ChatGPT

Feasibility and Execution Plan for Sentience: A GPT-Powered EVE Online Co-Pilot

Project Feasibility and Technical Architecture

Technical Scope: Building *Sentience* is feasible with moderate lift. It involves three main components: **(a)** connecting to EVE's official ESI API for game data, **(b)** leveraging a GPT language model for natural language responses, and **(c)** a chat-based interface (web or Discord) to interact with players in real time. ESI (EVE Swagger Interface) exposes rich JSON data about nearly every aspect of a character (assets, skills, wallet, market orders, industry jobs, etc.) via **119 authenticated endpoints** (player-specific) and public endpoints [1] [2]. The assistant's strength will be combining this live data with GPT's conversational abilities.

Key Tools & Frameworks: Essential tools include an **ESI API client** (e.g. Python libraries like **EsiPy or Preston** [3] to handle OAuth and requests) and the **OpenAI API** (e.g. GPT-4 via `openai` Python or Node SDK) for generating responses. A lightweight web framework or bot SDK will manage the chat interface (for instance, **Discord's API** via discord.py or a simple Flask web app for a browser chat). Fortunately, the heavy NLP lifting is offloaded to GPT – the developer focuses on data retrieval and prompt engineering. Many components can be **scaffolded using OpenAI's tools**: For example, OpenAI's platform allows defining *functions* or plugins that GPT can call in conversation to fetch external data [4]. We can define functions corresponding to ESI queries (like `get_wallet_balance`, `list_assets(character)` etc.) – GPT-4 can decide to call these when needed, which **greatly simplifies the logic**. In other words, rather than writing a custom intent parser, we let GPT's *function calling* feature interpret the user's request and trigger the right ESI data fetch.

Use of Custom GPT or Plugins: OpenAI's **Custom GPT** feature (launched 2024) and the plugin system could jump-start development. One approach is to create a **ChatGPT Plugin** with the ESI API's OpenAPI spec: this would give ChatGPT native access to ESI endpoints via the plugin. For instance, by uploading ESI's API schema, a custom GPT could call "get character assets" or "query market prices" directly [4] [5]. This **plugin approach** means you *don't* have to build a UI – you could chat with "Sentience" inside ChatGPT. However, distributing a plugin to end-users is non-trivial (users would need ChatGPT Plus with plugins, and you'd have to host the plugin service). It's still worth considering for rapid prototyping: OpenAI's plugin infrastructure handles OAuth (the plugin can use the standard EVE SSO flow) and provides a convenient chat UI. In summary, *Sentience* can start as a **custom GPT agent with an ESI plugin** for quick demos, then evolve into a standalone app.

Data and Privacy: The ESI integration uses **read-only OAuth2 scopes**, so the assistant can fetch data but not perform in-game actions (in line with EVE's third-party policies). You'll register a developer application with CCP to get a client ID/secret and specify scopes (e.g. wallet, assets, skills) [6]. Upon user login via EVE SSO, you obtain an *access token* (valid ~20 minutes) and a *refresh token* [7]. The technical lift here is managing this token exchange and securely storing the refresh token for each user so the assistant can retrieve data continuously. This is standard – many EVE tools use the same flow (e.g. EVEMon and others

now use ESI + SSO) and there are libraries and examples to follow [8] [9]. In short, **ESI integration is a solved problem** in the EVE dev community; you won't need to reinvent it, just glue it into the GPT pipeline.

Real-Time Performance: ESI queries are RESTful and usually fast for small data calls (though large asset lists might need pagination). GPT API calls introduce some latency (on the order of 1–3 seconds typical). For a co-pilot, this is acceptable – the assistant's responses arriving in a few seconds should be fine for chat. We should implement caching for any repetitive calls (e.g. cache market prices for a short time) to stay within ESI's error-limited request quotas [10]. Overall, the stack (ESI + GPT + chat interface) is cloud-friendly and can be quite lean – the heavy computation of language modeling runs on OpenAI's servers, and EVE's data API is maintained by CCP.

