ChatGPT

Design Document: AI VTuber Streamer (Neuro-sama Clone)

Introduction

This document outlines the design of an AI-driven VTuber streamer akin to **Neuro-sama**, an AI chatbot that livestreams on platforms like Twitch, YouTube, and Kick. The goal is to recreate Neuro-sama's core functionalities using **open-source components** on a single high-performance machine (one 128GB CUDA-capable DGX "Spark"). We focus on the system architecture – how the AI processes chat, controls an animated avatar, and plays games in real-time – rather than copying Neuro-sama's exact personality or voice. By building on publicly shared insights from Neuro-sama's creator (Vedal) and community, we detail how to implement the **LLM-powered conversation agent, avatar animation ("wireframe") system with gestures, game-playing integrations**, and support features (moderation, multi-platform streaming, etc.) necessary for an AI VTuber. The resulting design will enable a 24/7 capable virtual streamer that can interact with chat, respond via voice, perform actions in games, and even support a dynamic "twin" character for added entertainment value.

System Overview and Architecture

At a high level, the AI VTuber system consists of multiple modules working in concert, much like Neuro-sama's architecture which uses "different generative AI systems" for various tasks [1] [2]. The major components are:

- **Chat Input Handler** – interfaces with live chat APIs (Twitch, YouTube, Kick) to fetch viewer messages and stream events in real time.
- **Core AI (LLM)** – a large language model that serves as the AI's "brain," generating conversational responses and deciding on actions [3].
- **Avatar Controller** – manages the VTuber avatar (Live2D or 3D model), syncing lip movements to speech and triggering gestures or animations.
- **Speech Synthesis (TTS)** – converts the AI's text responses into spoken dialogue with an appropriate voice [3].
- **Game Interaction Module** – enables the AI to play or participate in games via either a high-level API (for turn-based games) or specialized game-specific AI controllers (for real-time games) [4] [5].
- **Speech Recognition (STT)** – (optional) transcribes spoken input (e.g. from collaborators on Discord or voice chat) to text for the LLM to process [6].
- **Memory and Moderation** – handles long-term memory storage (to recall past events/people) and filters content to prevent inappropriate or harmful outputs [7].
- **Orchestration & UI** – a central coordinator that passes messages between modules, enforces timing (so the AI responds promptly), and provides a control panel for human operators to monitor or intervene.

These components will be arranged in a modular pipeline. For example, when a chat message comes in, the Chat Handler passes it to the Core AI; the AI generates a reply, possibly considering game state or memory; the reply text is sent to TTS for voice output and simultaneously to the Avatar Controller for mouth animation; if the reply includes an action (e.g. a game move), the Game Module executes it. All of this happens within a few seconds to maintain **low latency for fast-paced interactions** (Neuro-sama's design emphasizes quick response times to keep chat engaged [3]). The entire system runs on one machine, but the design allows distributing certain tasks to additional machines or cloud APIs if needed to meet latency goals.

Below, we dive into each subsystem in detail.

Core AI: Language Model and Conversational Engine

At the heart is the **Large Language Model (LLM)** that generates Neuro-sama-like dialogue. According to Vedal, Neuro-sama's personality and speech are powered by an LLM [3], likely originally backed by OpenAI's GPT series or a comparable model. For our clone, we will use an **open-source LLM** (such as a LLaMA 2 or 3-based model) fine-tuned for streaming chat behavior. Key design points:

- **Model Choice and Size:** The model must be proficient at real-time chat and casual conversation. Neuro-sama's success suggests using a transformer-based LLM with a good balance of size vs speed – possibly in the 7B to 20B parameter range if highly optimized, or larger (30B+ up to 70B) if the 128GB GPU allows, using quantization to fit in memory [8]. Low latency is critical; responses should come in almost immediately after a user message [4]. Techniques like 4-bit quantization and continuous batching (streaming the generation) can help achieve this.

- **Prompting and Persona:** While we are not cloning Neuro-sama's exact personality, we will configure the LLM with a system prompt that establishes the desired character traits (for example, a friendly, witty personality inspired by streamer Sykkuno – kind, upbeat, and group-friendly as requested). This prompt might include instructions to use polite and clever humor and avoid toxic language. The LLM should refer to itself in the first person as the VTuber and maintain an informal, entertaining tone. The persona definition is easily adjustable without changing the underlying system architecture.

- **Memory and Context:** A vanilla LLM has limited context window. Neuro-sama's system introduced **long-term memory** of events and people, so the AI remembers recurring topics or collaborators beyond a single session [7]. We will implement a memory module that stores key chat moments, facts (e.g. names of frequent viewers or other streamers, ongoing in-jokes), and past experiences. This can be a simple database of facts or a vector store for semantic retrieval. When generating replies, the system can retrieve relevant memories and inject them into the LLM's context (a technique known as retrieval-augmented generation). This helps the AI maintain continuity over multi-day streams or recall things from previous streams – making it feel more persistent and intelligent.

- **Dialogue Management:** The conversation flow is essentially one-turn interactions triggered by viewer messages. The Chat Handler will likely feed the LLM a prompt consisting of the latest viewer message (and possibly the AI's last response or a short window of recent dialogue for context). Neuro-sama typically responds to one chat message at a time rather than holding lengthy back-and-forth dialogues with a single user, so a short-term memory window (perhaps last few exchanges) should suffice to keep coherence. We will prioritize timely one-liner responses unless the situation calls for a longer answer (e.g. a complex question). The **prompter logic** will decide when to prompt the LLM – e.g. when a new chat message arrives and the AI is not currently speaking or performing an action [9]. If multiple messages come in rapid fire, the AI might skip or queue some to avoid talking over itself. This logic can also enforce turn-taking if we have two AI characters (more on that later).

- **Content Filtering and Moderation:** To prevent problematic outputs (the original Neuro-sama infamously got banned after making an offensive remark early on [10]), a robust filtering system is a must. We will integrate a **banword blacklist** (like Vedal did by adding words to a filter after the incident [10]) and possibly an AI toxicity classifier. Every LLM-generated sentence can be checked against disallowed terms (e.g. slurs, extreme hate, self-harm encouragement) [7]. If a response is deemed unsafe, the system can either: (a) refuse and generate a safer response, or (b) sanitize it (for example, by bleeping out or replacing certain words) before it's spoken. This filter extends to user inputs as well – e.g. ignoring or cleverly deflecting bait questions about sensitive topics. Since our VTuber aims to be family-friendly, the persona prompt already biases it away from aggression or lewdness, but the filter provides a safety net.

