



MUHAMMAD RAAD SHAUKAT
01-136212-031

MUHAMMAD ASIF
01-136212-024

Early Retirement: An Open World Game

Bachelor of Science in Computer Science

Supervisor: Qazi Haseeb Yousaf

Department of Computer Science
Bahria University, Islamabad

June 2025



Bahria University
Islamabad Campus
(Department of Computer Science)

CERTIFICATE

We accept the work contained in the report titled “Early Retirement: An Open World Game” as a confirmation to the required standard for the partial fulfillment the degree of BS(AI).

Supervisor

Name: Mr. Qazi Haseeb Yousaf

Date: _____

Internal Examiner

Name: Qazi Mohsin

Date: 20/5/2025

External Examiner

Name: Osama Iqbal

Date: 22/5/25

Project Coordinator

Name: Dr. Sohail Akhtar

Date: _____

Head of Department

Name: Dr. Khurram Ehsan

Date: _____

Abstract

Early Retirement: An Open-World Game with Live NPC Interaction presents an open-world PC game developed in Unity, featuring real-time, voice-activated NPC interactions powered by a fine-tuned compact language model (SmolLM2-360M), which explores themes of trauma and self-discovery, aiming to enhance immersion by replacing static dialogue trees with dynamic, AI-driven conversations; this project specifically investigates whether these real-time, voice-driven AI interactions can significantly enhance narrative immersion and player agency in such open-world games. The system utilizes speech processing APIs alongside a custom backend to enable spontaneous, emotionally aware NPC responses to live player speech, a feat achieved through efficient fine-tuning of the language model using a substantial domain-specific dataset of emotional dialogue scenarios, ensuring functionality even under resource constraints. With an average response latency of approximately 3 seconds, the system demonstrates practical viability despite occasional incoherent outputs, and preliminary user testing indicated increased engagement and perceived realism, although further comprehensive evaluation is needed to fully assess long-term usability and sustained coherence. This work therefore contributes a novel framework for integrating lightweight language models into real-time game environments, enabling responsive, emotionally aware NPCs without requiring high-end hardware, thereby showcasing the transformative potential of AI in interactive media and providing valuable insights into the unique challenges and innovative solutions encountered during its development.

Acknowledgment

We express our sincere gratitude to our esteemed supervisor, Sir Qazi Haseeb Yousaf, for his invaluable guidance, unwavering support, and belief in our abilities throughout our final year project. His expertise has been instrumental in shaping our understanding of the complex concepts and methodologies involved in this research.

We are deeply indebted to him for his insightful feedback, constructive criticism, and constant encouragement, which have significantly contributed to the progress of our project. His dedication to our academic growth and his willingness to share his knowledge and experience have been truly inspiring.

We feel privileged to have had the opportunity to work under his mentorship.

*"My Lord, expand for me my chest (grant me self-confidence, contentment, and boldness)
And ease for me my task.
And untie the knot from my tongue.
That they may understand my speech."*

Quran, Surah Ta-Ha, Verses 25-28

Contents

Abstract

1	Introduction	1
1.1	Project Background	1
1.2	Game Description	1
1.3	Objective	2
1.4	Project Scope	2
1.5	Tools and Technology	4
1.6	Feasibility Study	4
1.7	Project Plan and Timeline	5
1.8	Research Questions and Answers	7
2	Literature Review	8
2.1	Large Language Models and Games a Survey and Roadmap	8
2.2	Evaluating LLMs with Interactive Multi-Agent Negotiation Games	8
2.3	Ghost in the Minecraft: Generally Capable Agents for Open-World Environments via LLM	9
2.4	PokéLLMon: A Human Agent for Pokémon Battles with LLM	9
2.5	Existing Systems and Related Work	10
3	Requirement Specifications	12
3.1	Proposed Methodology	12
3.2	Key Features	13
3.2.1	Real-Time NPC Interaction	13
3.3	Functional Requirements	13
3.3.1	AI Interaction Module	13
3.3.2	Game Module	14
3.4	Non-Functional Requirements	14
3.5	Software Requirements	15
3.5.1	For Development	15
3.5.2	For End-Users (Players)	15
3.6	Hardware Requirements	16
3.7	Use Cases	16
4	Design	17
4.1	System Architecture	17
4.1.1	Data Flow Summary	18
4.1.2	Rationale for Architectural Choices	18