Market Scan: Existing Copilots and Gaps

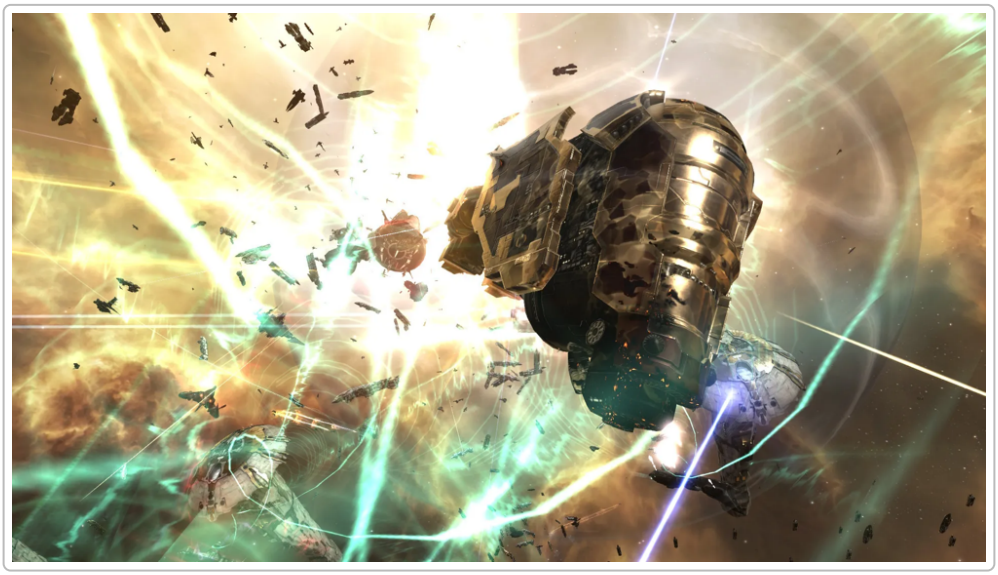

Figure: A large-scale battle in EVE Online. The game's complexity has spurred many third-party tools, but few offer an AI co-pilot that can digest personal game data and context in real-time [11].

Comparable Tools in EVE: There are a few existing assistants and tools, but none quite like *Sentience*. The closest might be **EVE OS** by Luke Kabbash – a web suite that even includes **an AI-powered chat assistant for EVE questions** [12]. EVE OS's AI likely answers questions about game mechanics or helps with intel, but it's part of a broader tool (maps, mail client, etc.) rather than a dedicated conversational copilot. Another is the **Torsten Feld EVE Alexa Skill**, which lets players ask Alexa about in-game info via ESI [13]. It's a voice Q&A format ("Alexa, how much ISK do I have?") and shows that integrating ESI data into natural-language responses is possible even on low-resources devices. However, Alexa's replies are scripted – not the dynamic, explanatory style a GPT can provide, and Amazon voice skills have privacy concerns and limited adoption among gamers [13].

On text platforms, **chatbots** exist but are mostly command-driven utilities. For instance, *Erik's EVE Tools* on Slack lets users type commands in Slack channels to get data like current Jita prices or structure info [14]. Similarly, Discord bots (e.g. **ThunderED**) provide corp management and intel via commands and

notifications [15], but **not** free-form questions or analysis. These tools lack the conversational AI layer – they require exact commands and don't do planning or recommendations.

A noteworthy experiment was **Neural Nexus**, an EVE corporation that made GPT-4 a sort of "AI CEO". They built a **ChatGPT Discord bot cognizant of EVE's mechanics** and consulted it for strategic decisions [16]. This shows the *interest* in AI copilots: even without API data access, an AI familiar with EVE's **lore and mechanics** can contribute creative strategy (Neural Nexus's bot helped propose measurable corp goals [17]). *Sentience* can go further by also incorporating *factual game state data* via ESI – something a plain GPT on Discord lacked. In short, current community uses of GPT in EVE are limited to either static knowledge Q&A or fun roleplay; no public tool yet **combines live personal data with GPT** for on-the-fly guidance.