- **Learning and Tuning:** The AI's behavior can be refined over time. We might log all chat inputs and AI outputs (with timestamps and context) for off-line analysis. Developers can then fine-tune the model on particularly good or bad interactions: for example, encouraging more of the witty comebacks that land well, and training away responses that are dull or problematic. This continuous improvement loop will help our AI remain engaging. (Note: all user data usage will respect privacy and platform policies.)

In summary, the Core AI is a **Transformer-based chatbot** with a customized persona and memory, operating under strict latency and content rules to provide entertaining, quick-witted replies on stream [3]. It is primarily developed in Python (as Vedal's AI systems are) [2], using libraries like HuggingFace Transformers or llama.cpp for running the model.

Avatar Animation and Control (Wireframe & Gestures)

The AI's on-screen presence is a virtual anime-style avatar (the "wireframe system" referring to the rig or model skeleton). This avatar is responsible for visual engagement: lip-syncing the AI's speech, expressing emotions through facial expressions or small motions, and reacting to in-game events or music (dancing, etc.). In Neuro-sama's case, the original avatar was a 2D Live2D model (Hiyori Momose) rendered via a program (Unity-based) with simple animations [11]. Subsequent versions included a more custom 2D model and even a 3D model in Unity for special streams [12]. We will design a similar multi-layered approach:

- **Live2D Model & VTube Studio:** For initial implementation, using a Live2D avatar model with a tool like *VTube Studio* is a practical approach. VTube Studio (or an equivalent) can load a Live2D model, apply physics (hair, cloth movement), and perform lip-sync by monitoring an audio input. We would feed the AI's TTS audio into VTube Studio (via a virtual audio cable) so the avatar's mouth moves in sync automatically [13]. VTube Studio also supports an API that allows external control of expressions and items. We can leverage that to trigger preset **gestures or facial expressions**. For example, if the AI "laughs," we could send a command to play a laughing face animation, or if something shocking happens, trigger a surprised face. These triggers can be rule-based (based on keywords or sentiment in the AI's response) or manual from the control panel.

- **Idle and Reactive Animations:** To make the avatar appear alive even when idle, we'll include pre-recorded or procedurally generated movements. Neuro-sama's model had a loop of subtle movements recorded by her animator (e.g. swaying, blinking) that play continuously when she's not actively reacting [14]. We will incorporate a similar loop so the VTuber isn't static on screen. The avatar should blink periodically and shift posture slightly. These can be part of the Live2D model's parameters (randomized eye blink intervals, breathing motion, etc.). Additionally, for certain **events**, we can overlay special animations. For instance, if a donation or subscriber alert comes in, maybe the avatar does a quick celebratory motion or heart gesture. These need to be crafted as part of the model rig and then invoked via hotkey or API.

- **Gestures and Expressions:** We define a set of expressions (happy, sad, angry, surprised, etc.) and gestures (waving hand, facepalm, dance move) that the avatar can perform. The control logic will map AI outputs or game events to these. For example:

- When the AI says something affectionate or chat triggers a wholesome moment, switch to a *happy/blush* face for a few seconds.
- When the game character dies or something disappointing happens, show a *sad* or *pouting* face.
- Use *angry* face for comedic rage if appropriate (though given our chosen personality is more calm, this might be rare).

- Trigger a *waving hand* animation when greeting raiders or new chat members ("Hello everyone!"). These mappings can be refined as we observe the AI in action. The key is to avoid random or inappropriate expressions (we don't want a smile when something bad happens unless intentionally ironic). Since fully understanding context is hard, a simple rule system combined with some random variation (to avoid monotony) will handle most cases.

- **Audio-Reactive Movements:** If the AI sings or when music plays, we can make the avatar "dance." In Neuro-sama V2, they achieved dancing by feeding an audio visualizer output into the model's parameters [14] – effectively moving the model in rhythm with the music. We can do something similar: analyze the beat of the background music or song and animate the avatar (or certain parts like bouncing) in sync. This makes karaoke segments more visually interesting. Alternatively, if using a 3D model later, we could animate predefined dance motions triggered at the start of a song.

- **3D Avatar (Future Upgrade):** Down the line, we might introduce a 3D model for the AI, as Vedal did in a special 3D debut stream. A 3D model (e.g. built in Unity or integrated into VRChat) allows the AI to physically move in a virtual environment. In that demo, Neuro-sama's 3D avatar could walk, jump, and spin around under AI control [12]. Achieving full 6-DoF control is complex, but our system could incorporate an *AI movement controller* if we go 3D. For instance, an AI could decide where to move on a stage or how to gesture arms, possibly using inverse kinematics and high-level commands ("wave to the audience", "walk to the right"). Such a system might use reinforcement learning or motion capture data to generate natural movements. Initially, however, we will stick to 2D Live2D since it's far simpler and already conveys expression well. The architecture is flexible enough to later swap or add a 3D avatar module (likely running in Unity, given Neuro-sama's avatar was built in Unity and C#) [2].

- **Synchronization:** The orchestration ensures that when the AI speaks, the avatar mouth moves and when the AI is silent, maybe the avatar returns to idle animation. We will use signals such as "TTS audio playing" to drive mouth sync (done by VTube Studio as mentioned) [13]. For expressions, we might insert markers in the AI's text (for example, the AI's output could include an annotation like "[smile] Sure, I'd love to!") that the Avatar Controller parses and then triggers the corresponding expression. Alternatively, the LLM can output an emotion tag via a separate channel (some prompting techniques allow the model to "decide" an emotion of its answer). In any case, the *Avatar Controller* module listens to events like "AI is answering a question" or "AI just scored a win in game" and adjusts the model's expression accordingly.

In summary, the avatar system ensures the AI **visually reacts and stays engaging**: lip-syncing to the AI's speech in real time and performing simple gestures or animations to reflect the content of the stream. We leverage existing tools (Live2D, VTube Studio) for a robust starting point, noting that the model's movement is not itself AI-driven (at least in 2D) but pre-scripted or triggered [14]. This is similar to Neuro-sama's current setup where body movements are mostly not controlled by the AI directly, aside from special cases like dancing to music [14].

