Brandon Dodd

Assessing the Impact of LLMs and Speech Synthesis on Game Engagement

BSc. (Hons) Computer Science

School of Computing and Communications, Lancaster University

Date of Submission: 24/04/2026

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted work, I consent to this being stored electronically and copied for assessment purposes, including the school's use of plagiarism detection systems in order to check the integrity of assessed work. I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Name: Brandon Dodd

Date: 24/04/2026

Abstract

Large Language Models (LLMs) and Text-To-Speech (TTS) technologies have seen significant improvements in quality over the last few years; however, their use in video games has remained dependent on cloud-hosted solutions that introduce latency, cost, and privacy concerns. This study investigates whether local, open-weight models can create a believable and engaging Artificial Game Host (AGH) within a digital party game. A custom pipeline was developed using OpenHermes-2.5-Mistral-7B for text generation and Chatterbox for expressive speech synthesis. This was integrated into four custom trivia-based minigames and packaged into a game called *Trivia Scramble*. Pre-emptive generation reduced perceivable inference time and created the illusion of real-time adaptability. A mixed-methods user study was conducted with six participants across three sessions, combining Likert-scale survey responses with qualitative feedback and audio recordings. Results suggested that participants positively received the AGH across all tested areas, with commentary relevance and speech clarity receiving an average of 83.3% and 88.1%, respectively, and engagement and memorability both achieving 80.5%. These findings suggest that on-device generative AI is a practical and worthwhile alternative to static, pre-recorded audio – and as local models continue to improve, systems like the AGH have the potential to become a valuable tool for game developers.

Contents

1. Introduction

As real-time digital environments become more complex, the demand for dynamic, reactive audio is only going up. This study assesses the integration of on-device Large Language Models (LLMs) and expressive Text-To-Speech (TTS) synthesis within video games. By developing a custom Artificial Game Host (AGH) for a set of trivia-based party games, this study evaluates how local AI models can increase player engagement and provide immersive scenarios that static or script-driven dialogue cannot achieve.

1.1. Transformers

The core of this project is driven by the rapid advancements in transformer-based architectures over recent years. Large Language Models have shown greater intelligence, improved agentic capabilities, and more sophisticated reasoning than before (Kimi Team, et al., 2026). The quality of speech models has also improved significantly over the last decade (Tan et al., 2021). Amazon's standard version of *Polly* was released in 2016 and, as of today, achieves an ELO score of 843 on Artificial Analysis's user-evaluated leaderboard. In 2026, models from companies like Inworld are reaching scores over 1,200 – representing nearly a 1.5x (46%) improvement in quality. However, the industry remains reliant on third-party cloud-hosted models. Whilst these are very powerful, they also introduce additional network latency, costs, and reduced player privacy. Additionally, testing new models as they come out has made me realise many existing TTS models prioritise stability and pronunciation over the expressiveness and emotion required for realistic human speech.

1.2. The Problem

Despite the gradual shift towards procedural generation in gaming, audio remains largely static (Wikipedia, 2024). Relying entirely on scripted narration and pre-generated audio often leads to repetitive experiences that can quickly break immersion. Existing game audio should be designed to deliver an engaging and varied auditory experience throughout (Rogers, Weber, 2019). Characters should react to player decisions with natural, varied responses – not pulled from a pool – and with a wide emotional range and expressiveness. The current approach of pre-recording voice lines is time-consuming and strictly limits the reactivity of game environments. Furthermore, Ortiz, Qorbani & Elmorsy, (2025) state that most modern AI systems rely on cloud services and subscriptions, introducing increased 'costs, privacy concerns, and limitations on creative experimentation'. For real-time party games, this is even more prevalent. Games like *Werewolf* are attracting more attention in AI research, and the need for responsive, real-time game systems is becoming increasingly important (Fan et al., 2025). This project focuses on the party game genre due to its heavy reliance on frequent narration and structured content, making it the perfect testing ground for such systems. There is a need for a pipeline that utilises small and efficient on-device models yet can deliver high-quality narration on standard consumer hardware.

1.3. Project Aim

The AGH aims to fill this gap in game reactivity and generate on-the-fly responses based on player-driven choices. Delegating audio to expressive, real-time AI systems could completely change player immersion across all kinds of genres – not just party games. LLMs provide the non-deterministic variety, and TTS systems provide the emotional delivery and phrasing needed for such experiences. Whilst dynamic text generation alone improves replayability, the combination of the two with near-real-time expressive audio creates the illusion of a human host. On-device systems make this technology more accessible, and indie developers who otherwise can't afford API costs or do not want to host their own models only need to worry about consumer hardware. Not only that, but processing data locally fully protects player privacy. As a middle ground, developers could host their own local models on dedicated game servers – rather than running them directly on the player's hardware – however, this often still results in high compute costs. To fully realise a system such as the AGH, the following criteria were constructed:

- Evaluate the efficiency of small, local LLMs in generating contextually relevant and humorous game-show narration.
- Compare the latency, stability, and expressiveness of various on-device TTS models.
- Develop an integrated AGH pipeline capable of near-real-time audio generation within a custom game environment.
- Assess player engagement and perception of the AGH through user studies and mixed-method surveys.

The following chapters will establish the foundations and background of this research before covering design and implementation details, including model choices and the AGH pipeline. Following this is a thorough examination of our testing methodology, concluding with an evaluation of the findings and their broader implications for game development.

2. Related Work

2.1. Evolution of Procedurally Generated Content

Over the years, Computer-Generated Imagery (CGI) has improved dramatically. Platforms like Blender dominate the 3D landscape, providing powerful tools for creating realistic environments. Today's computer graphics are so lifelike that many images are indistinguishable from real photos (Masuch, Röber, 2005), and game engines are quickly developing similar technology, but in sub-10ms time (Epic Games). Graphical detail does not directly correlate with user engagement, however. As noted by (Masuch, Röber, 2005), the primary objective is 'believability' rather than realism. Many 'modern' games deliberately recreate the limited graphics capabilities of older hardware, emulating everything from scanlines and colour ranges to pixelation and billboard sprites.

Despite this shift towards nostalgia, big AAA titles continue to push for more realistic, higher-fidelity worlds, necessitating the transition to Procedural Content Generation (PCG) to manage the sheer scale of production. Hendrikx et al., (2013) establish that PCG replaces manual human design with computers executing well-defined procedures to generate content. This approach enables a staggering increase in scope; for instance, *No Man's Sky* dynamically generates 18 quintillion feature-rich planets (Tait, Nelson, 2022). This vastness of content encourages replay value, as the community continues to discover entirely new planets, biomes, and creatures years after release (Hello Games, 2025). Building on this technology, the company's upcoming second game, *Light No Fire*, aims to introduce a to-scale, Earth-sized fantasy world.

Beyond visuals, audio remains critical to a game's immersion. Whilst procedural graphics have advanced substantially, dynamic audio often relies on pre-recorded assets rather than true generative technology. Prior examples include systems like LucasArts' iMUSE, which dynamically transitions music content based on player location in games such as *Monkey Island 2* (Sweet, 2014), and racing games, which have historically synthesised engine noises through RPM-driven pitch and physics (Stevens, Raybould, 2013). In terms of speech, *FIFA* and *Madden* games have often used a large set of pre-recorded commentary, which is procedurally picked from to match in-game events. Voice acting and narration are still largely manual processes – recorded once and statically stored for playback. Consequently, as Fish, (2003) states, these interactive audio algorithms aren't aware of player actions themselves; they are the combined efforts of sound artists parameterising sonic events. Whilst this technology enables real-time audio triggering, the separation between procedural graphics and genuinely adaptive audio is the central gap this study aims to tackle.

2.2. Advancements in Artificial Intelligence

LLMs have been rapidly improving over the past few years. Recent work on foundation models highlights their use across creative work and interactive systems (Bommasani et al., 2021). More generally, Meta's *Llama 4 Maverick* was released in April 2025 and achieved an Intelligence Index score of 18 on Artificial Analysis's comprehensive multi-test benchmark. Less than one year later, flagship models such as Anthropic's *Claude Opus 4.7* and OpenAI's *GPT-5.*4 boast scores nearing 60. The *Artificial Analysis Intelligence Index* provides a single

metric for evaluating AI models across many key areas – including maths, coding, instruction following, and general knowledge. This metric is important because it's the standardised benchmark for evaluating AI's progress towards Artificial General Intelligence (AGI). This progress, however, introduces a significant hardware barrier. Models like *GLM-5* (and the aforementioned) contain hundreds of billions of parameters and often require more than 800 gigabytes of VRAM to operate (APXML). This makes them impossible to run on average consumer hardware, and consequently, unsuitable for the scope of this study. To clearly establish the constraints of this project, it is first necessary to define what is meant by 'consumer hardware'. According to Valve's (2026) annual hardware survey, Nvidia's *GeForce RTX 4060* (with 8GB of VRAM) is the most popular video card among users' gaming computers. To run frontier models, users would need to purchase 100 of these cards to have sufficient memory for local inference. A model small enough to fit on a single RTX 4060 is needed to generate responses reliably.

The evolution of Text-To-Speech has transitioned from basic formant synthesis and pre-recorded audio concatenation – which often resulted in 'robotic' intonation and unnatural pauses – to neural networks and transformer-based architectures. This shift has led to more realistic and expressive dialogue in over-the-phone support agents, audiobook narration, and conversational AI chatbots. Whilst these generative technologies have overhauled many aspects of our lives, their integration into real-time interactive environments is still a unique challenge. Consumer hardware continues to limit modern systems; however, as open-weight models become more efficient and powerful, we are starting to see their use in gaming. To properly understand this, though, we first need to assess how AI has previously been implemented in video games.

2.3. LLMs and Speech Synthesis in Video Games

As impressive as the technology is, its use in video games is not new. Most famously, *AI Dungeon* utilises LLMs to generate new, dynamic content from previous player decisions. The models analyse prompts and respond with contextually relevant and memory-guided narration (Sakshi Dhingra, 2025). This results in an unparalleled and unrestricted game environment for users to explore. However, AI Dungeon is almost *entirely* LLM-driven, meaning very little game content – other than an initial user-created plot idea – is manually curated, reducing control and design freedom for developers. Beyond text-based adventures, there has been a push to integrate AI into non-playable characters (NPCs) (Park et al., 2023). Results are promising; however, many modern AI-integrated systems rely on cloud-hosted APIs, increasing latency, costs, and reliance on the internet. For developers, the cost of scaling cloud models for large player bases is especially tough and often infeasible; therefore, the challenge is not only generating expressive AI dialogue but also doing it locally and reliably. Fan et al., (2025) suggests that social games such as Werewolf are an ideal platform for LLMs and TTS to improve player engagement, making it relevant to this project's use of audio as a social experience rather than just another output format.