Gaming AI Assistants Beyond EVE: More broadly, AI copilots for complex games are just emerging. For example, **Gaming Copilot** (2025) on Steam is a general "AI gaming buddy" that offers strategy tips, voice chat, and even screenshot analysis across many games [18] [19]. It uses large language models via Azure OpenAI [20] and game databases to provide hints. However, it's *game-agnostic*; it doesn't plug into game APIs for personal data. *Sentience's* advantage in EVE is the **official API access** – enabling personalized, context-aware help (your wallet, your skill training, your industry jobs), which generic assistants can't provide. Another example is **Razer's AI Gamer Copilot**, focused on real-time coaching and performance (e.g. analyzing your gameplay video feed for tips) [21]. This is cutting-edge but again not deeply integrated with game data or economy planning. Microsoft's announced *Copilot for Gaming* was more about guiding players through games with Bing knowledge and FAQs [22] – useful for static content like puzzles or finding game info, but not an EVE-specific expert.

Market Gap: The gap is clear – EVE players have many tools for **data** and some for **chat/voice**, but none that unite both into a truly intelligent co-pilot. EVE's complexity (often jokingly "spreadsheets in space") means players juggle information about assets, markets, skill plans, etc., often via external apps or spreadsheets. *Sentience* could fill that gap by answering questions like *"Given my current minerals and blueprints, what's the most profitable item I can manufacture?"* or *"How many more skill points until I can fly a Freighter, and what skills should I train next?"* – in plain English, referencing the player's actual data. Competing tools either **lack AI** (so they can show data but not advise or explain) or lack **live data** (so an AI can only speak generally). This unique combination is Sentience's value proposition.

Development Workflow and GitHub Usage

Version Control from Day 1: Yes – this project should absolutely be developed in a Git repository from the start. Using GitHub (or similar) will provide code history, issue tracking, and easy collaboration if it expands beyond a solo effort. Even as a solo developer, treating the project professionally with commits and README documentation is beneficial for organization. Many successful EVE third-party tools have been open-sourced on GitHub, from **EVEMon** (skill planner) to **Pyfa** (fitting tool) [23] [24], which helped them gain community trust. *Sentience* likewise interacts with sensitive user data (wallets, etc.), so being open-source early could reassure users that nothing malicious is happening with their tokens.

Open vs Closed Repo: Early-stage, you can start in a private repo if you prefer to clean up the code before public release. But there's little reason to keep it secret – the idea is not to monetize immediately, and an open repo might attract contributors or at least early adopters willing to test. Given the EVE community's collaborative nature, an open-source MVP might actually get you feedback faster. In the long run, if the project grows, a public GitHub also serves as marketing: other EVE devs might suggest improvements or

integrate Sentience into their tools. If you worry about "overthinking monetization," lean towards **public development** – focus on making it work and worry about profit later as stated.

Collaboration & Project Management: Even as a solo dev, you can use GitHub issues to track tasks and feature ideas. For example, create issues for "Implement skill queue query" or "Add chart for wallet balance history". This effectively creates your roadmap in an agile way. If using a public repo, you might get community feature requests as GitHub issues too. It's also wise to use a branching strategy (feature branches, etc.) to avoid breaking a working prototype. And if you use any AI coding assistants (like GitHub Copilot – aptly named!), that can speed up boilerplate coding, but always review the code especially around API authentication logic.