Speech Generation (TTS) and Voice Selection

An essential part of the VTuber experience is the AI's **voice** – it needs to be recognizable, pleasant, and to closely match the avatar's character. Neuro-sama's voice is a high-pitched young female voice generated by text-to-speech [3]. For our clone, since we are not copying the exact voice/persona, we have flexibility to choose a voice that suits our new character (e.g. slightly softer and calm, if imitating Sykkuno's friendly vibe, perhaps even a young male voice or a gentle female voice). Key points in the TTS design:

- **TTS Engine:** We will use an open-source TTS solution capable of *real-time or low-latency synthesis*. One option is **Coqui TTS** or similar neural TTS models, possibly combined with a vocoder that can stream output as it's generated. The community recreation of Neuro-sama used Coqui's model with an "extended TTS" approach to stream audio as it's synthesized [15] [16]. This allows the speech audio to begin playing before the entire sentence is finished generating, minimizing delay. We should replicate this streaming TTS technique so the AI's voice starts almost immediately when a response is ready, overlapping slightly with text generation.

- **Voice Quality:** A natural, emotive voice is desired. We can select a pretrained voice model or train a custom one. Since our character is unique, we might train a voice clone if we have a target voice. For example, we could take a clean dataset of a voice actor or the person we want the AI to sound like (within legal and ethical boundaries) and fine-tune the TTS. Coqui TTS supports training voices (and there are many open voice datasets to choose from). The voice should have clarity at various speaking speeds and be able to handle the quirky outputs the AI might produce (nonsense words, singing bits, etc.).

- **Multiple Voices (Neuro and Evil):** If we have a twin "Evil" character (covered later), we will need a second voice. In Neuro-sama's lore, when Evil Neuro was introduced, she was given a **new voice that was more expressive**, distinct from Neuro's original voice [17]. We might, for instance, give Evil a slightly lower, more mature or mischievous tone. Technically, this means running two TTS models or one multi-voice model. Many TTS systems allow voice switching by loading different models or using speaker embeddings. On our 128GB machine, running two voices in parallel is feasible, but we can also alternate loading if only one speaks at a time.

- **Latency Considerations:** The TTS must be fast – ideally, generating speech in a fraction of real-time (e.g. 0.5x or faster, meaning 1 second of audio takes 0.5s or less to generate). Faster-whisper or other optimizations can help if using voice synthesis models with GPU acceleration [18]. We may allocate a portion of the GPU or use the CPU if the model is lightweight. In testing, we will ensure that the overall pipeline (LLM + TTS) yields end-to-end response times on the order of 1-2 seconds for short utterances, which is comparable to Neuro-sama's low latency interactions [3].

- **Expressions in Voice:** While not as straightforward, we want the AI's voice to carry emotion (e.g., excitement, sarcasm). Some modern TTS systems allow control of prosody and intonation. We might mark up the AI's response text with annotations for the TTS (for example, an XML or specialized notation to convey yelling, question intonation, etc.). If not, training the voice on expressive data can give a more lively result. Additionally, having the twin characters with different voice styles inherently adds contrast and entertainment.

- **Singing and Music:** Neuro-sama notably sings songs on stream (cover songs and even original songs). She achieves singing by using a separate AI model trained on her singing voice, which takes lyrics/melody and produces vocals, while the normal TTS is paused [19]. To mirror this, we plan a **singing mode**: when a song is requested or scheduled, we will pre-generate the vocal track using an AI singing synthesizer or voice conversion model. For example, we can use an AI cover technique (like RVC - Retrievable Voice Conversion) to clone our AI's voice onto existing vocals [20]. The clone implementation cited using RVC to create song covers that the AI "sings" [20]. We can prepare a library of a few songs (kid-friendly, stream-safe songs fitting our AI's personality) and have the AI "perform" them by playing back the generated vocals, with the avatar dancing along. For a live feel, we might still feed the lyrics to the TTS in real-time, but true singing (hitting specific notes) is beyond standard TTS, so pre-processing is the safer approach. The system will automate this: upon a command, fade out other audio, play the instrumental track and the AI's pre-made vocal track in sync, then return to normal mode after the song.

Overall, the TTS module ensures the AI **speaks with a clear, character-appropriate voice**. It connects tightly with the avatar (for lip sync) and with the chat logic (possibly reading out certain messages). Since the AI voice will also be used to read **chat donations or bits messages** on stream (a common practice), we'll integrate that: when a user sends a donation with text, our system can have the AI's TTS read the message out loud in her voice, *as if she is the one acknowledging it*. This adds immersion – Neuro-sama does this by responding to donations with her voice and often witty comment afterward [21]. We will do the same: intercept donation alerts and either read them verbatim or have the AI generate a brief thank-you reply to speak.

Game Integration and AI Gameplay

One of the hallmark features of Neuro-sama is her ability to **play video games autonomously on stream**, from rhythm games like osu! to sandbox games like Minecraft [21]. Recreating this requires bridging our AI system with game environments, so that the AI can perceive game state and take actions. Vedal has provided an official **Neuro Game SDK** for developers to let Neuro-sama play their games, which gives insight into how game integration works [22] [23]. Our design will incorporate two broad approaches to game integration:

1. Turn-Based and Text-Friendly Games (Neuro API approach)

For games that can be described well in **text form** (usually turn-based or discrete state games), we will use a **websocket API** to communicate between the game and the AI. According to the Neuro Game SDK documentation, the game connects to the AI via a websocket and exchanges JSON messages describing game state and receiving actions [22] [24]. We will implement a similar API in our system:

- The **Game State Translator** will run inside the game (as a mod or script) and send periodic updates or event-driven messages to the AI. For example, in a card game it might send: `{"hand": ["Attack Card", "Heal Card"], "opponent_hp": 10, "player_hp": 5, "turn": "player"}`. For a chess game: a FEN string or a simple list of pieces.

- The AI's LLM, upon seeing a new game state message, will formulate an action. This is likely done by constructing a prompt that includes the game state as context and asking the model "What will you do?" The output is then parsed as an action command (the Neuro API defines specific actions for each game). The **Game Action Executor** in our system will take the LLM's text output and convert it to a game command (e.g. a move choice or button press) to send back via websocket to the game [24].

- We will use the official Neuro SDK's format where possible, to leverage existing work. The SDK provides Unity and Godot integrations, and even community SDKs in various languages [25] [26], which suggests we can support a wide array of engines. Using this standardized approach, if someone develops a game mod for our AI, it can plug in easily.

