusion. Use their feedback to refine the UI/UX and agent behaviors.

12. Deployment: Deploy the backend on a cloud platform (could be AWS EC2 or a container on Kubernetes, etc. – Astra DB is serverless so just ensure connectivity and proper Astra credentials in env). Deploy the front-end (perhaps on Vercel if Next.js). Deploy the Orin agent on the physical device (it might auto-start on device boot and connect back to the cloud). Set up monitoring for all agents (logs, errors) and use a tool like LangSmith or custom dashboards to monitor agent decisions and outcomes in production.

13. **Maintenance and Future Enhancements:** Once live, maintain the system by updating models (e.g. incorporate X AI's latest model if available), adding new data sources (maybe other social platforms or more chains like Arbitrum, etc.), and improving the ZK verification (aim to make more and more of the process trustless with time – possibly even move some agent logic on-chain in simplified form). Also, consider community features: since it's a swarm, perhaps allow power users to run their own agent nodes that contribute data or analysis (similar to how folks might run their own instance of the agent connected via an API, crowd-sourcing intelligence). The architecture is designed to be modular and extensible, so new plugins (say a **DAO governance plugin** for the agent to propose/vote in DAOs, or a **DeFi lending plugin** to integrate Aave) can be added with minimal changes to the core.

Recommended Frameworks and Tools

- **Database and Vector Store:** **DataStax Astra DB** for a scalable serverless DB with Cassandra reliability and built-in vector search ([Astra DB Vector Store](#) |  [LangChain](#)). This avoids managing separate databases for structured vs. semantic data. Alternatives: Weaviate or Pinecone for pure vector search (if not using Astra's), and PostgreSQL for structured data if needed.
- **LLM and RAG Orchestration:** **LangChain** (Python) for chaining the retrieval and generation steps, and managing prompts. **LangSmith** for observability (tracking queries, responses, and evaluating success metrics in the RAG pipeline). The LLM can be OpenAI's GPT-4 (for best quality) or an open model via Hugging Face Transformers (like Llama-2 13B for local running). If Elon Musk's **xAI** releases a model or API, that could be integrated here as an LLM choice. We will also use smaller models for support tasks: e.g. sentiment analysis (perhaps a HuggingFace model fine-tuned for finance sentiment) and embedding models (e.g. **all-MiniLM** for quick embeddings if OpenAI embedding API is too slow or costly).
- **Multi-modal AI:** **Hume AI** platform for voice - it provides APIs/SDK for TTS and emotion recognition which we'll use for the empathic voice feature ([Home • Hume AI](#)). For speech-to-text, OpenAI Whisper or Google Cloud Speech can be used. For image generation, **Stable Diffusion** (with a custom LoRA model if we train on specific art style, as hinted by the Cheshire art engine) running on

the Orin Nano with optimizations. For image analysis (like reading blockchain QR codes or analyzing charts), OpenCV combined with a lightweight CNN could be used.

- **Web3 and Blockchain:** **Privy SDK** for authentication and wallet management in the dApp ([Privy – All your users, onchain.](#)). **Web3.py** for Ethereum/Base interactions (or ethers.js if using JS on that part). **Solana Python SDK (solana.py)** for Solana, or **Anchor** framework if needed to interact with Solana programs. We might also incorporate **Covalent** or **The Graph** for easier blockchain data querying (Covalent API can fetch historical transactions conveniently, TheGraph can get structured data from specific protocols). For on-chain monitoring, services like **Alchemy Notify** or **QuickNode webhooks** can simplify catching events without constant polling.
- **Agent Framework: Virtuals GAME SDK** (Python & TypeScript) for defining agent capabilities and autonomous decision logic ([GAME SDK](#) | [GAME OS](#)). This gives a structured approach to building agents that can plan actions (critical for the multi-action swarm and game NPC behaviors). We will also use Virtuals' **plugin library** if available for existing integrations (they might already have a plugin for "tweeting" or "sending a transaction" which we can use or adapt ([GAME Framework](#) | [Virtuals Protocol Whitepaper](#))). If not using Virtuals for some reason, an alternative could be **LangChain's agents** (which allow an LLM to choose tools/actions in a similar manner) or **Haystack** for a custom agent loop, but Virtuals is tailored to Web3 use cases.
- **Backend & Infra:** **FastAPI** for the Python API server (fast and easy to define REST and WebSocket endpoints). **Docker** for containerization of services (especially the Orin agent – likely using NVIDIA's base images for CUDA support). **Kubernetes** or Docker Compose to manage multi-container deployments (we might have a container for the main API, one for each agent type if separating concerns, etc.). Use **Redis** or **NATS** as a message broker for internal pub-sub (for the swarm agents to talk to each other or to queue tasks for Orin). **Celery** could be used with Redis if we need a task queue (though for real-time, event-driven might suffice).
- **Frontend: Next.js (React)** for the web interface, as it supports TypeScript and is great for building interactive dashboards. We'll employ UI libraries like **Chakra UI** or **Tailwind CSS** for a clean, reactive UI. For charts/visualizations

(displaying data trends), use **Plotly.js** or **D3.js** (since image embedding is off, we assume charts would be rendered on client side). The front-end will also utilize **Privy's React components** if available to streamline login, and perhaps **Web3Modal** or similar for wallet connections if users want to connect external wallets.

- **Communication: WebSockets (Socket.IO)** for real-time updates from backend to front (e.g., push new swarm findings to the dashboard immediately). **Discord bot framework** (like discord.py) if extending the social swarm to Discord communities as well. Possibly **Twilio** integration if we allow SMS or phone interactions (the Hume AI voice could even be used over a phone call to give verbal updates, which could be a neat feature for alerts).
- **NVIDIA Optimization: TensorRT and PyTorch with CUDA** on the Orin Nano for running models efficiently. NVIDIA's **JetPack SDK** will be used on the Orin to ensure we have all necessary drivers and libraries. We may also use **NVIDIA Triton Inference Server** on the Orin if we want to serve multiple models (it can optimize serving of multiple ML models behind gRPC/HTTP).
- **Security & ZK: Circom/SnarkJS** for generating zero-knowledge proofs in the browser or Node (if simple circuits for demo), and **Halo2 or ZoKrates** for more complex circuits. On the blockchain side, **Polygon ID or Mina** could be looked at for managed ZK identity solutions. To manage cryptographic keys and signatures, use **ethereumjs** libraries for signing data (for the agent signatures) and standard **ed25519** libs for Solana.

All these components together create a robust full-stack system. The combination of a Cassandra-based vector store, a swarm of specialized agents, high-performance local inference, and deeply integrated Web3 hooks (with modern authentication and ZK security) will fulfill the requirements. Each framework was chosen for being well-suited and proven: for example, Astra DB's vector search and LangChain make implementing RAG easier ([Evaluate a RAG application | LangSmith](#)), and Virtuals/Game SDK is explicitly designed for agent actions in Web3/gaming contexts. The end result is a **cutting-edge RAG-powered swarm intelligence platform** that bridges AI and blockchain, capable of driving rich user experiences from DeFi dashboards to immersive games – all under the **Cheshire "Based & Verified" banner**.