Infrastructure & Platforms for a Lightweight MVP

Hosting and Compute: Aim for a **serverless or low-cost cloud solution** to avoid upfront expenses. The MVP could run on a single small container or function. Consider these options:

- **Cloud Functions / Serverless:** If building a simple web API (for a web UI or ChatGPT plugin), you could use AWS Lambda, Google Cloud Functions, or Cloudflare Workers. For instance, Cloudflare Workers can run JS/TS code at the edge with free tier limits – ideal for handling chat HTTP requests. However, a persistent connection (as needed for a Discord bot) is not straightforward on pure serverless. A compromise is **AWS Lambda behind API Gateway** for request-response (e.g. for web chat or plugin) and perhaps a separate lightweight process for any persistent tasks (like listening to Discord events).
- **Container Hosting:** An easy path is to use a service like **Fly.io or Railway** which offer free tiers for running small apps 24/7. A Discord bot or a small Flask server can run on these with minimal config. For example, a single shared CPU VM with 256MB RAM is likely sufficient initially (most heavy work is calling external APIs). These platforms are essentially free VPS with auto-deployment from GitHub – very scrappy-friendly.
- **Traditional VPS:** A $5–10/month VPS (DigitalOcean, Linode) could host a Python/Node app and a database. This isn't free, but it's still low cost. If you foresee needing to persist user tokens and run background tasks reliably, a tiny VM might be simplest, but only opt for this if serverless options prove too limiting.

Auth and Data Storage: You'll need to store user credentials (especially ESI refresh tokens) and possibly some user preferences or chat history. A managed backend-as-a-service can speed this up:

- **Firebase** (Google's BaaS): Provides authentication, Firestore (NoSQL DB), and hosting in one. While Firebase doesn't have a built-in EVE Online auth, you can use its Custom OAuth setup or just use Firebase's Firestore to store tokens after you do the OAuth flow manually. Firebase shines in quick setup and has generous free tier, but its **NoSQL JSON data model** can be clunky for relational data. EVE data (like assets, skills) is relational by nature. Still, Firebase could handle storing a simple mapping of `user -> refresh_token` and maybe caching some JSON responses. Just be mindful of its pricing model: it counts reads/writes which can be unpredictable at scale [25] (though likely fine for a small user base).
- **Supabase**: An open-source alternative to Firebase, built on PostgreSQL. It offers **built-in user management and OAuth** and would allow storing data in a relational form (Postgres tables) which might feel more natural for EVE data (e.g. you could have a table of characters, linked to assets, etc.,

if you ever store more than just tokens). Supabase also supports *edge functions* (serverless functions) and real-time subscriptions. It has a free tier and the **predictable pricing** of a standard DB (pay for storage, not per access) [25] . A plus: Supabase being open-source means you can self-host if needed, avoiding vendor lock-in [26] . Given a solo dev MVP, Supabase could be overkill if you only need to store a handful of tokens, but it's worth considering if you plan to expand features (and it might handle EVE SSO via a custom OAuth provider configuration).

- **DIY minimal DB:** For early prototyping, you might not even need a full database. You could store refresh tokens in an encrypted JSON file or use a lightweight key-value store (like Redis or SQLite). Since the focus is not on heavy data persistence but live fetches, you might get away with keeping things in memory (noting that if the server restarts, users might need to re-auth). To avoid re-authentication hassle in testing, even a flat file of tokens is fine. Just plan to migrate to a proper secure storage before inviting more users.

Other External Services: You mentioned charts and Markdown reports. To generate **charts**, you have options like using a Python plotting library (matplotlib or Plotly) to produce an image, or a JS chart library on a web client. For a quick MVP, an approach could be: if the assistant is asked for a chart (e.g. "show my wallet balance over time"), have the back-end fetch the data, use `matplotlib` to create a PNG, and send it as an image file (if on Discord, upload image; on web, send URL or base64 data). This is achievable even in a serverless function (may need a headless backend or use an API like QuickChart). It's a "nice-to-have" feature, so implement a simple textual summary first, and add charts once basic QA is stable.