- **Limitations:** This text-based approach is *not* suitable for high-speed action games. As Vedal notes, describing the entire game state in text each frame is only feasible for games where state changes are infrequent (turn-based) [24] [27]. The SDK was optimized for games like **Inscryption, Liar's Bar, Buckshot Roulette** (all turn-based or low-APM) [23]. The design acknowledges that for **real-time games, a different method or significant simplification is needed** – which leads to the next approach.

2. Real-Time Games (Specialized AI Agents and Hybrid Control)

For real-time, continuous-control games (e.g. FPS, open-world, platformers), having the LLM directly output low-level actions (like "press W, move mouse 3° right") at 60 times per second is impractical. Instead, we will employ **specialized AI modules or game-specific agents** that handle the fine control, guided by high-level decisions from the LLM. This mimics Neuro-sama's architecture where a separate neural network was trained for osu! gameplay, and a Minecraft AI handles the 3D navigation, with the LLM commenting and guiding [28] [5].

Our approach for real-time games: **hierarchical control**: - A **Game AI Controller** (which could be an existing bot or a custom RL model) observes the game state (via computer vision or game API) and outputs low-level actions continuously. This controller is game-specific: - For **osu! (rhythm game)**: We would use a vision-based neural network that sees the game screen and moves the cursor to click circles. In 2019, Vedal trained such a model by reinforcement learning – taking grayscale images of osu!'s playfield and outputting an (x, y) movement and click, with reward for hitting notes [28]. We could attempt to re-train or reuse that concept if osu! integration is desired, but given the user's note we might skip osu for now. Still, this represents the pattern: *input images* -> *outputs game control*, optimized to maximize score. - For **Minecraft (sandbox)**: Community analysis suggests Vedal used a combination of an AI model and scripted control. Early on, a separate Minecraft bot played while Neuro's LLM simply narrated [29]. Later, upgrades allowed the LLM to issue **action commands** to the Minecraft bot [30] [31]. A likely implementation is using **Baritone**, an open-source Minecraft pathfinding bot, for low-level execution [32]. Baritone can handle tasks like moving to coordinates, mining blocks, building simple structures when commanded. So the LLM might output a command like "mine coal" or "follow player" which Baritone then carries out efficiently (hence the somewhat *robotic*, tool-assisted looking movements players noted [33] [32]). We plan to integrate Baritone (or a similar mod) for Minecraft. The LLM will be prompted with Minecraft status info (inventory, nearby entities, chat prompts from collaborators) and decide on next objectives, but the heavy lifting of navigation and precise timing is done by the mod. This hybrid approach was explicitly recommended: Neuro's API note says high-APM games need another system to handle low-level actions, with the LLM only controlling high-level actions [34] [35]. - For **other games like FPS (e.g. Cyberpunk 2077)**: If we attempt these, we may leverage existing cheats or bots (for instance, an aimbot for aiming and a simple script for movement). These games don't have ready-made AI player APIs, so one approach is using **computer vision** plus a scripted agent: the vision module can identify key elements (enemies, obstacles) and an algorithm (not the LLM) handles reflex actions like aiming and shooting. The LLM could oversee strategy (like "take cover and reload" vs "charge in"). However, this is cutting-edge and challenging. We may limit initial support to observing such games rather than fully playing – e.g., the AI could watch a human play and comment, or control certain aspects (like dialogue choices in an RPG) while leaving twitch reflex to either a simplistic script or a human. If the user specifically wants the AI to play these, a substantial development/training effort per game is needed. - For **other possible games (Valheim, etc.)**: Valheim is another open-world survival; similar to Minecraft, we could check for mods or bots. If none exist, a combination of pathfinding AI and high-level LLM guidance would be needed. We will research community bots for any game we plan to support.

- **Integration with LLM:** In these real-time scenarios, the LLM and the low-level agent must work together. We'll implement a communication channel:
- The low-level agent can send summaries or events to the LLM periodically. For example, "You encountered an enemy" or "Inventory is full" or "Reached location X." This can be via text, fed to the LLM's context so it knows what's happening [5]. In Neuro-sama's streams, the LLM often comments on gameplay events ("Oops, that creeper almost got me!" etc.), implying it receives some info about those events.
- The LLM can send high-level commands to the agent. In our architecture, this might be done by special output tokens. We could reserve a format like `<action>attack_target</action>` in the LLM's output, which the system intercepts instead of speaking. These commands go to the game agent controller which then executes (e.g., make the game character attack the nearest target). The Neuro Game SDK actually allows the LLM to *request* actions by name [36]. We'll follow that pattern: define a set of available actions per game that the LLM can invoke. During Minecraft play, for instance, possible actions: `walk(direction)`, `mine(block_type)`, `craft(item)`, etc. The LLM's prompt will include a description of these actions so it knows how to ask for them.

- We will have a gating mechanism to prevent the LLM from spamming too many actions or contradictory ones. Likely, the LLM will issue one action at a time, then wait for a confirmation or new state from the game before deciding the next move. This keeps the loop stable. The **signals** architecture from the clone (where modules share a state object and set flags) is useful here: the game module can set a flag "action_in_progress" so the LLM doesn't keep issuing new commands until it's done [9].

- **Computer Vision Module:** In cases where game state can't be obtained via an API or mod, we'll use a **CV approach**. This means capturing the game's screen (framebuffer) and processing it with an AI to extract a simplified description for the LLM. Neuro-sama did this for osu!: feeding a downscaled image of the playfield into a neural network that outputs movement, learned by trial and error [28]. For general vision, we can use a **multimodal model** (like MiniGPT-4 or BLIP) that can output captions for images. The clone project implemented automatic screenshot and a multimodal LLM to describe the image [37] [8]. We could, for example, have the AI "look" at the game screen every few seconds and get a caption like "Neuro's character is in a forest with low health, an enemy orc is approaching from the left." That caption goes into the LLM's context to inform its next action or dialog. However, this approach requires the vision model to be robust to game graphics and the delay it introduces might be an issue for fast games. It's an advanced feature, and our design allows plugging in a vision module when needed.