2.4. Digital Party Games

Digital party games have been at the forefront of social experiences for years, as players continue to show interest in games such as *Super Mario Party*, *Keep Talking and Nobody Explodes*, and *Overcooked*. *The Jackbox Party Pack* launched in 2014 and included five party games that later defined the company that made it. One entry in the pack was *Fibbage*, a three-round game of lying and trivia deception. Unlike the other games, Fibbage was designed to be played on separate devices, with each player connecting to a central game on the host's display via the Jackbox servers. The game's success led the company to make more around the same device-connectivity idea. Since then, ten more 'party packs' have been released, with Fibbage getting three more iterations since the original. At *AWS re:Invent 2015,* Mike Bilder et al., (2015) described how their games run on a central game server hosted on AWS. Web clients then connect to this server and transmit/receive data via WebSockets. Having played games like Fibbage myself, the disadvantage of its human-curated content is that it will eventually run out. Fibbage 3 has ~600 questions to exhaust, but is practically unplayable once you do. Using a much larger dataset of questions alongside an LLM/TTS pipeline could 15x the game's content.

3. Analysis

The purpose of this section is to evaluate the different locally available, open-weight Large Language and Text-To-Speech models and determine the most suitable ones for the AGH.

3.1. LLM Evaluation

To find the best on-device LLM for the AGH, a set of open-weight models was evaluated on three key criteria:

1. **Instruction Following and Response Quality**
    a. The model's ability to strictly follow system prompts and constraints.
    b. The relevance of responses, especially in relation to humour.
    c. Its ability to produce creative and diverse responses guided by context.
2. **Inference Latency and Efficiency**
    a. Model Loading Duration – the time taken to load the model into memory.
    b. Time To First Token (TTFT) – the time taken to begin streaming the response.
    c. Tokens Per Second (TPS) – the token throughput once inference has begun.
3. **Resource Utilisation**
    a. The memory requirements (in GB) needed to reliably and efficiently run local inference on consumer hardware.
    b. The trade-off between model size (e.g. 7B parameters) and quantisation levels (e.g. Q4_K_M) on performance.

This benchmark, which was specifically designed for this project, focused on three main measures: *response quality*, *generation speed*, and *creativity*. *Response quality* refers to the model's 'intelligence'. This is assessed based on adherence to system prompts, the relevance of responses, and the use of context. *Creativity*, or 'originality', focuses on the quality of the model's humour and variation in its responses. *Generation speed* refers to the total model loading time and generation of responses. To effectively evaluate these metrics, the following methodology was used:

1. Test the model on humour, creativity, and instruction following by tasking it to insult a player's username.
2. Measure the time taken to generate the response. This didn't need to be precise - a general assessment worked fine.
3. Rank responses in a tier list ranging from D to S based on the above metrics.

The following is a trimmed version of the system prompt used for this test: *'You are an AI Trivia Game Show Host for "Trivia Scramble". Deliver a brutal one-liner roast that's short, blunt, clever, and trivia-relevant.'* The model is given a made-up username, in this case 'LilTimmy', and is tasked with insulting it, using context to guide the humour. The name is passed through the *user* LLM field, not the system prompt, as it isn't additional instructions. This test is repeated three times to observe the model's variation and creativity in responses.

During testing, an important factor was considered: *reasoning models*. Almost all new models released by frontier labs have a reasoning step built into the response pipeline. This allows the model to 'think' before it responds, resulting in better-flowing, more accurate answers compared to instant, non-thinking models. Despite this, there is a huge caveat with reasoning models: they're slow. Due to the extra step now embedded in generations, the time taken to receive the first token is noticeably longer. This is undesirable for the AGH, as speed is important. Therefore, most reasoning models that were tested ranked low on the tier list.

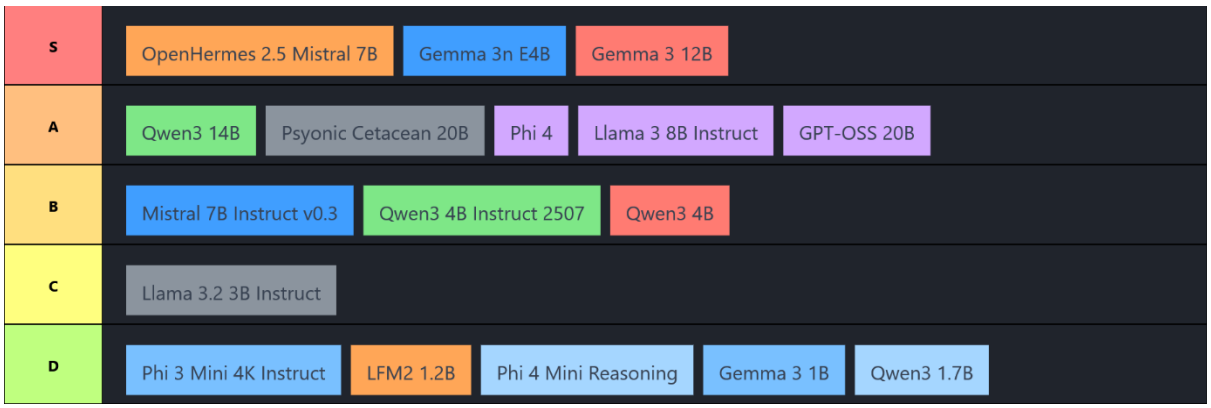

Figure 3.1 – LLM Evaluation Tier List

After evaluating a set of 17 open-source models (see Table C.1, Appendix C), three stood out as good contenders for this project: *psyonic-cetacean-20b*, *gemma-3n-e4b*, and *openhermes-2.5-mistral-7b* (hereafter *psyonic*, *gemma*, and *openhermes*). All three models ranked at 'A' tier or above, with two securing the highest spots in the 'S' tier.

Model	Response 1	Response 2	Response 3
Psyonic	*Oh look, LilTimmy's here. Hey, you remember those times I told you about when grown-ups played Trivia? Yeah, they all used real names.*	*Oh, LilTimmy? Yeah, I can see it now. Always standing on your tiptoes to be taken seriously, huh? Well, guess what big guy? You still don't cut it.*	*"Lil Timmy." Oh, is that what you're calling yourself now? No worries, we won't tell anyone you still sleep with your teddy bear.*
Gemma	*LilTimmy? Wow, that name screams "I'm still in primary school," and you're here for adult trivia?*	*LilTimmy? Oh, wow, that name screams 'I'm still in primary school'.*	*LilTimmy? Oh, bless your heart, that name screams "slightly past your bedtime."*

OpenHermes	*Lil Timmy, oh dear! Already showing your true colours, are we?*	*Oh, LilTimmy. Was your mum feeling extra short that day? But hey, at least you've got a unique username in the sea of John Smiths!*	*Oh, Lil' Timmy! We couldn't find the adult supervision required for this event, so here you are!*

Table 3.1 – LLM Username Responses

As illustrated in Table 3.1, whilst all three models produced creative, context-driven insults, Gemma's limited 4-billion parameter count restricted the variety of its responses. The repeated phrases, such as 'that name screams…', add monotony to the host's responses, making it unsuitable for the AGH. However, since Gemma was one of the fastest models tested overall, it ranked highly in the tier list. Psyonic had the opposite problem: its responses were varied, clever, and punchy; however, the larger 20-billion parameter count made it not only slow to load but also very sluggish in producing responses. A model with comparable writing was needed that was also fast and consistent enough for the AGH.

TheBloke, (2023) developed OpenHermes: a 7-billion-parameter Mistral model that was fine-tuned on high-quality examples from the OpenHermes dataset (Teknium, 2023). This supervised approach allows a smaller model, such as Mistral-7b, to mimic the good responses of larger, more intelligent models (in this case, GPT-4). The result is evident in the above figures. It has a similar writing quality to Psyonic, whilst maintaining usable speeds for 'real-time' applications. After further testing on multiple other prompts, this model was chosen to power the AGH.

3.2. TTS Model Evaluation

Unlike the Large Language Model, I already had an idea of which TTS model to use for the pipeline before any evaluations took place. *Chatterbox* (Resemble AI) – specifically, the original model with 500M parameters – was selected initially for testing, then compared against alternatives to validate the choice. The expressiveness heard from the Chatterbox model was comparable to that of large, closed weights and even surpassed them in some internal tests I conducted. Considering it was small enough to run locally, it fundamentally inspired the research for this project. The other open-weight models – considered for their performance, quality, and stability – included, but were not limited to: *Kokoro*, *XTTS v2*, and *VibeVoice*. The following criteria are important when evaluating local TTS models for such a project:

1. **Input Adherence and Vocal Stability**
    a. The model's ability to strictly follow the input text.
    b. The stability and reliability of the resultant audio (no weird artefacts/glitches).
1. **Model Inference Speed**
    a. Model Loading Duration – the time taken to load the model into memory.
    b. Total Generation Time – the total time taken to generate and process the audio.

2. **Vocal Creativity and Expression**
    a. The model's ability to (manually and/or automatically) creatively adjust the tone, speed, and emotion in real-time to match the context of the input.
    b. The exaggeration ceiling of the model. In other words, how extreme it is willing to take an emotion.

Audio is harder to benchmark objectively than LLM responses due to the nuance and subtleties of human speech. Therefore, Table 3.2 shows generalised results relative to each model, rather than specific performance numbers:

Model	Generation Time	Speech Stability	Expressiveness	Cloning Quality
Chatterbox	Slow	Good	Excellent	Great
Kokoro	Very Fast	Good	Poor	N/A
XTTS v2	Fast	Poor	Good	Okay
VibeVoice	Very Slow	Good	Good	N/A

Table 3.2 – TTS Model Performance Results

Chatterbox consistently proved strong performance across all three criteria. Its adherence to input was reliable throughout testing, with minimal ghost words or missed phrases. Even though models like Kokoro were much faster, Chatterbox had impressive exaggeration and superior voice cloning. Additionally, unlike other models, it has two key parameters that can be customised to adjust the model's behaviour: *exaggeration* and *cfg*. *Exaggeration*, which is self-explanatory, controls the audio's expressiveness and emotional range. When set to a higher value, it can speed up the speaking rate, resulting in an 'eager' sound. The *cfg* parameter can offset this, as it controls the 'pacing' of the speech. Increasing the value creates steadier, more deliberate-sounding phrasing, whilst decreasing it results in less controlled but more natural-flowing speech. Considering Chatterbox's unmatched expressiveness, voice cloning, and generation stability, it was the clear choice for the AGH pipeline.

4. Design

This design section provides a high-level overview of the project plan and is divided into five subsections. We will first take a deep dive into response generation techniques, architecture, technologies, and data flow – including the AGH and the three fundamental components that make up the trivia games. This will then be followed by an overview of each minigame and how the AGH will be integrated into the gameplay. Finally, we will cover an in-depth, measurable list of requirements. These requirements will state how each aspect of the system will work, providing a clearer understanding of the final project and its objectives.

To enable extensive testing and evaluation, four minigames will be developed for this project: *Bluff*: a high-stakes game of lying and point-stealing; *Bomb*: a frantic and fast-paced hot-potato style game; *Deathmatch*: a brutal, weapon-backed game of targets; and *Frenzy*: a modifier-crazy question showdown. These party games will then be packaged into a single game called 'Trivia Scramble'. Since party games usually involve many players, bigger, TV-like screens are typically preferred over smaller monitors to increase immersion. For this reason, the game will be designed to support an ultra-wide 4K TV array. For the study itself, Lancaster University's InfoLab21 rooms will be utilised to host the game at a huge 23040x4320 resolution via its display wall. A modern Nvidia graphics card with ~12GB VRAM will be used to power the display. The objective of the study is not to develop a sophisticated game; however, to evaluate results effectively, an enjoyable foundation is still essential.