Architecture Summary: The MVP architecture might look like:

- A small **backend service** (Python Flask/FastAPI or Node.js) providing endpoints for login (ESI OAuth redirect) and for processing chat messages.
- That service calls **ESI REST API** using HTTPS (ESI is open internet API) with the user's token to get JSON data [27] [28] , then calls **OpenAI's API** with a composed prompt (including the user's question and data).
- The response is returned to the **frontend**: which could be a web UI (simple chat window) or directly posted to Discord via a bot account.
- **Database/storage** holds user tokens and possibly some cached data to avoid hitting ESI for every repeated query.
- Use **GitHub Actions** or similar CI to deploy this continuously as you update (to the chosen platform).

This stack is *lightweight*. All components have free tiers for initial usage: OpenAI API has a free trial and then pay-as-you-go (you can cap usage to a few dollars in testing), ESI is free to use (just abide by error limits and include a User-Agent per CCP guidelines [29]), and the hosting/database can start on free plans as discussed.

Roadmap: First 1–3 Months (Solo Dev Plan)

Below is a **step-by-step, prioritized plan** for developing *Sentience* from scratch, focusing on rapid progress without over-engineering:

1. **Setup Project Foundation (Week 0-1):** Initialize a GitHub repository for Sentience. Write a clear README with the project's vision and checklist of features. Set up the basic tech stack locally (choose a language – *Python* with FastAPI + discord.py, for example, or *Node.js* with Express + Discord.js).

Initialize the OpenAI API client with a test key and ensure you can make a simple GPT query. *Outcome:* "Hello World" commit where the app can log "Sentience started" and perhaps respond to a test ping.

2. **Register EVE Application & OAuth Flow (Week 1):** Create an EVE developer application via the EVE dev portal [6]. Choose **scopes** for data you plan to use in MVP (e.g. `esi-wallet.read_character_wallet.v1`, `esi-skills.read_skills.v1`, `esi-assets.read_assets.v1`, `esi-industry.read_character_jobs.v1`, `esi-markets.read_character_orders.v1`). Implement the **SSO login flow** in the app:

3. Add a route (or Discord command) to start login: directs user to EVE Online SSO URL with your app's client ID.
4. Handle the callback URL: capture the auth `code`, trade it for tokens (using an HTTP POST to EVE's token endpoint), and save the refresh token [9] [7]. For now, store it in a simple in-memory structure or a file (since it's just you testing).
5. Use the access token to call a trivial endpoint (e.g. verify character identity via `/characters/{id}` ESI endpoint) to confirm it works.

6. *Outcome:* You can log in with your EVE character and the app prints your character name or ID as confirmation.

7. **Core Data Retrieval Functions (Week 2):** Write modular functions to fetch key data from ESI using the token:

8. `get_wallet_balance()`, `get_asset_summary()`, `get_skill_queue()`, `get_industry_jobs()`, etc., returning Python dicts/objects. Keep these scope-focused (e.g., start with just character wallet balance and transaction list, not corporation wallets).
9. Test each function standalone and handle pagination or large data gracefully (perhaps limit results initially, e.g. only fetch first 50 assets or aggregate by location to avoid overload).
10. Consider using an ESI client library (like **EsiPy**) to simplify endpoints, or direct `requests` calls with the proper Auth header.

11. *Outcome:* A small library of EVE data helpers that you can call in code to get up-to-date info. This forms the "knowledge base" for Sentience to pull from.

12. **Integrate GPT for Q&A (Week 3):** Connect the data layer to GPT:

13. Decide on a strategy for prompts. A simple approach: always prepend a system message like *"You are Sentience, an AI co-pilot for EVE Online. You have access to the player's data and the EVE universe knowledge. Answer succinctly and helpfully."* Then for each user query, compile relevant data. For example, if the user asks "How much ISK do I have and is it enough to buy a Battleship?", the app would call `get_wallet_balance()` and maybe fetch a rough price of a battleship from markets, then feed those facts into the prompt before GPT.
14. Implement a basic logic for a few query types: detect keywords like "wallet", "assets", "skills" in the user query (or use GPT itself to decide, via function calling). To keep it scrappy, you might start with manual triggers. E.g., if query contains "wallet" -> fetch wallet; if "skill" -> fetch skills.