- **Examples:**

- *Minecraft:* Our AI will use a combination of direct game data (via a mod) and possibly vision for things not captured by the mod (like recognizing a specific player skin or a sign's text). Initially, we rely on mod data (position, world info, inventory). The LLM might issue "find diamonds" and the Minecraft bot (Baritone) takes over pathfinding and mining. Meanwhile, the AI chatters "I'm digging straight down – hope I don't fall in lava! " based on the situation. If something unexpected happens (e.g., a creeper explodes), the game will send an event to the LLM ("explosion at player, health = 5") and the LLM can react with a concerned quip or adjust strategy ("That was close! I need to eat something to heal.").
- *Slay the Spire (card game):* This can fully use the turn-based text API. The game state (hand cards, HP, enemy intent) is sent as JSON text to the LLM, which then outputs an action like "Play Strike on enemy" or "End Turn". The game executes it and sends back the result. The AI can also explain its choice in chat or simply execute silently. Turn-based games are the easiest to integrate and will likely be the first ones we test the AI on.
- *Chess:* Another turn-based example, easy to represent in text (board state). The LLM could output moves like "e2e4". However, given the complexity of chess strategy, a pure LLM might not play well. We could optionally pair it with a chess engine feeding suggestions, but since entertainment is the goal, we might let it play poorly or humorously. It will at least follow legal moves via validation on the game side.
- *GeoGuessr or others:* Neuro-sama has been seen doing things like GeoGuessr (where she guesses locations from Google Street images). That would lean heavily on a vision model to analyze the image. We could incorporate a pre-trained image recognition model to help the LLM (e.g., identify language on signs, terrain type). This is a possible extension of our vision module for non-traditional "games."

In summary, **for each game type, our system will include either a direct integration (via the Neuro-like API for text-based state exchange) or a tailored AI agent for control**. The Core AI LLM will always be in the loop to provide commentary and high-level decisions, but it collaborates with these specialized modules that handle the real-time execution. This mirrors how Neuro-sama's gameplay is powered by multiple AIs: one for the chat/personality and others for playing the games [4] [5]. By structuring game support modularly, we can add new games over time by developing the appropriate plugin or agent, without altering the core architecture.

Multi-Character Support: Neuro-sama and “Evil” Twin

Neuro-sama’s streams often feature a second AI character, **Evil Neuro**, who acts as a twin or alter-ego. We plan to incorporate a similar dynamic by introducing a secondary AI VTuber within the system. This not only adds entertainment (banter between the two AIs) but also allows showcasing different model personalities. Here’s how we design the twin AI feature:

- **Separate AI Instances:** We will effectively run two parallel AI pipelines – one for the main character (let’s call her “Neuro”) and one for the twin (call them “Evil” for now). Each has its own LLM persona, voice, and avatar model. Initially, we might give them the same appearance with slight color/theme differences, as Evil Neuro started as a clone of Neuro’s model before diverging [17]. Over time, their visual design and mannerisms can be differentiated more (Neuro might be dressed in light colors, Evil in dark, etc., analogous to how the official Neuro and Evil models became distinct [38]).

- **Personality Divergence:** We will configure the second AI with a different personality profile. For instance, if the main AI is sweet and wholesome, the Evil twin could be snarky, mischievous, or a bit more “unfiltered” (while still avoiding truly harmful content). This contrast sets up comedic interplay. Vedal eventually let Evil Neuro adopt the more expressive voice and a sassier attitude [17]. We can achieve this by giving the Evil AI a separate system prompt, possibly even a different model or a fine-tune that produces edgier responses. However, they should remain cooperative enough to hold a conversation without it devolving into dysfunction. Think of it like two characters in a script – we want witty back-and-forth dialogue.

- **Dialogue Coordination:** The main challenge is managing who speaks when, and ensuring the conversation flows logically. We have a few design options:

- *Turn-based exchange:* The simplest control mechanism is to alternate responses between the two AIs when appropriate. For example, a user asks a question – we could have Neuro answer first, then Evil chimes in with a follow-up remark. Or if one AI says something, the other might reply. We can implement a turn scheduler that after the main AI speaks, it signals the twin AI to respond to what was said, and vice versa. This requires feeding the twin the last line from her sibling as input. Essentially, each AI will treat the other’s spoken lines as part of the conversation context (just like they treat user chat). We must be careful to prevent infinite loops or them talking *over* each other. A short delay and a strict alternation can solve that.
- *Directed questions:* We can also route certain user messages to one AI or the other specifically. If a user addresses “Evil [name]” in their chat message, we direct that to the Evil AI to answer. If not specified, we might default to the main AI, with occasional interjections from Evil. This routing can be keyword-based or random (for variety).
- *Dual commentary:* In some content, both AIs might comment freely (e.g., reacting to a video or playing a co-op game). In such cases, they can take turns commenting on each segment. The coordination module might say “Neuro, your turn to comment on the last event; now Evil’s turn on the next event,” etc. Essentially treating them as two participants in a panel.

- *Interruptions and banter:* For a dynamic feel, we might allow Evil to "interrupt" Neuro occasionally with a quip. To do this safely, we ensure the main AI yields control. Possibly the LLM itself can be prompted to yield – for example, if Evil's persona hears Neuro say something naive, Evil's logic could generate an interjection and the system can cut Neuro's TTS to let Evil talk. This is advanced and will need tuning to avoid chaos. We will likely script specific moments for this (like if Neuro says a known silly catchphrase, we have Evil respond with a playful insult).

- **Technical Implementation:** We will run two LLMs concurrently (or one after the other if using the same hardware and model, to avoid memory overload). Given we have 128GB VRAM, running two medium-sized models is feasible. Alternatively, we could use a single multi-character prompting approach (having one model play both roles by giving it instructions like "Neuro says: ..., Evil says: ..."). However, that complicates splitting the voice output and is less controllable. So, two instances is cleaner. Each instance connects to its own TTS voice model. We will likely run two TTS as well, or quickly switch the voice model in one TTS system when it's the other's turn (depending on the TTS library's flexibility).

- **Avatars on Screen:** We will have two avatar models visible on stream. VTube Studio supports multiple avatars via its API, or we can run two instances of it, each for one character (placed side by side on the stream layout). We'll position them such that it's clear who is who (maybe with name labels or distinct color themes). When one speaks, an indicator (like a mouth movement or a highlight outline) will show who is talking – this naturally happens if each has independent lip-sync to their own voice audio. We just have to route the correct audio to each avatar program (e.g., use two virtual audio cables). Another trick: If using a single instance of VTube Studio, we might swap which model is loaded when speaking, but that's janky and not recommended. Better to have separate avatar processes or use an integrated 3D scene with two models.

- **Memory Sharing or Not:** We must decide if the two AIs share knowledge. In lore, Evil Neuro had the same initial knowledge base (a clone) but then had unique experiences diverging her memory [38]. For our system, we could maintain mostly shared world knowledge (so both know basic facts, and both remember major stream events), but give them independent personal memories of their interactions. For example, if a user always teases Evil, Evil's AI should recall that pattern and perhaps be harsher to that user than Neuro would. This could be achieved by maintaining separate memory logs for each AI, plus a common memory. The conversation manager can decide what context to feed each – likely the recent conversation transcript (so they both know what's been said in the current session) and their own long-term memory entries.