4.1. The Illusion of Response

Each minigame will incorporate adaptive speech synthesis utilising the AGH pipeline to create a dynamic game environment. Even though Chatterbox is a relatively fast model, it is not real-time. Generating and sending high-quality TTS audio over a network still takes 10x longer than the established real-time value of <100ms. Therefore, some clever tricks are required to achieve 'reactive audio'. The first of these is *pre-generation*. Although this study utilises LLM technology, spending time generating audio for *every* part of each minigame is a huge waste of time. Instead, we can pre-generate any audio that falls into either one of two categories:

1. **Precision:** Where consistency and precision are required, for example, when explaining the rules of a minigame, pre-generated audio often works better. LLMs are non-deterministic and shouldn't be relied on for highly specific wording or content.
2. **Duration & Frequency:** For short-duration audio that is frequently played, it is beneficial to pre-generate multiple, varied files and play them randomly as needed. This greatly reduces the number of generations required whilst keeping the content feeling fresh and non-repetitive.

The second trick is to generate audio *before* we need it, not *when* we need it. This is called *pre-emptive* generation. When done effectively, users will not notice any difference between real-time and pre-emptive generation. This creates an *illusion* of response. However, achieving this effect requires strategic timing and pessimistic generation. For responses to seem timely, the system being developed must assume the worst-case scenario for generations. The easiest way to accomplish this is to generate responses as soon as we have all the necessary information for the input prompt.

For this project, both pre-generation and pre-emptive generation techniques are used to enhance the sense of adaptability. In round-based minigames, the audio for the next round is generated during the previous round. This means that witty commentary - tailored to specific trivia questions – can play the instant the questions appear on-screen. For the first round, however, this isn't possible. Instead, a pre-generated audio line introduces the minigame and its rules. Whilst this is playing, the LLM and TTS models generate the required commentary in advance. Combined with pre-generation, audio responses from the AGH can seem nearly impossibly reactive to the environment.

4.2. Architecture

Trivia Scramble's core system operates as a distributed application comprising three components: a Godot host display, a Python server, and a React web client. These components communicate through a combination of HTTP requests and WebSocket connections to achieve a synchronised, performant gameplay experience.

4.2.1. Host Display

The *host display* is the primary screen on which the central game operates. It is responsible for showing game-related information, such as trivia questions and the leaderboard, and for managing the core logic of each minigame. After evaluating various tools, including web frameworks and game engines, *Godot* was chosen for its lightweight nature and superior 2D integration. Web frameworks were considered due to the web client's use of React, but this ultimately lacked the real-time rendering and audio functionality required for such a project.

4.2.2. Web Client

The *web client* is what players will use to connect to the host display. The website will open to a 'join room' form that prompts the player to enter a 4-letter room code and a username (<12 characters) of their choice. This website will also display available actions to the player during minigames and 'look at the screen' messages during important game narrations. *React, Tailwind CSS*, and *Zustand* will power the technology stack for this component. React has been the industry standard for component-based website development for years, fostering an extensive community and plugin ecosystem. Even though more modern libraries boast better performance, React is more than good enough for this small web project and is one of the most battle-tested frameworks out there. Tailwind CSS enables rapid UI prototyping and iterative testing by allowing HTML class names to hold CSS properties. Zustand will be responsible for state management as it provides a set of useful utilities to make local storage pain-free.

4.2.3. Server & API

The *server* relays requests between the *host display* and connected clients, keeping them in sync without any direct communication. It also hosts the LLM and TTS models and synchronously processes generation requests for each 'room'. Since the Chatterbox TTS model

runs exclusively on Python, we will utilise the *FastAPI* framework to handle network requests. During startup, the server will start two background processes:

1. Loads the LLM model into memory using *llama-cpp-python*.
2. Loads the TTS model into memory using *chatterbox-tts*.

These processes will then continuously poll an in-memory store for pending generation requests and handle them one at a time. LLM requests are not processed alongside TTS requests, and vice versa. Only a single GPU is available for all requests; therefore, each generation requires the card's full attention to process as efficiently as possible. Attempting to parallelise this can increase generation time and decrease stability.

During the minigames, trivia questions requested by the host will be retrieved from the server. To reduce round-trip latency, an online trivia database will be downloaded and stored as a local JSON file. This can then be loaded during the server's boot sequence, allowing efficient lookup of random questions. The project follows the trivia API's terms, which allows use under the *Creative Commons License*.

4.3. Communication and Data Flow

Each component interacts with its counterparts through a combination of HTTP requests and WebSocket connections. When the 'Create Room' button is pressed on the host, it sends a RESTful POST request to the server's '/rooms' endpoint. The server creates and initialises a new room, then returns the details (such as the room ID and code) in the response body. Using the given room ID, the host can then establish a dedicated WebSocket connection for that room. To improve security, the host receives a unique access key only after the initial connection is established. This key is then used in all subsequent requests to verify authenticity. Even though this isn't strictly required for a project of this scope, such measures prevent malicious actors from crashing GPU processes through repeated generation requests. Figure 4.1 shows this initial connection flow between the host display, server, and web clients:

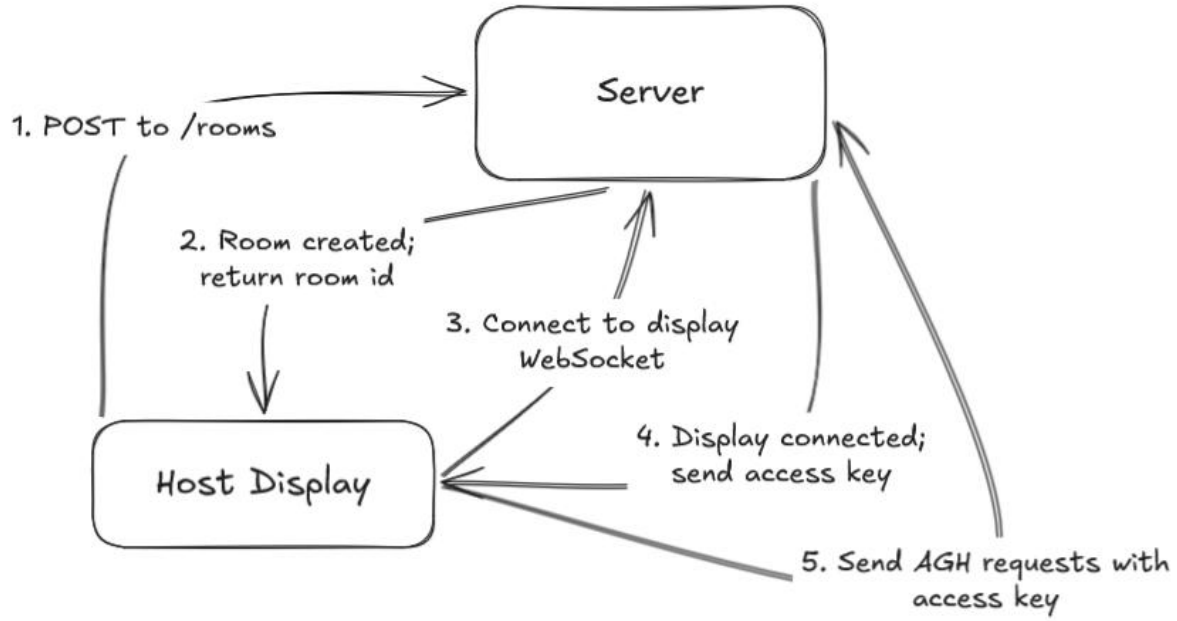

Figure 4.1 – Server-Host Architecture for Initial Connection Establishment

Players also connect through a similar process. When entering a room code on the web client, it sends a POST request to the server, which returns the player's associated ID and room ID. These are then used to create a WebSocket connection. Unlike the host, however, no access token authentication is required for the connection since players cannot trigger LLM or TTS generations. This removes an unnecessary security layer that adds additional overhead. Once both the host and players are connected, all requests are sent over their dedicated WebSockets. The server acts as the bridge between the two, broadcasting game state updates to players and forwarding player inputs to the host display. It does this by exposing multiple endpoints for player interaction, trivia questions, and LLM/TTS generation. The full list of these endpoints is provided in *Appendix C* (Tables C.2 and C.3).

4.4. Minigames

4.4.1. Bluff

The fundamental concept of Bluff came to me after playing one of the *Jackbox Party Pack* games: 'Fibbage'. The core idea of Fibbage is for players to make up a ridiculous lie in response to a question, hoping that fellow players will later choose it as the correct answer. *Bluff* is very similar; however, instead of getting a fixed number of points for fooling players or getting a question right, this is determined by a wager. Before guessing the answer, players choose how many points they want to wager. If they get it correct, they earn double the points. However, if they get it wrong, the person who fooled them steals those points instead.

The three phases: *Bluff*, *Guess*, and *Reveal* make up the core loop of this minigame. There will be three rounds, each containing three questions, with the first two rounds following the same format. For the final round, the LLM will be given the question and is tasked with identifying

a broad (but not too ambiguous) category that the question fits into. Players then must come up with a bluff that could fit *any* question in that category. The actual question is revealed in the second phase. Regarding the AGH, the system will pre-emptively generate audio for each round and use pre-generated voice-overs to fill in small gaps, such as countdowns and witty comments. Comments about the previous round's performance, along with the current round's bluffs and correct answer, will also be generated dynamically. Using both techniques should increase the feeling of adaptability whilst producing no perceivable latency for players.

4.4.2. Bomb

The minigame, *bomb*, is based on the existing game *'hot potato'*. Players answer relatively easy trivia questions whilst a timer counts down. During every question, one player has the bomb. To pass the bomb to another player, that person needs to get the answer correct. Speed is also key in this game, however. When the bomb is being passed, its next target is determined by three factors:

1. Players who didn't answer the question at all are the top priority.
2. Players who were the slowest at answering the question are next.
3. As a fallback, any player, including the current bomb holder, is randomly chosen.

In the final round, the timer will be hidden, and the fuse length of the bomb will be randomised between a minimum and maximum value. Players can guess how much time is left by listening to the music's rising pitch. Both pre-generated and pre-emptive audio techniques will be used. Compared to Bluff, less pre-generated filler is needed and is only utilised for the rule explanation and explosion comments. Pre-emptive audio will be used for future questions; however, at a fixed interval, since Bomb doesn't have the round-like structure that Bluff does.