15. Call OpenAI API with the composed prompt (user question + data inserted). **Important:** Format the data clearly in the prompt (as bullet points or a table) so GPT can reliably use it. For instance: *"User's wallet balance: 1,234,567 ISK. Current skill in training: Mining V (3 days remaining)."*
16. Get GPT's answer and return it to the user. Ensure you **preserve any Markdown** in the answer (since the user expects Markdown-formatted output).

17. *Outcome:* End of Week 3, you should be able to have a conversation with the bot like:

    - User: "What's my ISK balance?" -> Bot (after fetching): "You have 1.23 million ISK in your wallet [30]."
    - User: "Can I afford a Raven battleship?" -> Bot (fetch price, compare): "Ravens cost ~200 million ISK, so no, your 1.23 million ISK is far from enough. Consider smaller ship or earning more first."

18. **Chat Interface MVP (Week 4):** Choose the interface for the prototype:

19. **Discord Bot**: Easiest if you're comfortable – register a bot, invite it to a server. Use a library to pipe messages to your backend. Implement a simple command like `!login` that returns the EVE SSO link for the user, and listen for messages to respond. Discord provides user identity so you can map Discord user -> EVE character token (store this mapping when they complete OAuth). Many EVE players already use Discord, so this is a natural environment.
20. **Web UI**: Alternatively, spin up a minimal web page with a chat textbox (could use an off-the-shelf chat widget or just a form). On submit, an AJAX call goes to your backend which returns the GPT answer. Web is nice to show formatted Markdown and images in-line. You might use a framework like Streamlit for quick prototyping of a web app with minimal HTML/CSS.
21. Either way, **don't over-engineer the UI**. It should be just enough to send user queries and display answers with formatting. Focus is the backend logic.

22. *Outcome:* By end of month 1, you have a working end-to-end *Sentience* prototype: you can log in with your EVE character, ask a question in natural language, and get a relevant answer that combines live data and GPT-generated text.

23. **Testing & Refinement (Weeks 5-6):** Use the assistant as you play (or simulate playing) EVE. Note where it fails or gives wrong answers:

24. Test a variety of questions: "Where are my assets located?", "How much have I earned in the last 7 days?", "What's the best way to increase my mining yield?" etc. Ensure that for purely knowledge-based questions (like game mechanics) the GPT doesn't hallucinate – consider injecting EVE Uni Wiki facts if needed, or restrict it to only use provided data. You might add a **safety net** in the prompt like: *"If you don't know a factual answer, respond with 'I'm not certain'."*
25. Pay attention to formatting: Are Markdown tables or bullet lists displayed correctly in the chosen interface? Tweak the prompt style or post-processing to fix issues (e.g., Discord might not support some Markdown features that the web would).
26. If using Discord, test DM vs channel, and that only the querying user's data is used (privacy of data is crucial). Implement basic permission checks (the bot should not answer another user with someone else's data – likely by design since token is per user, but double-check contexts).

27. *Outcome:* A more robust version of the assistant. By now, core functionality (wallet, assets, skills queries) should be reliable. This is a good checkpoint to consider sharing it with a friend or two for feedback.

28. **Optional – Enhance GPT Integration (Week 7):** At this point, you might integrate the more advanced "function calling" feature of OpenAI properly. This means instead of your own keyword detection, you define a schema for functions like `get_wallet()` and let GPT decide when to call them. OpenAI will then return function call instructions that your code can execute, then you feed the results back. This can make the bot more flexible (it might call multiple functions in one conversation turn if needed). It does require carefully specifying function inputs/outputs. Evaluate if the added complexity is worth it for the MVP; it might be something to iterate on after initial user feedback. *Outcome:* (Optional) The assistant can handle more complex multi-step queries automatically, using GPT as an agent to orchestrate data retrieval.