4.2	Design Constraints	18
4.3	Design Methodology	20
4.3.1	General Approach	20
4.3.2	Detailed Design Methodology	20
4.4	Distribution of Responsibilities	21
4.4.1	Project Direction and Joint Foundations	21
4.4.2	Game Development	21
4.4.3	AI System Development and Creative Oversight	21
4.5	System Architecture	22
4.6	Sequence Diagram	23
4.6.1	Input Processing Module Design	23
4.6.2	Language Model Module Design	23
4.6.3	Communication Module Design	23
4.6.4	Response Delivery Module Design	24
4.6.5	Voice Controller Module Design	24
4.6.6	Backend Server Module Design	24
4.7	Game World and Inhabitants	24
4.7.1	Game Setting: Vata and Fehari	24
4.8	GUI Design	25
5	System Implementation	42
5.1	System Architecture Implementation	42
5.2	Architecture and Component Integration	42
5.3	Implementation Details	43
5.3.1	Unity Client Implementation	43
5.3.2	Integration Challenges Encountered	43
5.4	Tools and Technologies Implementation	44
5.5	Development Environment	44
5.6	AI Model Implementation	45
5.7	Voice Interaction Walkthrough	45
6	System Testing and Evaluation	47
6.1	Reason For Testing	47
6.1.1	Verification	47
6.1.2	Validation	48
6.2	Unit Testing	48
6.3	System Testing	48
6.4	Usability Testing	49
6.5	Software Performance Testing	49
6.6	Compatibility Testing	50
6.7	Exception Handling Testing	50
7	Conclusions and Future Work	51
7.1	Conclusion	51
7.2	Future Work	51
7.2.1	Latency Reduction	51
7.2.2	Dialogue Quality and Model Enhancements	52
7.2.3	Narrative and Gameplay Expansion	52
7.2.4	Interaction System Improvements	52

7.2.5	Robustness and Evaluation	52
	References	53

List of Figures

1.1	Timeline: Muhammad Asif's Tasks	5
1.2	Timeline: Muhammad Raad's Tasks	6
1.3	Timeline: Joint Project Tasks	6
3.1	AI Integration Flowchart	13
4.1	Agile-Incremental Development Methodology	21
4.2	System Architecture Diagram	22
4.3	NPC Voice Interaction Sequence Diagram	23
4.4	Initial Start Menu Interface	25
4.5	In-Game Main Menu Options	26
4.6	Bird eye view of Fehari & Vata	26
4.7	Map Overview of Game World Zones (Vata & Fehari)	27
4.8	Tyler Wayne (Protagonist)	28
4.9	Netan (Antagonist)	29
4.10	Anne & Qash (Tyler's Wife & son)	30
4.11	Figure 4.11: Baez (Tyler's Best Friend)	31
4.12	Samuel (Gun Supplier/Mechanic) & Roman(Drug Dealer)	31
4.13	Dalas(Junkie) & Cobbler(Robber)	32
4.14	Tob & Todd (Twins)	32
4.15	Example Generic NPCs in the Game World	33
4.16	First mission with & without torch View	34
4.17	Second Mission View	34
4.18	Tyler's Safehouse	35
4.19	Baez's House Ext. & Int.	35
4.20	AI Enhanced aerial View of Fehari & Vata	36
4.21	Player-Accessible Vehicles	37
4.22	Car Parking Ext. & Int.	38
4.23	Police Station Ext, Int. & Jail	39
4.24	Hospital & park	40
4.25	Bank Ext. & Int.	40
4.26	Netan's building for final mission Ext. & Int.	41

List of Tables

3.1	Hardware Requirements for “Early Retirement” Game Client	16
-----	--	----

Acronym/Abbreviation	Full Form
AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
Ext	Exterior
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
Int	Interior
JSON	JavaScript Object Notation
LLM	Large Language Model
LTS	Long-Term Support
NPC	Non-Player Character
ONNX	Open Neural Network Exchange
OOM	Out Of Memory
OS	Operating System
PC	Personal Computer
PCM	Pulse-Code Modulation
PEFT	Parameter-Efficient Fine-Tuning
QLoRA	Quantized Low-Rank Adaptation
RAM	Random Access Memory
RAG	Retrieval-Augmented Generation
RLHF	Reinforcement Learning from Human Feedback
SDK	Software Development Kit
SLM	Small Language Model
SmolLM	Small Language Model (referring to SmolLM2-360M)
SSD	Solid State Drive
STT	Speech-to-Text
SUS	System Usability Scale
TTS	Text-to-Speech