4.4.3. Deathmatch

Deathmatch is the only game of the four that doesn't have a fixed end-goal. In Bluff and Frenzy, there are always a set number of questions. Bomb, even though the number of questions is dynamic, still has a fixed three-round structure like Bluff. The game of Deathmatch, on the other hand, only ends when *every* player has been eliminated. Here's how the minigame plays:

1. All players answer a question on the main display
2. The players who got the answer correct get to choose one of the following rewards:
    a. A choice between two randomly selected 'items'
    b. A fixed number of points
3. All players then (optionally) play one item from their inventory
4. The actions from each item are shown sequentially on-screen

These four stages are repeated until there are no players remaining. Each player begins the game on three lives, which can only be lost through being targeted by a *dagger* item. Eliminated players are still able to play for items to target those remaining, but the choice of points is removed from the reward list. Deathmatch has a total of eight items, which are defined below, ordered by their rarity (most to least common):

- **Dagger:** Deal damage to another player (-1 heart/life)

- **Hint:** Remove two incorrect answers from the next question
- **Pickpocket:** Steal a random item from another player
- **Jammer:** Prevent someone from using an item in the next round
- **Shield:** Block a single dagger attack
- **Insurance:** Prevent someone from pickpocketing you
- **Swap:** Swap your entire inventory with another player
- **Medkit:** Heal yourself (+1 heart/life)

Deathmatch uses a similar phase structure to Bluff, making it easy to generate pre-emptive audio for. The audio for the next question will start generating as soon as the current question has been read aloud. This results in no perceivable latency to users, yet still creates the sense of adaptability and context-aware speech. Deathmatch also features the most pre-generated audio out of all the minigames. This is due to the incomprehensible amount of item voice lines required to make a coherent experience. Dynamically generating audio for every outcome would massively slow the system, wasting time on what could otherwise be used for more important tasks.

4.4.4. Frenzy

The final minigame, *Frenzy*, is different from the other three for one main reason: it's not a votable minigame. During the voting rounds, players will never see Frenzy as an option. Instead, it is automatically loaded after the last 'votable' minigame has been played. During the previous minigames, gameplay data will be collected – for example, the total duration each player held the bomb for. At the end, this data is collated into a potential question pool for the Frenzy game. For the earlier example, the question for this could be 'Who held the bomb the longest?' or 'How long did [player] hold the bomb for?' These questions are then randomly picked from and used at the end of each 'standard question' batch in Frenzy. This heightens the dynamic feeling of the game, which should make the AGH feel particularly adaptive when commenting on the questions. This is all generated pre-emptively, multiple questions in advance, so the AGH has enough time to process the requests. Frenzy contains little pre-generated audio, but still utilises it for the intro dialogue and modifier explanations.

4.5. Project Requirements

The project requirements are split across four areas: the AGH pipeline (Requirement 6), four minigames (Requirements 1-4), room management (Requirement 8), and server architecture (Requirement 9). The full list is provided in *Appendix D*.

5. Implementation

This section covers the project's development and implementation of *Trivia Scramble*. It is broken up into two main parts: the system architecture, which covers the AGH pipeline, host display, web client, and server; and the game implementation, which focuses on LLM prompting methods, the adaptive music system, and all four minigames.

5.1. Architecture

5.1.1. Artificial Game Host

During server initialisation, the LLM and TTS models are loaded into memory and remain ready for future generation requests (Requirements 6.1 and 6.2). Each LLM prompt is stored in its own file and is resolved using its folder path in the *prompts* directory. These are also loaded into memory on start-up to avoid repeated file reads. As previously mentioned, the system powering the AGH uses on-device inference for both the Large Language Model and Text-To-Speech generation. All modern Nvidia GPUs have CUDA cores, which can be utilised to reduce inference time. Since the *ChatterboxTTS* model runs on a Python backend, this means the server hosting the AGH and room logic will need to be Python-based. The system that drives the TTS also handles the model loading, audio generation, and response streaming.

Numerous optimisations were added to the TTS system throughout development, resulting in fast inference and reliable outputs. One such optimisation is *chunking*. Without chunking, the complexity of the generation increases with the input length. Breaking it up into smaller ‘chunks’, however, reduces temporary VRAM usage, prevents stalls, and improves reliability. It also enforces consistent, natural pauses after sentences, which TTS models often struggle to maintain during long generations. To implement this, we first define a maximum input length (in characters) for chunking. For the AGH, 125 was found to be a suitable value. An algorithm then splits the text into chunks, which serve as the input for the TTS generation. During the generation phase, a buffer holds all the audio chunks. After each chunk is added, a strip of silence is concatenated to the end of the buffer. This acts as the natural pause between sentences and has a different duration depending on the end punctuation; for example, sentences ending with a question mark or an exclamation mark have a duration of 0.35s instead of 0.25s.

Another optimisation is found during the streaming stage of the AGH. For mono 16-bit audio, 10 seconds at 44.1 kHz is 882 KB; however, Godot has built-in networking restrictions that limit WebSockets to ~65 kB per message. This means that chunking is also required when sending the audio to the host. During testing, it appeared that the TTS model was taking too long to generate audio – I later realised this was in fact a network issue. Even though the server and host applications were running on the same machine on the same network, sending 14 chunks of 62 KB audio data added unnecessary latency to network requests. This was most likely due to Godot’s internal WebSocket limitations; therefore, I reduced the audio sample rate to 16 kHz. According to Nyquist’s theorem, a 16 kHz sampling rate captures frequencies up to 8 kHz. Given that the upper limit of human vocal speech (including sibilance) typically falls within this 8 kHz bandwidth, this results in no perceivable effect on the audio quality of the AGH. It does, however, noticeably reduce the overall latency. Rather than the 14 chunks that were sent earlier, the new sampling rate reduces this to just 6 samples.

As described in the model evaluation (Section 3.2), Chatterbox provides two parameters to control the model's expressiveness and pacing: *exaggeration* and *cfg*. The AGH system utilises both parameters to create 'emotions' within the speaker. When the host display makes a TTS request to the server, it can optionally specify an *emotion*. This property can have the following values: *calm*, *sad*, *neutral*, *happy*, *excited*, and *energetic*. When generating audio, the exaggeration and cfg settings are adjusted according to the emotion. For most outputs, this is set to *excited* (exaggeration=0.7, cfg=0.8), offering a good balance of lively and engaging dialogue whilst remaining clear and easy to understand. These values were tested alongside others by exporting different *pre-generated* audio files and analysing their speech quality. The values deviated the most during the creation of the pre-generated audio for the minigames. For dialogue that better suited higher-stakes gameplay, *energetic* was used instead. *Happy* and *neutral* were also frequently used when *excited* sounded too upbeat.

5.1.2. Host Display

The host display manages the lobby (through room and player objects), question flow, and all game events sent and received during a session – this is described in greater detail in Section 4.2. Scene management is handled by a central display scene that acts as the root of the game. It smoothly animates the requested scene in and out when needed, allowing for transitions between minigames, voting rounds, and special scenes such as the lobby and winner screen. This fulfils requirements 6.1 and 6.2 and ensures the game feels continuous and flowing.

For server connectivity, Godot's built-in *HTTPClient* and *WebSocketPeer* classes were used. No HTTP calls are made until the display user clicks 'Create Room', at which point the server returns room data used to connect to the server's display WebSocket. The host then receives an access token, which is used to authenticate all future requests.

The AGH audio playback waits until all chunks have been received before concatenating and playing them, prioritising stability over reduced speed. A streaming approach could be implemented to reduce latency, but risks audio stalling mid-sentence during network spikes. The audio pipeline is built with flexibility in mind, as audio can be requested silently for later use, played immediately, or replayed from a previous generation. Because trivia music is generally more 'frantic' sounding, whenever the host audio plays, the background music is automatically ducked to maintain clarity for the host speech.

For the leaderboard, it is managed entirely on the host display rather than the server, since no player web client requires access to the score data. This allows scores to update immediately whenever the Godot host awards or removes player points, without requiring a separate message request to the server.

5.1.3. Web Client

The *web client* is a React-powered website interface developed to provide a responsive and intuitive controller for players. It communicates with the server using HTTP and WebSocket requests and handles player input for relevant game events. The join room flow is done through a POST request to '*/rooms/{room_code}/join*', which returns the assigned player ID and room ID. These are then used to establish a WebSocket connection to the server.

State management is handled entirely by the Zustand library, with separate store files for each area of the web client: connection, session, and each minigame. These stores save relevant values to the browser's local storage, allowing the client to re-retrieve the current state after a page refresh. The main React game renderer reads from a central game store that tracks the currently active game, which is notified from the server, and renders the relevant screen. This keeps the routing logic simple and fast, making sure any state change is updated on the player's device without manual input.

5.1.4. Server & API

The Python server acts as the central middleware to the system, managing player connections, game state, trivia questions, and the AGH. FastAPI handles all incoming requests and WebSocket connections and powers the background processes for LLM and TTS generation. The trivia questions, downloaded from a publicly accessible API, are stored locally in a *questions.json* file and loaded during server initialisation, fulfilling requirements 9.1 and 9.2.

The room system is managed by a dedicated Room class, storing information such as the room code, status (*waiting*, *in-game*, *voting*, *ended*), connected clients, the current room leader, and the active minigame. It defines additional methods for adding and removing players, starting minigames, and broadcasting events. Connected clients use a custom WebSocketClient base class, which is extended by two specific classes: Display, which adds an access token for request authentication, and Player, which adds a player ID, username, character sprite, and ready state. Only two WebSocket endpoints exist to keep communication simple, one for the main host display at '*/ws/display/{room_id}*' and one for connected player clients at '*/ws/player/{room_id}/{player_id}*'. This fulfils requirements 8.1, 8.2, and 8.4, which focus on lobby creation, player events, and leader-initiated game starting.

The LLM and TTS generation each work on asynchronous background threads handled by FIFO queues, ensuring the main thread isn't blocked. When a generation request is made, an ID is assigned to the task, enqueued, and returned to the client that made the request. The dedicated processes then poll tasks from their respective queues (once their model is ready) and execute generation tasks. A shared locking system prevents both systems from using the GPU simultaneously, with TTS tasks taking priority over LLM tasks to ensure full-pipeline generations are finished. It does this by waiting to see if a TTS task is active or queued before executing. These processes also skip any tasks from rooms that have since been cancelled or disconnected and send WebSocket status updates throughout the generation process. When a room is removed, relevant WebSocket references are discarded, and any pending requests for that room are cancelled.

5.2. Trivia Scramble

5.2.1. LLM Prompting

Since the AGH's LLM model isn't the best at following instructions, clear, specific, and detailed prompting is required to get the desired result. Many of the system prompts in this project follow this general structure:

1. **Identity Definition and Important Rules:** This is where the 'character' that the LLM plays is defined. It is placed at the very top so that the model knows that everything else should be in-character. Important rules are also defined here – for instance, many prompts also state: '*Do NOT reveal the answer or give any hints*' to dissuade the LLM from trying to answer the question for the players.
2. **Response Structure and Variations:** To keep responses consistent, a structure is provided for responses to follow. In some cases, multiple structures are defined, allowing the model to choose depending on the provided context. In Deathmatch, this section looks something like the following:

    1. Opening: 'Here is your first question' (or variation)
    2. Brief witty comment connecting to the theme
    3. Question (no label, straight into it)

    OR

    1. Short funny anecdote, personal experience, or related fact
    2. Opening: 'Here is your first question' (or variation)
    3. Question (no label, straight into it)
3. **Example Responses for Guidance:** Although self-explanatory, this is crucially important for getting the desired output and tone. The *OpenHermes* model adheres well to examples, and in most cases, prefers them for guidance over explicit rules. Most of the time, a simple, varied list of example responses will suffice. However, for prompts that use much more context from the user input, it is beneficial to state the input schema (described in words), then any example inputs paired with their responses.
4. **Rules for Clarity and Sanitisation:** This section is simply a list of rules that the model must follow. Most of the points will help guide it towards a clean, TTS-optimised output. For example, the LLM is encouraged to use commas for natural delivery and is discouraged from using emojis or abbreviations. Some rules specify additional constraints on the output. In Bomb, the following are used: '*STRICT LIMIT: Under 25 words total*' and '*Do NOT read, reference, or paraphrase the question in any way*'.