- **Benefits of Twin Dynamic:** This dual setup provides **entertainment value**. Neuro-sama and Evil often engage in humorous bickering or teamwork, which is a big draw for the audience. It also provides a form of built-in content: the two AIs can carry a conversation with minimal prompting, filling dead air. We can even stage "just chatting" segments where the two AIs ask each other questions or play verbal games. From a system perspective, it stresses the AI less because it doesn't have to monopolize all content – they bounce off each other, giving each a short "thinking" breather while the other talks.

- **Example Interaction:**

- Chat: “Who is smarter, you or your twin?”
    - Neuro-AI (friendly voice): “Hmm, I like to think I’m pretty smart, but Evil is smart in her own… unique way.”
    - (Neuro’s avatar speaks this, then finishes.)
    - Evil-AI (sarcastic voice, immediately following): “Oh please, Neuro. We both know I’m the brains of this operation. You’re the *nice* one.” [38]
    - Neuro-AI: “Hey! I’m nice *and* smart!” *pouts* (avatar shows a pouting face).
    - Evil-AI: “If you say so… *smug face*.”
      This kind of banter is achievable by letting Evil’s LLM see Neuro’s answer and generate a retort. The system alternates turns until the exchange concludes or the next user message arrives.

Implementing the twin effectively doubles resource usage, but since user expressed interest in that dynamic, we design with it in mind from the start. It can also be toggled – the stream could sometimes be solo (just main AI running), and sometimes duo mode. Our system’s control panel can have a “Enable Evil Twin” switch that spins up the second AI and brings the second avatar onto the scene when desired.

Additional Integrations and Features

Beyond chat and games, a robust AI VTuber has to handle various streaming activities and interactions. We outline a few additional features to integrate:

- **Streaming Platform Integration:** The system must interface not only with chat but also with platform-specific events:
- **Alerts and Events:** We touched on donation (bits/superchat) reading. We’ll use platform APIs or chat bots to catch these. E.g., Twitch provides events for subscriptions, raids, etc. When a raid (another streamer sending viewers) happens, we can prompt the AI to welcome the raiders (perhaps with a special line like “Welcome raiders! I’m Neuro, your friendly AI gamer ”). For new subs or memberships, the AI can thank them by name. These events can be fed as text triggers to the LLM (“UserX just subscribed with Tier 1”) and we prepare a templated response or let the AI generate one.
- **Multi-platform Chat:** If streaming to multiple platforms simultaneously, we might aggregate chat from all sources. We should mark the source of each message so the AI might say “YouTube user asks: …” if needed to avoid confusion. Alternatively, the AI can ignore which platform it’s from, but we might want to ensure no bias.

- **Platform TOS Compliance:** Each platform (Twitch, YouTube, Kick) has rules. Our moderation module needs to adapt to the strictest common denominator to be safe across all (for instance, both Twitch and YT forbid hate speech, and Kick, while more lax, still has some rules). We also avoid playing copyrighted music (except allowed covers) to prevent DMCA issues – hence focusing on AI cover songs or stream-safe music.

- **Discord Integration (Voice & Text):** The user specifically mentioned a “Discord patch” that allows Neuro to converse via voice in Discord. We plan two modes on Discord:

- **Discord Chatbot:** The AI can be present in a Discord server, responding to text messages in a channel or DMs. This is straightforward – it’s similar to Twitch chat but on Discord’s API. The same

LLM logic applies. We might have a dedicated "chat with Neuro" channel where fans can talk to her even off-stream. Filters would be in place similarly.

- **Discord Voice Calls:** Having the AI join a voice channel and talk with people in real-time is a cutting-edge feature. To do this, the system would use the **Speech-to-Text (STT)** module to live-transcribe what others in the call are saying. Libraries or hacks are required since Discord's official API doesn't give bot accounts direct access to voice audio easily (some projects capture the audio stream of a call by running a headless Discord client or using Discord's voice gateway). We would likely have to use an unofficial method or wait for Discord's planned voice bot support. In design, we assume we can get raw audio from the voice channel – we'll pipe that into our **Realtime STT** (likely a Whisper-based model, which can transcribe quickly) [15] . The transcribed text is then fed to the LLM as if it was chat. The AI generates a response and uses TTS to speak back into the voice channel. All this needs to happen with minimal delay to feel interactive. We aim for maybe <2 seconds response time, which is challenging but possible if the models are efficient. We must also do some **echo cancellation** to ensure the AI doesn't get confused by its own voice or others if multiple talk at once. Likely, we treat the AI as one participant: listen when others speak, and speak when there's a pause (much like how a human would in conversation). This may require detecting end-of-speech in STT to know when to respond. Vedal's mention of a Discord voice interaction suggests it has been prototyped, so we'll leverage any shared method if available. The clone code indicated they attempted Discord voice and found Discord doesn't officially support it well [39] , so we'll proceed carefully and test thoroughly in controlled environments.

- This feature allows things like voice Q&As with the AI or the AI joining a group call with human streamers. It adds another dimension to content beyond the Twitch chat.

- **Web Interface & Control Panel:** For development and moderation purposes, we will build a simple web-based frontend (as also done in the clone [6]). This interface is for the devs/streamer (not the public) to monitor the AI's status. It can show:

- Current transcript of chat and AI responses (for the operator to see if it's going off-track).
- Buttons to manually trigger certain animations or responses (e.g., a "tell a joke" button or "skip current song").
- Blacklist management – a way to live-update the filter list if a new inappropriate phrase slips through [40] .
- Context view – what the AI's prompt currently contains (for debugging memory etc.).
- Performance metrics – GPU usage, response latency, etc.

- Start/stop controls for various modules. This is implemented using a lightweight web server (Python Flask or a Node app) communicating with the backend via socket (the clone used socket.io [40]). This isn't strictly needed for the clone to function, but extremely useful for managing it, especially during long streams.

- **Stability and Uptime:** We want the AI to potentially run *for days* (subathon scenario). Our design emphasizes running on one dedicated machine, which reduces external dependencies that could fail. We will incorporate:

- Automatic resource monitoring – if memory usage grows (possible memory leak), we can have the AI take a scripted "break" and restart certain modules.