29. **Feature Expansion (Weeks 8-10):** With basics solid, implement more of the "nice to have" features if they align with user requests:

30. **Market queries:** e.g., "What's the current price of Tritanium in Jita?" – This could use ESI's market endpoints or a third-party market API for convenience (like EVEMarketer's public API). Provide comparison of buy/sell if useful. This adds value for traders.
31. **Industry planning:** If the user asks about industry, fetch their **blueprints or current industry jobs** (ESI has endpoints for active industry jobs [31]). The assistant could calculate completion times or whether required materials are in assets. You could also integrate static data for blueprint recipes (from the Static Data Export or something like *Fuzzwork* API since ESI doesn't fully list materials by itself [32]). Even a basic answer like "You have a Ferox blueprint and the necessary minerals to build 5 of them" would be impressive.
32. **Skill recommendations:** Use the character's skills info to answer questions like "What can I train to fly X ship?" This might require static data (ship skill requirements). Perhaps limit scope: focus on recommending based on what they have (e.g., "You're 2 days away from Mining V which will improve your yield by 5% [33]."). GPT can handle general suggestions (it knows typical skill progressions), but ensure it sees the actual skill levels via data to avoid bad guesses.
33. **Output formats:** Allow the user to ask for JSON or a chart. For JSON, you can pretty directly dump the data object (maybe through a pretty-printer) – this is straightforward. For charts, implement a couple of example charts (wallet over time if you have transactions data, or skill progression). Use a library to generate and host the image; the assistant can then embed it in the reply. *Keep this flexible*: not every answer needs a chart, only generate on explicit ask to avoid slowing responses.

34. *Outcome:* By around the end of month 3, Sentience should handle a range of capsuleer queries and produce helpful responses – from straightforward factual ones to insightful advice. It should feel like a true co-pilot, not just an API data dump.

35. **Deployment & Beta (Weeks 11-12):** Deploy the application for external access. If Discord-based, this might just mean hosting the bot on a cloud VM 24/7 and inviting some users. If web-based, deploy the frontend (e.g. on Vercel or Netlify for static, or as a managed app on Heroku/Fly.io). Make sure **environment variables** (API keys, client secrets) are configured securely on the host. You may integrate a simple database now if you haven't (to persist user tokens beyond app restarts – a small SQLite or a Supabase instance can do). Do a **security review**: ensure tokens are encrypted at rest,

and that if any error occurs the app doesn't spill sensitive data to logs. At this stage, consider making the GitHub repo public if it isn't – and add a proper open-source license (MIT or BSD for simplicity, unless you have reasons to restrict usage).

36. Invite a handful of EVE friends or post on an EVE forum/Discord about your "Sentience AI co-pilot beta". Emphasize it's experimental and gather feedback.

37. *Outcome:* A live MVP accessible to others, and initial user feedback collected for the next iteration.

38. **Monitoring and Iteration (Ongoing):** Once others use it, you'll quickly learn what features are most wanted. Monitor for any issues:

    - Watch for **OpenAI usage costs** – with a few users it should be low, but set up billing alerts. If needed, you can restrict to GPT-3.5 for less critical answers to save cost (e.g., use GPT-4 only when the question is complex).
    - Monitor **ESI rate limits** – if you have many users, implement request queuing or more caching (ESI is error-limited, so handle errors gracefully, perhaps by pausing calls for a bit if limit reached [10]).
    - Keep an eye on **edge cases**: unexpected user inputs, or GPT giving EVE-incompliant advice. Maintain a log (with user consent) of questions and answers to review how Sentience is performing and refine prompts or data retrieval logic accordingly.
    - *Outcome:* A stable, improving co-pilot ready to either grow in userbase or be enhanced with new capabilities (perhaps one day even integrating write actions like setting skill queue – though that's beyond read-only scope and would raise ethical questions, so likely avoid for now per EVE's third-party rules [34]).