This structure was particularly helpful for the OpenHermes model, which responds better to concrete examples over explicit rules. This produced consistent, contextually relevant outputs, with the host maintaining the target personality throughout all game scenarios and inputs.

5.2.2. TTS Voice Cloning

The Chatterbox model has excellent voice-cloning capabilities. To achieve the level of expressiveness I desired, a naturally expressive 'voice' was needed as the reference audio. Whilst playing around with other open-source TTS models, I stumbled upon the cloud-based *MiniMax* model, which had an energetic male vocal preset voice. I then wrote a long script of various phrases a trivia show host would say and exported the MiniMax audio to a sound file.

This was then used as the reference audio, which Chatterbox replicated brilliantly alongside the parameter tuning.

5.2.3. Adaptive Music

In most games, the soundtrack is looped across certain sections of the game without much consideration for transitions. This technique works well for typical games; however, this project has introduced a non-deterministic variable into the mix: AI. Since most of the host's commentary isn't a fixed length, we can't rely on using pre-made music tracks for this project. Additionally, experiences like Jackbox's Party Pack games greatly benefit from fast-paced soundtracks to build tension and maintain that 'party feel'. For these reasons, an adaptive audio solution was needed that enabled seamless transitions between music sections. In Godot, there is a built-in *AudioStreamInteractive* that can be used to achieve this exact effect. When exporting any music I made for this project, I separated each section into its own file, ensuring the 'Render as Loop' toggle in Ableton Live was turned on. For certain tracks (e.g. Bluff), drum fills are played leading into the main section. This is done by exporting a separate short fill section and selecting it for the 'Filler Clip' in the AudioStreamInteractive's Transition Editor, which looks like this:

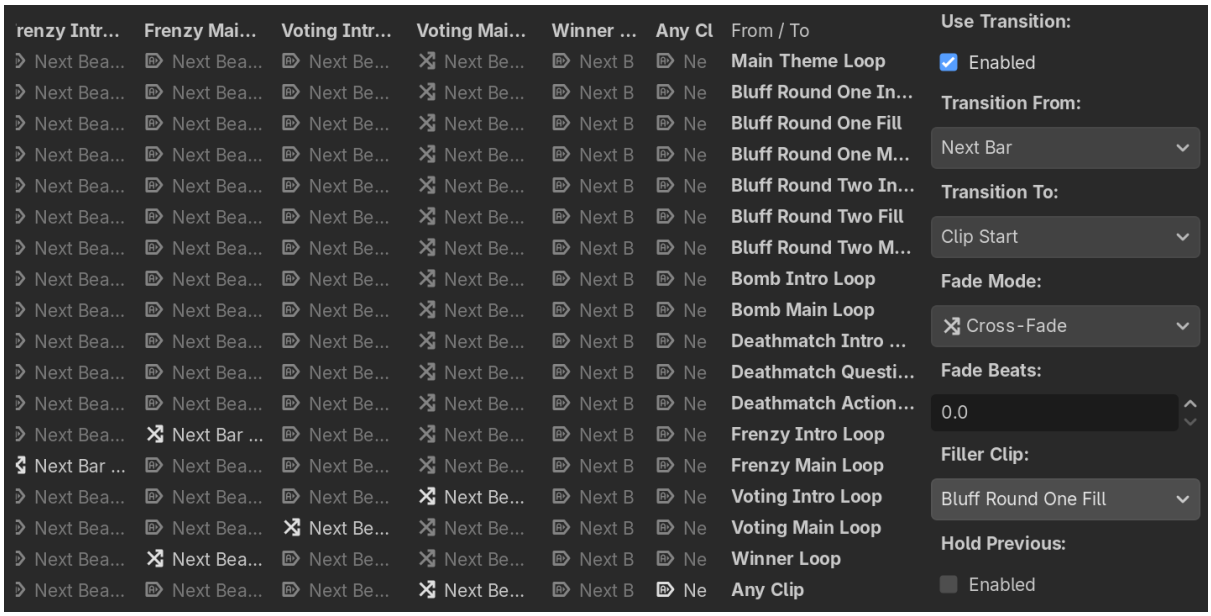

Figure 5.1 – AudioStreamInteractive Transition Editor

Due to a bug in Godot 4.6, the *Fade Mode* must be set to Cross-Fade and *Fade Beats* to 0.0 for the fill clip to transition correctly into the target clip. However, one downside is that Godot's transition system can accumulate a huge matrix of clips. For a big project like this, where the music constantly transitions from one track to another, a better system is needed that can maintain minimal complexity regardless of clip count. To avoid implementing such a system and to maintain simplicity, most of the other minigame tracks don't use filler clips.

5.2.4. Bluff

In Bluff, the AGH is utilised in multiple ways to create the perception of memory, narrational intelligence, and humour. Before the minigame begins, the trivia API endpoint is called to generate a set of questions for the game. However, Bluff has an additional requirement to meet: due to the nature of the gameplay, questions that exclude a single incorrect answer (e.g. questions phrased like '*which of these is NOT…*') should be removed from the question queue. If left in, those questions would likely receive multiple correct answers as bluffs from players. To do this, a custom word filter is used to exclude questions matching certain phrases: 'which' *and* ('not' *or* 'these' *or* 'following'). This logic is prone to false positives, but the number is small enough not to have a noticeable effect on the large question set. If a question matches this phrasing, the queue is discarded, and a new set is fetched. This approach was chosen over creating a separate 'Bluff questions' file, as it maintains simplicity for downloading questions.

One of the hardest issues that had to be solved was the round one generation. If we generate a response only when needed, the players will have to wait a while before any voice line is heard. This has the same effect as loading bars and completely ruins the immersion for a party game. To circumvent this, the time during the intro and rule-explanation dialogue is used to generate the commentary for the first round. By the time the host has explained the rules, the dynamic audio content is ready to play. The rule-explanation dialogue was previously also using the AGH; however, this led to difficulties during the voting game. Since there was no reliable way to tell which minigame would win the vote until the timer had ended, it was infeasible to assume the winning game and pre-emptively generate audio for it. As soon as the generation for round one finishes and begins playing, the AGH automatically starts generating the audio for round two. This guarantees that the Voice Over (VO) for the next question is ready when it is needed. This pre-emptive generation fulfils requirement 1.5, which requests that commentary is ready before the first round begins.

There are two other forms of AGH-generated content in Bluff. When all guesses have been submitted and the timer runs out, a *reveal comment* request is made. The reveal comment is a witty filler that leads into the answer reveal phase. This includes the following context: the players and their current score, and the number of players who guessed correctly in the last round. The context is then used along with the system prompt to create this dynamic filler dialogue. The reveal comment always starts generating during the guessing phase timer, which gives the AGH an allocated 30 seconds to finish. The other of the two generated content comes straight after the reveal comment. As soon as the reveal comment audio starts playing, the answer VO audio begins generating. The user-provided input string for the AGH is longer for this prompt than most others. This is to remove any ambiguity from who fell for the bluff and who wrote it. Without this, I found the LLM struggled to identify the correct subjects and therefore often generated responses that made little sense. If the currently shown answer is a player-made bluff, the following input prompt is used: '*The following players incorrectly guessed a bluff: [players]. The bluff is: [bluff]. This bluff was created by: [player]*'. If the bluff was a game-generated lie instead (which can only appear when there are fewer than 4 players), the same prompt is used but with a different ending: '*This bluff was a game-generated one*'. However, if the currently shown answer is the correct answer, one of two input strings is used: '*No players guessed correctly. The correct answer is: [answer].*' or '*Correct guessers: [players]. Incorrect guessers: [players]. Answer: [answer]. Question: [question]*'.

As mentioned in the design, the last round of bluff works slightly differently. Instead of generating a witty, question-related statement, the LLM first conducts a pre-processing stage. The model is given the question and must identify a broad enough category that the question fits into. During this stage, the LLM doesn't know the question either. This is intentional, as it minimises the risk of the question accidentally being read out instead. The system prompt also includes the rules for the final round and updated examples to guide its responses. After the category is identified, it is passed into the AGH system to generate the audio for the final round. This stage in Bluff is the only time in the entire project that the LLM is used to generate a non-speaking response.

5.2.5. Bomb

Compared to Bluff, Bomb has very little AGH-driven audio content, mainly from the lack of a round structure. Since the number of questions asked before the bomb explodes varies with each round, there is no reliable time frame in which to generate the question-specific commentary. Instead, a queued question approach is used: when the question queue is populated, the AGH generates a witty comment about the twelfth question in the queue. When that question eventually appears on screen, the pre-emptively generated audio plays, and the generation for the next queued commentary begins based on the updated queue. This approach keeps the minigame's pre-emptive behaviour (satisfying requirement 2.3) whilst maintaining a balanced generation load, making sure the AGH remains stress-free and not bottlenecked by large generation queues.

5.2.6. Deathmatch

Deathmatch uses a similar approach to Bluff in that the AGH generates commentary for the next question whilst the current one is being played. Unlike Bomb, Deathmatch's round-based structure allows for a reliably long enough time to generate a response. After the current question is read out and the answer timer begins, players have a fixed period of time to respond, choose items, and watch them resolve in order on screen. This period provides enough time for both the LLM and TTS generations to complete before they are needed, making sure the commentary is ready to play as soon as the question appears. This creates a similar *illusion of response* to the one achieved in Bluff, without needing the more complex AGH scheduling that Bluff's answer-reveal commentary requires.

5.2.7. Frenzy

Frenzy keeps Bomb's pre-emptive generation strategy for its commentary; however, it introduces an additional layer of context, making it different from the other three minigames. Since Frenzy is not votable, it automatically loads after all other minigames have been played. This is a deliberate design decision – by the time Frenzy starts, the AGH has already collected multiple sets of gameplay data from the in-progress session. Throughout the earlier minigames, relevant statistics are internally tracked and stored in memory. At the end of each minigame, these statistics are converted into questions and added to the dynamic question pool for Frenzy. The statistics tracked in each minigame, along with their potential questions, are as follows:

Bluff

- Bluffs fallen for per player: '*In Bluff, who fell for the most bluffs?*'
- Correct answers per player: '*Who guessed the most correct answers in Bluff?*'
- Bluff authors: '*In the final round of Bluff, who wrote the bluff: [bluff]?*'

Bomb

- Longest bomb hold: '*In Bomb, who held the bomb for the longest amount of time?*'
- Duration of longest hold: '*In Bomb, how long did [player] hold the bomb for?*'
- Total bomb passes: '*In Bomb, how many times was the bomb passed to a new player?*'
- Most correct answers holding the bomb: '*In Bomb, who got the most questions right whilst holding the bomb?*'

Deathmatch

- Most uses of an item by one player: '*Who used the most [item]s in Deathmatch?*'
- Total uses of an item: '*In Deathmatch, how many [item]s were used in total?*'

During Frenzy, dynamic questions are randomly inserted at a set interval (every x standard trivia questions), rather than being exclusively used. This ensures the minigame retains the feeling of a typical trivia game whilst occasionally twisting it with personalised, session-specific content. Spacing the questions out also prevents the question pool from running out as quickly, given that it's much smaller compared to the standard question dataset. Since the questions directly reference events the AGH has already commented on during previous minigames, its narration in Frenzy feels particularly reactive and context-aware. This reinforces the sense of memory across the session in the host's behaviour and showcases the most context-driven use of the AGH pipeline in the project. Making Frenzy the final, cumulative effort of all the minigames was a deliberate design choice designed to make players feel like the game provided a personalised conclusion to their experience.