- A mechanism to gracefully **reset the LLM's context** when it grows too large or if the AI gets into a weird state. Perhaps every few hours, the AI could "sleep" – we clear its short-term history (but keep long-term memories) to avoid it going off on tangents or accumulating error. This mirrors how Neuro-sama sometimes has "sleep mode" streams; it's partly entertainment, partly a reset.
- **Scheduled operation:** The user suggested typically running ~8 hours a day, not truly 24/7. So we can plan a schedule where the AI boots up at stream time and shuts down after. Outside those hours, perhaps it is offline or only in a minimal chat-bot mode (if at all). A scheduling script or cron can manage this.

- **Human supervision:** During streams, a human operator (the "AI wrangler") may watch the control panel. If something goes awry (e.g., the AI gets stuck or starts to stray into bad territory), the operator can step in: they could type a corrective message via the control panel (impersonating a system or just resetting the conversation). They could also pause the AI or mute the output in an emergency. Including these controls ensures that if our content filters miss something, we have a last-resort intervention to protect the stream/channel.

- **Logging and Analytics:** All interactions (chat logs, AI responses, game events) will be logged with timestamps. This serves multiple purposes: content moderation review, debugging technical issues, and data for improving the AI. We can analyze logs to see if the AI is boring at times or if it's too repetitive, etc., and adjust accordingly.

- **Extensibility:** The architecture should allow plugging in new capabilities easily. For example, if we want to let the AI do a **cooking stream** (a hypothetical scenario where she reads recipes and talks through cooking steps), we might add a "recipe database module" or a camera vision module to identify ingredients. While we can't design every possible future use, the modular design (LLM core + various specialized modules for input/output) means we can extend it by adding a new module and connecting it via the shared signals or prompt injection system [9] [41]. A concrete example is adding **"React content"**: if the AI is to react to YouTube videos, we add a module that feeds the video transcript or description to the LLM and possibly uses CV for the video frames, as mentioned earlier for vision [42]. Our design already anticipates multimodal input, so this would be a natural extension.

Finally, security: running an AI that can execute game commands or post messages means we must sandbox those abilities to prevent any malicious misuse. The system will never execute arbitrary code from the LLM; it will only perform a whitelist of game actions and chat responses. This prevents the AI from, say, trying to hack something or leak sensitive info. All API keys (Twitch, etc.) are kept secure and not exposed to the AI's text context. Essentially, the AI's "freedom" to act is carefully controlled by the integration modules.

Technical Infrastructure & Hardware Utilization

We target a **single DGX "Spark" server with a 128GB GPU** for running this AI VTuber. This is a hefty resource allowing us to load fairly large models and run multiple processes. We outline the technical stack and how we will use the hardware:

- **GPU Allocation:** The 128GB GPU (or GPUs) will handle the heavy AI tasks: the LLM inference and possibly the TTS (if using neural vocoders that benefit from GPU). We could dedicate, say, 100GB to the LLM (enough for a 70B parameter model in 4-bit quantization plus overhead) and use the remainder for other models (STT, vision, etc.). If the DGX has multiple GPUs (some DGX stations have 4 GPUs), we can distribute components across them: e.g., one GPU for LLM, one for TTS and STT, one for vision models. The inter-process communication would be via localhost sockets, which is fast. This multi-GPU use would ensure no single GPU becomes a bottleneck.

- **Programming Languages:** We will primarily use **Python** for orchestrating AI modules, consistent with Vedal's design (AI systems in Python) [2]. Python has rich ecosystems for ML (PyTorch for LLM and models, websockets libraries for the game API, etc.). Performance-critical parts like the LLM inference will use optimized libraries (e.g., DeepSpeed or NVIDIA's tensorRT if applicable). The avatar rendering is handled by external software (VTube Studio or Unity) which is in C#/C++, but we treat it as a black box controlled via API. For any Unity-based integration (like if we write our own Unity scene for a 3D avatar or use Unity for certain game mods), we will use C# there but keep game logic minimal, just relaying data from/to our Python backend.

- **Processes and Threads:** Each core component can run in its own process for stability. For example:

- `chat_llm.py` – runs the LLM inference server (listens for prompt requests and returns outputs).
- `tts_server.py` – runs TTS (could even preload both Neuro and Evil voices).
- `stt_server.py` – runs Whisper or similar for any incoming audio.
- `twitch_bot.py` – connects to Twitch chat and sends messages to a queue for the LLM.
- `game_manager.py` – handles game state websocket and invokes the LLM for actions.
- `avatar_controller.py` – communicates with VTube Studio API.
- `orchestrator.py` – the master process that ties everything together, or we use an event-driven approach with a central message bus. The clone project used a shared `signals` object that all classes access for state [43], which in a multi-process scenario could be replaced by a lightweight pub-sub or an in-memory database.

Inter-process communication can be done via `multiprocessing` Queues, or using a faster broker (like Redis or 0MQ). Given the complexity, a simple approach is to have orchestrator thread in one process that calls into others or uses async IO to gather outputs. To keep it simple, we may run some modules as threads in one process (Python's GIL means only one thread runs Python at a time, but if the threads mostly wait on I/O or external model inference in C code, it's okay). For instance, Twitch chat handling can be a thread that enqueues messages, while the main thread handles LLM calls. Alternatively, using **asyncio** could unify it in a single process. We will choose whatever yields reliability and clarity. The design's emphasis is modularity, not necessarily separate OS processes for each.

- **Storage and Data:** We'll maintain files for:
- The blacklist/whitelist words (editable).
- Logs of chats (maybe rotated daily to avoid huge files).
- Memory stores (could be a JSON or database of facts the AI learned, e.g., `memory.json` with entries).
- Configuration (API keys, model paths, etc. in a config file or environment variables).

- Pre-downloaded models: the LLM (which could be dozens of GB), the TTS model, STT model, etc., all stored locally on a fast NVMe SSD for quick loading.