Alternate Considerations: Along this roadmap, you'll make choices like *Firebase vs Supabase for auth*. As discussed, both can handle an MVP – **Firebase** offers simplicity and quick start, whereas **Supabase's relational approach and open-source nature** may better suit as you scale (especially if you want to self-host to cut costs long-term) [35] [25]. For a scrappy MVP, you might even avoid both and just use a file or memory, but plan to migrate once you have even dozens of users. Another alternative is using an existing **EVE OAuth service** or alliance auth if one exists, but that's likely overkill and out of scope (and would conflict with keeping things lightweight).

In summary, the plan emphasizes **quick iterative development**: start with the simplest working slice of the idea (one or two queries answered with live data) and progressively broaden the capabilities based on real needs. By focusing on core features first and leveraging existing services and free tiers, you avoid sinking cost or time into premature optimizations. *Sentience* can grow organically – from a rough chatbot that shows your ISK, to a polished AI co-pilot that players wonder how they lived without. Good luck, and fly safe o7!

Sources: The information above was informed by EVE's official API documentation and third-party tool examples, as well as industry best practices for integrating AI: - EVE Swagger Interface (ESI) details – data coverage and OAuth usage [1] [28]
- Examples of existing EVE tools and assistants (EVE OS, Alexa skill, Slack bot) [12] [13] [14]
- AI co-pilots in gaming (Gaming Copilot, Neural Nexus corp) [20] [16]
- Firebase vs Supabase trade-offs for startups [35] [25]
- OpenAI integration mechanisms (custom GPT actions and plugins) [4] [5].

1 2 10 27 28 29 30 31 32 34 EVE Swagger Interface - EVE University Wiki
https://wiki.eveuniversity.org/EVE_Swagger_Interface

3 Celeo/Preston: Python 3 library for accessing EVE Online's ESI API
https://github.com/Celeo/Preston

4 5 Custom GPT Actions in 2025: How AI Agents Are Taking the Lead | Lindy
https://www.lindy.ai/blog/custom-gpt-actions

6 7 8 9 GitHub - zKillboard/eveonlineoauth2: Simple SSO Library for Eve Online using OAuth2
https://github.com/zKillboard/eveonlineoauth2

11 16 17 Eve Online player corp names ChatGPT its new leader | Polygon
https://www.polygon.com/23979964/eve-online-chatgpt-ai-neural-nexus-corporation

12 13 14 15 23 24 Third-party tools - EVE University Wiki
https://wiki.eveuniversity.org/Third-party_tools

18 19 20 Gaming Copilot: AI Companion on Steam
https://store.steampowered.com/app/3145640/Gaming_Copilot_AI_Companion/

21 Razer AI Gamer Copilot: Your Gaming Companion
https://www.razer.com/software/razer-ai-gamer-copilot?srsltid=AfmBOoplhOvz3QW5zdKbUcA13vxFn7XgAb2BCOV49AeHQyp4PAa0Vgqs

22 Microsoft unveils Copilot for Gaming, an AI-powered ... - PC Gamer
https://www.pcgamer.com/software/ai/microsoft-unveils-copilot-for-gaming-an-ai-powered-ultimate-gaming-sidekick-that-will-let-you-talk-to-your-console-so-you-dont-have-to-talk-to-your-friends/

25 26 35 Supabase vs Firebase: Choosing the Right Backend for Your Next Project
https://www.jakeprins.com/blog/supabase-vs-firebase-2024

33 I asked ChatGPT about intergration with EVE - General Discussion - EVE Online Forums
https://forums.eveonline.com/t/i-asked-chatgpt-about-intergration-with-eve/404161