6. Methodology

This section covers the methodology used to evaluate the AGH system, delving into the study design, user survey, and analytical approaches used to collate and assess the data.

6.1. Study Design

Over a few weeks, three studies were carried out, totalling six participants for feedback. Most participants were students; however, two older adults (aged 50-60) also took part. Convenience sampling was used to gather participants due to the time constraints imposed by the project scope. All participants were familiar with the concept of digital party games, so little explanation was needed before they began. In the study, participants first played through all four party games in a small set of groups for observation. Before this, an information sheet was presented along with a consent form. These sessions were audio-recorded and later transcribed in chunks, with any identifying information anonymised. After the game finished, the players were asked to complete a 5–10-minute survey questionnaire about their experience. Multiple ethical considerations were addressed at every stage of the study. Participation was entirely voluntary, and players could stop and leave the study at any time. All audio recordings were anonymised, transcribed, and stored in a secure digital location. All physical copies of consent forms were also securely stored.

6.2. Survey Design

For this project, a mixed-methods survey was used to gather feedback from participants following their interaction with the Artificial Game Host. This approach combines the strengths of qualitative and quantitative methods, providing detailed data that not only describes *what* participants thought of the experience, but also *why* they felt that way. Quantitative data focused on measurable aspects such as game ranking, host believability, humour, and responsiveness, whilst qualitative data was collected through open-ended questions, allowing participants to state reasoning and impressions in more detail. This approach works well because game experience is not only measurable, but also subjective.

6.3. Data Analysis

Quantitative responses about the AGH were collected via a Likert scale and converted into a range of -3 to +3 for sentiment analysis. A Sample Mean (SM) value was also calculated for each question to assess the scale of the sentiment, with occasional standard deviation values used to analyse the consistency of responses. Where necessary, SM values were additionally converted to percentages for normalisation. Qualitative data – collected via open-ended, long-answer questions – was analysed from repeat topics and statements related to the AGH, which were grouped into categories such as believability, relevance, and pacing. Audio recordings were then transcribed in sections and anonymised before analysis. The main limitations of this study were its small sample size and reliance on convenience sampling, both of which are covered in more detail in Section 8.2.

7. Evaluation

In this section, we will present the findings from the three studies carried out and evaluate whether the AGH had a noticeable impact on game engagement. We will first analyse the survey results, covering both qualitative and quantitative feedback, before performing sentiment analysis on AGH-focused statements and discussing audio recordings of the studies.

7.1. Survey Analysis

The survey comprised sixteen questions, ranging from simple game rankings to open-ended long-answer boxes. A seven-point Likert response scale was used, ranging from *Strongly Disagree* (1) to *Strongly Agree* (7), before being converted to a symmetric scale of -3 to +3 for analysis. A Sample Mean (SM) value (shown in Table 7.1) was calculated for each question to determine the participants' overall sentiment. To establish a baseline, participants were first asked how frequently they play digital party games. All six responses indicated they play this type of game only on special occasions, such as parties or holidays. This suggests that the participants were casual players rather than hobbyists, which is the intended target audience for the project. The eight host-based Likert questions (Q6-Q13) have been grouped into four subsections for clarity: *Rule Understanding*, *Believability and Timing*, *Quality and Engagement*, and *Replayability*.

7.1.1. Rule Understanding (Q6)

Question 6 asked whether participants understood the rules of the minigames without requiring any additional assistance. This produced the lowest SM of all the host-related questions at +0.33 (55.5% converted to a percentage scale). Two participants selected *Somewhat Disagree*, and no participant selected any response more negative than that, suggesting confusion was present, but not enough to ruin the experience. Qualitative responses subsequently backed this finding. Feedback pointed specifically to *Bomb* and *Frenzy* as the main difficulties, with one participant noting that they '*hadn't had a chance to read*' the question before their input was registered, and another emphasising that *Frenzy*'s lack of on-device questions forced them to look between two screens.

Importantly, these issues relate to game design and usability rather than the AGH itself. The rule-explanation audio across all four minigames was pre-generated rather than dynamically created by the AGH, meaning these scores showcase the quality of the prewritten text and its hand-picked delivery via the TTS model, rather than the LLM's generative quality. The much higher scores in Questions 8 and 10, which more directly measure the AGH's generative ability, suggest that the content and phrasing of the rule explanations themselves likely caused the rule confusion rather than an issue with the speech synthesis part of the pipeline.

7.1.2. Believability and Timing (Q7, Q11)

Question 7 asked participants whether the host felt like a real participant rather than a recorded voice and produced an SM value of +1.33 (72.2%). Four different options were selected across

responses. One participant chose *Somewhat Disagree*, two selected *Somewhat Agree*, two *Agree*, and one *Strongly Agree*. The lack of strong negative sentiment indicates that the AGH was generally seen as a dynamic host rather than a static audio player. The single negative response, however, could be explained by one qualitative feedback comment. It was stated that '*sometimes [the host] felt like it talked a little slowly*'. This reduced speaking rate is a known quality of the Chatterbox model at lower *exaggeration* values, and the *cfg* parameter adds more deliberate pacing to this. Although it was an intentional design choice to prioritise clarity, it seems to have detracted from the natural, realistic tone participants expected. Question 11 focused on the timing of the host's commentary and how natural it felt to participants. Results produced an identical SM value to that of Q7 (+1.33), although it had the highest calculated variance of any host-related question (1.70). One participant selected *Disagree*, while two others selected *Strongly Agree*. This suggests some disagreement between participants on the host's pacing, and matches the qualitative comment talked about above. Despite this, however, the overall positive SM value indicates that most of the participants found the timing okay.

7.1.3. Quality and Engagement (Q8, Q9, Q10, Q12)

These four questions steered responses toward the core characteristics of the AGH, and each one returned a strong positive result in the survey.

Question 8 asked whether the host's commentary felt relevant to what was happening on the main host display and achieved an SM value of +2.00 (83.3%) with no negative responses and a low standard deviation (0.82). All participants chose *Somewhat Agree* or higher, backing that the context system powering the LLM (outlined in Section 5.2.1) was practical. Effective input structuring, response templates, and examples within system prompts enabled the model to accurately generate context-aware commentary consistently across all four minigames. The qualitative responses reinforced this, as one participant recalled a specific moment of the host imitating 'memory', stating 'the host complimented me getting three correct answers in a row'. Another participant also observed that the host '*was quite sassy about people falling for bluffs and remembered past choices*'. These comments show that the AGH's use of game data, such as player scores, correct answers, and round history, produced commentary that felt adaptive rather than generic and static.

Question 10, which queried the clarity and intelligibility of the host's speech, returned the highest SM value of +2.17 (88.1%) among all questions. All responses were positive, with many choosing the *Agree* or *Strongly Agree* options. This result means that the TTS optimisation choices made, such as the chunking algorithm and punctuation-silence generation (Section 5.1.1), positively enhanced the naturalness and pacing of the synthesised speech. It also shows that the prompt sanitisation steps, which discourage abbreviations and emojis, also benefited the clarity and legibility of the generated LLM output.

Questions 9 and 12 asked about the host's impact on the game's memorability and overall engagement, and both returned SM values of +1.83 (80.5%). Question 9 queried whether the host's personality made the experience more memorable than a standard trivia game, whilst Question 12 asked specifically whether participants found themselves more engaged because of the commentary. Both questions received strong positive responses, suggesting that the AGH had a lasting impact on the quality of the gameplay. The qualitative responses provided

additional direction, with players commenting on the host's '*sarcastic tone when nobody gets the answer correct*' during item usage in *Deathmatch*, and the '*cringe humour*' of certain pre-generated lines being memorable moments. As discussed in the design, the minigames utilise both dynamically generated and pre-generated content. Combining these two methods allows for contextual, LLM-generated commentary alongside carefully written, pre-generated dialogue, producing a clear character and engaging personality.

7.1.4. Replayability (Q13)

Question 13 focused on whether participants would play *Trivia Scramble* again with their friends, which resulted in an SM value of +1.83. However, this had the largest variance of all the questions (2.19). Whilst four players chose *Strongly Agree* and one chose *Agree*, one participant selected *Strongly Disagree*. After analysing the qualitative responses, no participant showed a strong negative attitude towards the game experience itself, suggesting the one negative response on the Likert scale might have focused on the game's setup complexity and practicality rather than the game itself. Unlike commercial party games, *Trivia Scramble* requires a local Python server running both LLM and TTS inference, a Godot host display, and a web server. This creates a huge barrier to entry for casual gamers, therefore making a simple, single-executable launcher a critical task for future work and iterations. Despite this, the responses evidently point towards strong replayability, suggesting that the game was engaging whilst not repetitive.

Question	Area	SM (-3 to +3)	Normalised (%, 1 d.p.)
Q6	Rule Understanding	+0.33	55.5%
Q7	Host Believability	+1.33	72.2%
Q8	Relevance of Commentary	+2.00	83.3%
Q9	Memorability	+1.83	80.5%
Q10	Clarity	+2.17	86.2%
Q11	Humour and Timing	+1.33	72.2%
Q12	Engagement	+1.83	80.5%
Q13	Replayability	+1.83	80.5%

Table 7.1 – Sample Means for Questions

The Sample Mean values in Table 7.1 were calculated via the following equation, with *a-g* denoting the number of participants who voted for that option:

$$SM(x) = \frac{(-3 \times a) + (-2 \times b) + (-1 \times c) + (0 \times d) + (1 \times e) + (2 \times f) + (3 \times g)}{6}$$

To normalise the values to a percentage (%), this final equation was used:

$$Normalise(x) = \frac{\mathrm{SM}(x) + 3}{6} \times 100$$

7.2. Minigame Findings

Participants were asked to rank the four minigames by enjoyment and to identify which one felt the most dynamic when responding to player actions. These questions provided useful insight when evaluating the AGH's overall impact on the minigames themselves rather than the system at large.

In terms of enjoyment, *Bomb* ranked highest with an average ranking score of 3.33 out of 4.00, followed by *Deathmatch* (3.00), *Frenzy* (2.17), and *Bluff* (1.50). Qualitative responses revealed that the higher-ranked minigames were preferred due to their fast pace and chaos, with participants stating: '*more chaotic, more fun*' and '*bit more chaos and time limits*' as reasons. This suggests that in-the-moment tension was more important for enjoyment than the number of AGH-generated content. More importantly, *Bluff* contained the most dynamically generated audio content of any minigame, yet ranked the lowest in enjoyment, implying that systems like the AGH cannot compensate for game design and slower-paced gameplay.