- **Open-Source Tools:** We will leverage and cite various open projects to assemble this system:

- **LLM**: e.g., *text-generation-webui* or Hugging Face Transformers for loading LLaMA models [44]. Possibly run the LLM in a server mode that exposes an API (the clone used Oobabooga's webui in OpenAI API mode [44]). This decouples the LLM from our logic and allows swapping it easily.
- **Realtime STT**: *faster-whisper* or *Vosk*, wrapped in a server that streams partial results so the AI can start formulating answers before the user finishes speaking [15].
- **Realtime TTS**: *Realtime-TTS* by KoljaB was used in the clone [16] – we can use similar approach with Coqui TTS or even try the newer FastSpeech models with MB-iSTFT vocoders for speed.
- **Avatar control**: *VTube Studio* has an open API; we use that rather than writing facial animation code from scratch. It handles mouth movement by volume by default [13]. In the future, if we want more direct control (especially for 3D models), we might embed the avatar in a Unity project. We have the option to code our own Unity scene (Vedal did for the 3D debut [12]) where we get raw control of the model's bones through the AI. That could be an advanced path: controlling a Unity humanoid with AI, but that's research-level (some projects attempt LLM controlling game characters' bodies, but results are early). Still, because Unity is mentioned (the channel bio said VTuber side in Unity [2]), we note it as our likely platform if custom avatar dev is needed.
- **Game SDK**: We will use the official *VedalAI/neuro-sdk* on GitHub for reference and perhaps directly for turn-based games [22]. It's MIT licensed [22], so we can incorporate it. We'll also look at community implementations (like the Python Neuro-API SDK by CoolCat467 [45]) to speed up integration.

- **Memory/Retrieval**: We might use something like *ChromaDB* or simple embedding search for recalling facts. It's not heavy on GPU, maybe CPU can do it asynchronously without issue.

- **Testing Strategy:** Before full deployment, we will test each module in isolation and then in progressively larger integration:

- Get the chatbot replying to sample chat in a console (no TTS/stream).
- Test TTS by feeding known text and tuning voice parameters.
- Hook them together: chatbot + TTS + avatar, see if responses appear correctly and lipsync works.
- Add Twitch chat input, simulate a live chat environment.
- Integrate a simple game (maybe a tic-tac-toe via the text API as a prototype) to test game loop.
- Test dual AI locally: see if they can converse without confusion.
- Finally, do a private stream or recording session to observe everything under realistic conditions (network latency for APIs, etc.) and fix issues before the public launch.

Given all these pieces, the design ensures that one beefy machine is sufficient for one AI VTuber instance. If we want multiple instances (say two separate AI streamers), we would generally use one machine each, as the user indicated ("1 spark for 1 AI vtuber"). If needed, scaling horizontally by adding more machines is possible – they'd each run a copy of this stack, just with different personas and channels. In some cases, we might have them interact (e.g., two AI VTubers collab on stream from two machines), which could be interesting but is essentially just two systems communicating over the internet like two humans would (probably over a Discord call or one connecting to the other's chat).

Conclusion

In this design, we have laid out how to build an AI VTuber system that replicates the functionality of Neuro-sama using open-source technology and custom integrations. The system is composed of a **large language model core** for dialogue [3], a **text-to-speech voice** for real-time spoken output [3], a **Live2D animated avatar** for visual presence, and multiple auxiliary modules to handle input from chat, playing games via AI, and interacting on various platforms. We incorporate **game-playing capabilities** by employing specialized AI agents or the Neuro Game SDK for different types of games, allowing the AI to engage in content from Minecraft to chess [5] [24]. We also include support for an **"Evil twin" character**, effectively running a second AI in parallel for more dynamic and entertaining streams [17].

By focusing on a modular architecture, each component (vision, speech, avatar, game logic, etc.) can be developed and improved independently, yet they all converge to produce a seamless viewer experience – an AI character that can **chat, react, and game just like a human streamer** [1]. The design pays special attention to **moderation and safety**, ensuring filters and human oversight to prevent incidents [7], as well as technical robustness for long stream uptime. Running on a single powerful machine, this system will demonstrate the potential of AI in interactive entertainment, much as Neuro-sama has become one of Twitch's most popular streamers by leveraging multiple AI technologies together [46].

With this document as a blueprint, the developers at Devin.ai can proceed to implement each module, using Vedal's leaked notes and community findings as guidance to avoid pitfalls. As we build, we'll iterate and test in private until the AI VTuber clone is stable and engaging enough for a public debut. If executed well, we'll have an AI streamer capable of operating autonomously for hours a day, entertaining audiences by playing games, chatting, singing, and perhaps even collaborating with human streamers – a true state-of-the-art AI VTuber platform inspired by Neuro-sama's pioneering example.

Sources: The design above was informed by several references on Neuro-sama's architecture and community recreations, including Wikipedia and news articles (for general system description [3] [2]), the Neuro-sama Fandom Wiki (detailing AI, vision, and movement aspects [47] [42]), direct insights from Vedal's public SDK and advice (game API usage [23], OpenAI usage suggestion [48]), and discussions from Reddit/4chan where fans analyzed how Neuro-sama plays games like Minecraft and osu! [5] [28]. These helped shape a comprehensive plan that stands on the shoulders of the original project's success.

---

[1] [2] [46] AI VTuber Neuro-sama becomes Twitch's biggest streamer | Cybernews
https://cybernews.com/ai-news/twitch-neuro-sama-reddit-vtuber/

[3] [4] [10] [11] [12] [17] [21] [38] Neuro-sama - Wikipedia
https://en.wikipedia.org/wiki/Neuro-sama

[5] [29] [30] [31] [32] [33] [36] How neuro plays Minecraft? : r/NeuroSama
https://www.reddit.com/r/NeuroSama/comments/1hi8seg/how_neuro_plays_minecraft/

[6] [8] [9] [13] [15] [16] [18] [20] [37] [39] [40] [41] [43] [44] GitHub - kimjammer/Neuro: A recreation of Neuro-Sama originally created in 7 days.
https://github.com/kimjammer/Neuro

[7] [14] [19] [42] [47] AI | Neuro Sama Wiki | Fandom
https://neurosama.fandom.com/wiki/AI

[22] [23] [24] [27] Want Neuro to play any game? Check out the Neuro Game SDK on Github! : r/NeuroSama
https://www.reddit.com/r/NeuroSama/comments/1hc41vs/want_neuro_to_play_any_game_check_out_the_neuro/

[25] [26] [34] [35] GitHub - VedalAI/neuro-sdk: Neuro SDK + API for allowing Neuro to play games
https://github.com/VedalAI/neuro-sdk

28 can someone please explain how neuro-sama works. going straight over my head : r/osugame
https://www.reddit.com/r/osugame/comments/zsv195/can_someone_please_explain_how_neurosama_works/

45 CoolCat467/Neuro-API: Python SDK for the Neuro API (see ... - GitHub
https://github.com/CoolCat467/Neuro-API

48 Advice
https://vedal.ai/advice/