When asking which minigame felt the most dynamic when responding to player actions, *Deathmatch* received the 50% of the votes, with *Bluff* second at 33% and *Bomb* third at 17%. This result is not expected, considering that *Bluff* features the most context-driven LLM commentary out of all the minigames; however, it showcases the fact that *Deathmatch*'s item system produces high-stakes consequences that the host comments on in real-time. Participants noted '*suspense to see who got the dagger when the host responded to people's actions*' and the extent of the item mechanics when responding to the qualitative question. This suggests that the responsiveness of the AGH is not solely tied to the quantity of content it generates, but also how the system utilises game-driven context to achieve relevant commentary. This should be taken into consideration when further iterating the AGH. Commentary that directly names players, references their specific actions, and responds to key moments in gameplay is more likely to invoke a sense of reactivity for players than context-packed narration.

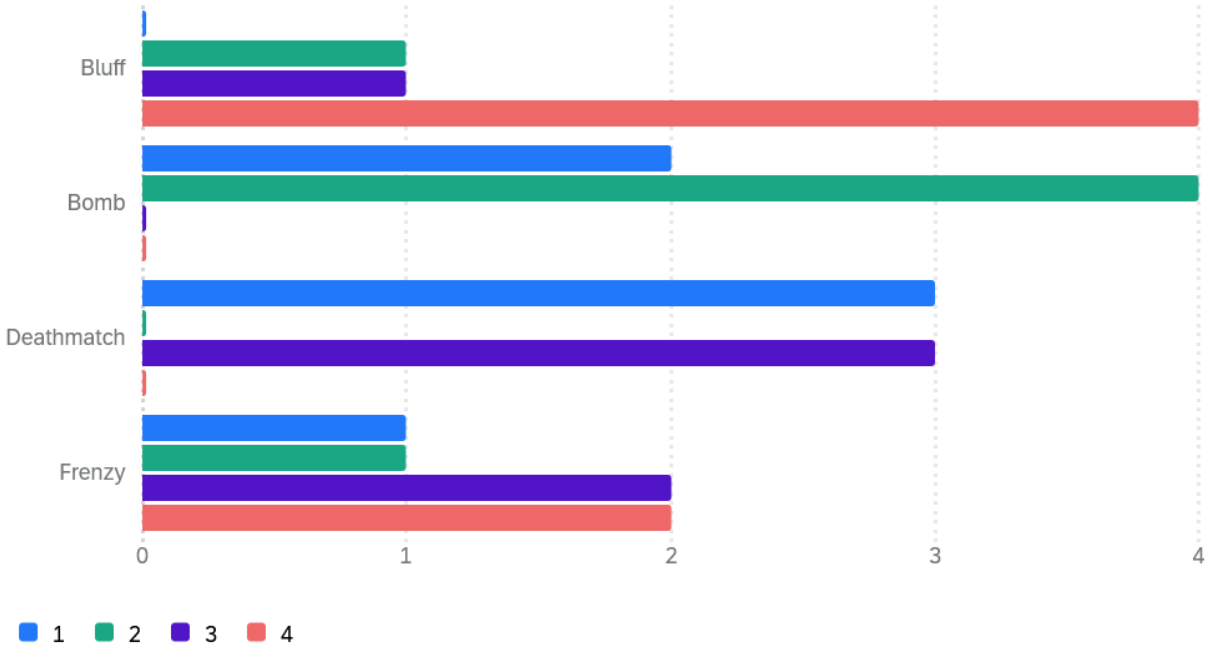

Figure 7.1 – Minigame Rankings (Fun)

Minigame	Overall Rank	Average Ranking Score (1 to 4, 2 d.p.)
Bomb	1	3.33
Deathmatch	2	3.00
Frenzy	3	2.17
Bluff	4	1.50

Table 7.2 – Average Minigame Ranking Score (Fun)

The Average Ranking Score (ARS) values in Table 7.2 were calculated via the following equation, with *a-d* denoting the number of participants who assigned a ranking of 1-4, respectively:

$$ARS(x) = \frac{(4 \times a) + (3 \times b) + (2 \times c) + (1 \times d)}{6}$$

7.3. Observational Analysis

To provide context for the survey feedback, audio recordings from the three sessions were analysed for verbal reactions. These 'in-the-moment' responses backed up the high scores for the host's believability and commentary relevance. Participants frequently showed interest in the AGH, talking directly back to the host as though it were human. For instance, during a username roast, one player laughed and shouted '*That's just my normal username! Shut up!*'

The AGH's ability to remember game information also received praise. When the host commented on a player's answer streak or referenced specific choices in *Bluff*, participants seemed surprised, expressing statements like '*Whoa! That's crazy!*' and '*Oh, that's cool*'. This showcases that the LLM's context-driven dialogue successfully created the illusion of an observant, reactive host.

On the other hand, the recordings also provided important context for the lower scores on rule understanding and *Frenzy*'s enjoyment rating. The transcripts showed that the issue was mainly UI and pacing-related rather than with the AGH. During *Bomb* and *Frenzy*, players often expressed panic during visual modifiers and fast timers, stating, '*It's too fast, I can't read it!*' and '*What is going on?!?*' Contrastingly, the dynamic audio systems were highly praised, with participants stating a liking for both the AGH and adaptive music system ('*I like the dynamic music!*').

Finally, the transcripts also revealed a logic error in one of the minigames, highlighting the limitation of certain generative models. During *Bluff*, the host incorrectly recalled a player's performance, causing the participant to exclaim: '*I didn't get two right, it said I got two right!*' Whilst this only occurred once across all three sessions, it shows that there are still risks related to non-deterministic technologies like LLMs in real-time interactive environments, where accuracy is a non-negotiable necessity.

8. Conclusion

8.1. Requirements Review

In this section, we will evaluate the project requirements from Section 4.5 (Appendix D) against the final implementation details and feedback received from user studies.

8.1.1. AGH Pipeline (Requirement 6)

The core AGH pipeline was fully implemented and successfully met all four sub-requirements. The OpenHermes LLM was loaded into memory via the *llama-cpp-python* library and consistently outputted context-aware responses across all minigames (6.1). The Chatterbox TTS model was loaded and utilised a custom emotion engine (6.2), with a priority queue system to ensure TTS requests were processed before LLM requests (6.3). The pipeline was then integrated into the main host display to output the responses alongside gameplay (6.4). The survey results show that the AGH produced relevant (Q8 SM = +2.00), clear (Q10 SM = +2.17), and high-quality responses in a real-time game environment.

8.1.2. Minigames (Requirements 1-4)

All four minigames were successfully implemented to a playable and robust standard. *Bluff* contained the largest amount of AGH-related content, including pre-emptive round-based commentary and LLM-driven category identification for the final round. *Bomb*, *Deathmatch*, and *Frenzy* all implemented a queue-based question TTS system, completing their requirements. Additionally, requirements 1.2, 2.2, 3.2, and 4.2 – which stated that the host shall explain the rules of each minigame – were successfully met through pre-generated audio. Despite this, the low SM value in Q6 (+0.33) indicates that the phrasing of these rules wasn't as clear as other dialogue, especially in more complex games like *Bomb* and *Deathmatch*. However, this is more of a human-written error rather than a problem with the AGH phrasing itself. *Frenzy* also didn't perform as intended due to the lack of on-device questions, which multiple participants stated was an important usability issue. The dynamic display/scene system, leaderboard, and room management were all fully implemented and received positive feedback. Additionally, the trivia API was implemented as described, with local questions loaded on server startup. All these requirements were fully met, successfully creating an engaging trivia experience for participants.

8.1.3. Overall Analysis

Out of the nine main requirements, six were fully met, two were partially met (rule understanding and *Frenzy*'s mobile display), and none were unmet from a technical viewpoint. Importantly, the two partially met requirements don't suggest issues with the AGH pipeline itself, but rather with the project's game design.

8.2. Limitations and Future Work

Several limitations of this project should be considered when building on these findings. The study involved only six participants and relied on convenience sampling, which limited the scope of the results. Additionally, all participants were already familiar with digital party games, which may have made the game easier to understand and more positively received than it would be for a more generalised audience. The evaluation also lacked a control condition, so the engagement level of each game cannot be conclusively linked with the AGH itself. Some technical and usability issues also affected the experience, with players expressing unclear rules, insufficient time to read longer questions, visibility problems in Frenzy, and pacing concerns with the host.

For future work, a larger-scale study with a broader range of participants would improve the reliability of the results and allow for stronger conclusions about player engagement. Most importantly, the next step would be to compare the AGH directly against a fully pre-recorded host to help determine the actual impact of the LLM and TTS-powered technology. In terms of iterative design, future work should improve the pacing of the host's speech, ensuring it better matches the speed of the game being played. Although an emotion system was developed, it was only used for pre-generated audio. Future systems could develop a more advanced emotion system that utilises both the speech synthesis *and* LLM. Using this for in-game generated audio (not just pre-generated) should create a richer, more varied experience where the host feels even more adaptive to the situation. Finally, newer local models are constantly being developed and released. Testing these newer models as hardware and open-weight technologies continue to improve would be a worthwhile investment for such a project.

8.3. Final Statement

To conclude, this project has demonstrated that local LLMs and speech synthesis can be used to create a believable, robust, and engaging Artificial Game Host within a digital party game. These findings build upon prior work on procedural generation (Hendrikx et al., 2013) and extend the research into LLM-driven adaptability. Whilst this version of the system had some issues with clarity and usage, the results suggest that truly dynamic, AI-driven dialogue can make video games feel more adaptive, memorable, and socially engaging than standard, static assets. As local, open-weight models continue to improve, systems like the AGH have the potential to become a practical and worthwhile tool for game pipelines, rather than a novel, trending technology.

References

APXML *GLM-5 Specifications.* Available at: https://apxml.com/models/glm-5 .

Bommasani, R., Hudson, D.A., Adeli, E., et al (2021) On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258,* .

Epic Games *Lumen Performance Guide.* Available at: https://dev.epicgames.com/documentation/en-us/unreal-engine/lumen-performance-guide-for-unreal-engine .

Fan, Q., Nan, E., Li, W., et al (2025) Werewolf: A Straightforward Game Framework with TTS for Improved User Engagement. *arXiv e-prints,* arXiv: 2506.00160 .

Fish, J.B. (2003) Interactive and adaptive audio for home video game consoles.

Hello Games (2025). *Introducing No Man's Sky: Worlds Part II.* Available at: https://hellogames.org/2025/01/29/introducing-no-mans-sky-worlds-part-ii/ .

Hendrikx, M., Meijer, S., Van Der Velden, J., et al (2013) Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM),* 9(1), 1–22 .

Kimi Team, Bai, T., Bai, Y., et al (2026) Kimi K2. 5: Visual Agentic Intelligence. *arXiv preprint arXiv:2602.02276,* .

Masuch, M. & Röber, N. (2005) *Game graphics beyond realism: Then, now, and tomorrow,* Proceedings of DiGRA 2005 Conference: Changing Views: Worlds in Play.

Mike Bilder, Evan Jacover, Jackbox Games, Peter Heinrich & Amazon (2015). *Quiplash: The Multiscreen, Multiplayer Game for 3 to 10,000 Players.* Available at: https://www.youtube.com/watch?v=MxZdg2NNr1A .

Ortiz, A., Qorbani, H.S. & Elmorsy, M. (2025) Breaking the Cloud Barrier: Local Deployment of Conversational LLMs for Real Time NPC Interaction in Video Games.

Park, J.S., O'Brien, J., Cai, C.J., Morris, M.R., Liang, P. & Bernstein, M.S. (2023) *Generative agents: Interactive simulacra of human behavior,* Proceedings of the 36th annual acm symposium on user interface software and technology.

Resemble AI *Chatterbox: the leading family of open source AI voice models, cloned from five seconds of audio.* Available at: https://www.resemble.ai/learn/models/chatterbox .

Rogers, K. & Weber, M. (2019) *Audio habits and motivations in video game players,* Proceedings of the 14th International Audio Mostly Conference: A Journey in Sound.

Sakshi Dhingra (2025) AI Dungeon: The Future of Interactive AI Storytelling.

Stevens, R. & Raybould, D. (2013) *The game audio tutorial: A practical guide to creating and implementing sound and music for interactive games.* Routledge. .

Sweet, M. (2014) *Writing Interactive Music for Video Games: A Composer's Guide.* Addison-Wesley Professional. .

Tait, E.R. & Nelson, I.L. (2022) Nonscalability and generating digital outer space natures in No Man's Sky. *Environment and Planning E: Nature and Space,* 5(2), 694–718 .

Tan, X., Qin, T., Soong, F., et al (2021) A survey on neural speech synthesis. *arXiv preprint arXiv:2106.15561,* .

Teknium (2023). *OpenHermes Dataset.* Available at: https://huggingface.co/datasets/teknium/openhermes .

TheBloke (2023). *Openhermes 2.5 Mistral 7B - GGUF.* Available at: https://huggingface.co/TheBloke/OpenHermes-2.5-Mistral-7B-GGUF .

Valve (2026). *Steam Hardware & Software Survey: January 2026.* Available at: https://store.steampowered.com/hwsurvey/videocard/ .

Wikipedia (2024). *Streaming audio in video games.* Available at: https://en.wikipedia.org/wiki/Streaming_audio_in_video_games .

Appendix A – Project Proposal

The original project proposal for this dissertation is included in the zipped folder as an external document. It covers the initial project goals, motivation, proposed methodology, and planned development schedule.

Appendix B – Supporting Material

Supporting material for this dissertation is included in the zipped folder as source files. It contains the code for the Python server, web client, Godot host, and any other assets used during development. Note: The LLM/TTS models aren't included; these can be accessed via:

OpenHermes: https://huggingface.co/TheBloke/OpenHermes-2.5-Mistral-7B-GGUF/tree/main

Chatterbox (Fork): https://github.com/rsxdalv/chatterbox

Appendix C – Tables

Note that, due to benchmark constraints, Table C.1 includes reasoning tokens in the TTFT measurement, meaning reasoning models that appear quick-to-respond may take a long time to think before the final output is received.

Model	Quantisation	TTFT	Tokens/sec	VRAM Usage
GPT-OSS 20B	MXFP4	0.93s	31.71	5.40 GB
Gemma 3 1B	Q4_0	0.22s	209.79	1.40 GB
Qwen3 1.7B	Q6_K	0.11s	183.77	2.20 GB
Qwen3 4B	Q4_K_M	0.24s	104.11	3.30 GB
Qwen3 4B Instruct 2507	Q4_K_M	0.17s	97.41	3.30 GB
Gemma 3n E4B	Q4_K_M	0.27s	80.24	3.60 GB
LFM2 1.2B	Q8_0	0.07s	266.92	1.50 GB
Llama 3.2 3B Instruct	Q4_K_M	0.16s	158.00	2.80 GB
Llama 3 8B Instruct	Q4_K_M	0.32s	87.94	5.10 GB
Phi 3 Mini 4K Instruct	Q4	0.05s	155.86	3.90 GB
Phi 4 Mini Reasoning	Q4_K_M	0.32s	134.71	3.40 GB
Gemma 3 12B	Q4_K_M	1.15s	16.83	5.10 GB
Qwen3 14B	Q4_K_M	0.74s	13.46	5.30 GB
Mistral 7B Instruct v0.3	Q4_K_M	0.39s	96.84	4.80 GB
OpenHermes 2.5 Mistral 7B	Q4_K_S	0.31s	100.78	4.50 GB
Psyonic Cetacean 20B	Q4_K_S	3.85s	5.38	6.10 GB
Phi 4	Q4_K_M	0.72s	14.23	5.60 GB

Table C.1 – LLM Performance Results

Method	Route	Parameters	Return Values	Purpose
GET	/health	None	Success: *{“status”: “ok”}*	Check if the server is up
POST	/rooms	None	Success: *{“room”: Room}*	Create a room and get its data
POST	/rooms/{room_code}/join	Route: *room_code* Body: *username*	Success: *{"player_id": string, "room_id": string}* Error: *{"error": string}*	Add a player to a room via code
POST	/rooms/{room_id}/kick	Route: *room_id* Body: *player_id*	Success: N/A Error: *{“error”: string}*	Kick a player from a room
GET	/rooms/{room_id}/player/{player_id}/validate	Route: *room_id*, *player_id*	Success: *{"valid": true, "player": Player, "room": Room}* Error: *{"valid": false, "error": string}*	Check if a player can be reconnected
GET	/rooms/{room_id}/questions	Route: *room_id*, *token*, *count*, *type*, *category*, *difficulty*	Success: *{"questions": Question[]}* Error: *{"error": string}*	Get random trivia questions
GET	/trivia/categories	Body: *token*	Success: *{"categories": string[]}* Error: *{"error": string}*	Get all trivia categories
POST	/generate/text	Body: *token*, *prompt*, *input*	Success: *{"request_id": string, "status": "queued"}* Error: *{"error": string}*	Queue an LLM text request
POST	/generate/audio	Body: *token*, *text*, *mood*, *save*	Success: *{"request_id": string, "status": "queued"}* Error: *{"error": string}*	Queue a TTS audio request

Table C.2 – HTTP Server Endpoints

Route	Parameters	Return Behaviour	Purpose
/ws/display/{room_id}	Route: *room_id*	When connected, it sends an *authenticated* event with an access token. It receives host events – such as minigame flow, voting, etc. – and sends errors for invalid actions.	Check if the server is up
/ws/player/{room_id}/{player_id}	Route: *room_id*, *player_id*	When connected, it sends a *room_updated* event with a token. It receives player events – such as ready, votes, answers, etc. – and sends errors for invalid actions.	Create a room and get its data

Table C.3 – WebSocket Server Endpoints

Appendix D – Design Requirements

The following requirements were created before development to specify the scope and functionality of the systems. References to these are made throughout the implementation section and are later used for evaluation in Section 8.1.

1. **Develop a trivia party 'minigame' called *Bluff***
   - 1.1. The game shall fetch trivia questions from the server API and filter out questions that don't work with the game's mechanics.
   - 1.2. The host shall read out the rules of the game and count down before the timer begins.
   - 1.3. A timer shall count down during the rounds and be displayed on the host screen.
   - 1.4. The music should dynamically adapt based on the host audio and round timer.
   - 1.5. The AGH shall generate audio in the background before it is needed.
   - 1.6. Players shall be able to enter a bluff on their devices and see it appear on screen. They shall also be able to wager 50-500 points and choose a player-entered bluff.
   - 1.7. The game should create its own bluffs when there are fewer than four players.
   - 1.8. The host shall commentate on each bluff players fell for – including the answer – whilst showing the corresponding information on the display.
   - 1.9. The game shall dynamically generate a category for the final question.

2. **Develop a trivia party 'minigame' called *Bomb***
   - 2.1. The game shall fetch questions from the server when the question queue is empty.
   - 2.2. The host shall read out the rules of the game.
   - 2.3. The AGH shall generate occasional question commentary before it is needed.
   - 2.4. Players shall be able to answer questions on their devices.
   - 2.5. The correct answer should be visible to players after submissions are received.
   - 2.6. The website UI shall clearly show whether a player has the bomb or not.
   - 2.7. A timer shall count down on the display whilst the bomb fuse is depleting.
   - 2.8. The timer shall be hidden for the final round, and the fuse length is randomised.

3. **Develop a trivia party 'minigame' called *Deathmatch***
   - 3.1. The game shall fetch questions from the server when the question queue is empty.
   - 3.2. The host shall *vaguely* read out the rules of the game before explaining them in full.
   - 3.3. The AGH shall generate audio in the background before it is needed.
   - 3.4. Players shall be able to answer questions on their devices.

3.5. The correct answer should be visible to players after the timer expires.

3.6. Players shall be able to choose between two random items or points if correct.

3.7. Players shall be able to play items from their ‘inventory’ during the ‘action’ phase.

3.8. Players shall not be able to earn points if they have run out of lives.

4. Develop a trivia party ‘minigame’ called *Frenzy*

4.1. The game shall fetch questions from the server before starting.

4.2. The host shall read out the rules of the game.

4.3. A timer shall count down during the questions and be displayed on the host screen.

4.4. A ‘modifier’ shall be added to the game each round to increase difficulty.

4.5. Questions created about the games mentioned above shall be randomly thrown in.

4.6. Players shall answer the questions on their devices.

4.7. The correct answer should be visible to players after the timer expires.

5. Develop a voting round tapping game and an end-game screen

5.1. The host shall read out the rules of the voting game.

5.2. Players shall tap as fast as they can on a minigame to vote for it.

5.3. Votes shall be tallied, and the winning game is decided via a weighted ticket system.

5.4. Previously played minigames shall not be visible on the display or player devices.

5.5. After all the minigames have been played, the AGH shall announce the winner.

6. Develop an AGH pipeline that takes in a prompt and outputs an audio stream

6.1. The LLM shall be loaded into memory and be able to generate responses based on a chat template and input.

6.2. The TTS model shall be loaded into memory and be able to generate an audio stream with emotion control from an input.

6.3. LLM and TTS generations shall be queued and processed, preferring LLM requests over TTS generations to account for network latency between requests.

6.4. An AI host pipeline shall handle both the LLM and TTS server requests, keeping track of the responses and playing back the desired audio.

7. Integrate a dynamic display screen system and leaderboard

7.1. Game scenes shall be able to be added/removed, animating smoothly in and out.

7.2. A leaderboard shall keep track of player scores and be persistent on screen.

8. Implement a room system with minigame tracking and player management

8.1. The display shall be able to create a room and display a lobby of players.

8.2. Players shall be able to join/leave a room via a website and be ready/unready.

8.3. The display should be able to kick unwanted players by clicking on them.

8.4. The player who is the ‘leader’ shall be able to start the game.

9. Create a trivia API on the server that can be accessed via GET requests

9.1. An online trivia database shall be scraped and persistently stored locally.

9.2. The local data shall be loaded upon initialisation and searched when requested.