Chapter 1

Introduction

1.1 Project Background

The gaming industry is evolving rapidly, with increasing demands for immersive experiences that blur the lines between reality and virtual environments [1]. Traditional non-player character (NPC) interactions in games rely on pre-scripted dialogues, which often fail to provide the depth and realism players desire [2]. To address this gap, advanced artificial intelligence (AI) technologies such as advanced AI language models offer a promising solution for creating dynamic, context-aware NPC interactions [3].

"Early Retirement" leverages these advancements to deliver an open-world game that combines a compelling narrative of revenge and self-discovery with real-time, voice-activated NPC responses to enhance player engagement.

1.2 Game Description

The core challenge addressed by the "Early Retirement" project is the inherent limitation of conventional Non-Player Character (NPC) dialogue systems in contemporary open-world games. Even in highly acclaimed narrative-driven titles such as Red Dead Redemption 2, interactions rely on extensive but ultimately pre-scripted dialogue trees. This traditional approach, while capable of delivering high-quality authored content, fundamentally lacks spontaneity, true contextual awareness, and the capacity for emergent conversation, often leading to repetitive encounters and restricting player's expression to predefined choices [2]. This mechanical feel can disrupt player immersion and prevent a deeper connection with the game's world and characters [3].

To address this issue, "Early Retirement" integrates advanced artificial intelligence technologies to create real-time, contextually aware NPC interactions. By utilizing AI-driven dialogue in conjunction with speech recognition systems, the project enables dynamic conversations where NPCs can respond intuitively to player voice inputs. This approach aims not only to enhance the potential for emotional complexity within gameplay interactions but also to foster a deeper

connection between players and the game’s narrative, thereby redefining the boundaries of player immersion and engagement.

1.3 Objective

The principal objectives guiding the development and research of the *Early Retirement* project are delineated as follows:

- i. To construct a functional prototype of an open-world game environment using the Unity engine for the Windows PC platform, establishing the necessary setting (the cities of Vata and Fehari) and basic player mechanics (movement, limited vehicle interaction, save/load) to serve as a testbed for the AI interaction system.
- ii. To design, develop, and integrate the complete technological pipeline necessary for enabling real-time, voice-driven NPC interactions within the game environment.
- iii. To enable the system’s NPCs to generate dialogue responses that are dynamically created based on the immediate player voice input, the assigned role and emotion of the NPC, and potentially minimal conversational context, aiming for responses that are coherent and relevant within the interaction turn.
- iv. To quantitatively and qualitatively assess the performance characteristics of the implemented system, focusing primarily on the end-to-end latency of the interaction cycle (targeting ~ 3 seconds). Additionally, to identify and analyze the limitations, particularly those related to the dialogue quality generated by the compact SmoLM2 model and the challenges imposed by network dependency and API response times.
- v. To provide a practical demonstration of integrating advanced, real-time AI voice processing and language model inference technologies within a standard game development workflow of Unity under typical academic project constraints.

1.4 Project Scope

The scope defines the functional and technical boundaries of the *Early Retirement* project, outlining what features and capabilities are included within the developed prototype and what aspects are explicitly excluded. This delineation was crucial for managing development efforts within the project’s timeframe and resource constraints.

Inclusions

1. Core Game Environment: The development includes a functional open-world game prototype built using the Unity engine, specifically designed for the Windows PC platform (Windows 10/11 64-bit). This environment features two distinct city zones, Vata and Fehari, constructed using modular assets from Sketchfab and Unity Asset Store.

2. AI Voice Interaction System: The central innovative feature, enabling real-time, voice-activated conversations between the player and designated NPCs, is fully included. This encompasses the entire pipeline: voice capture, Deepgram STT processing, WebSocket communication, AI inference using the fine-tuned SmolLM2-360M model on the locally hosted server, Deepgram TTS synthesis via REST, and audio playback in Unity.

3. Narrative Context: Implementation of some core elements related to the central storyline focusing on themes of trauma and self-discovery, providing context for player motivation and some NPC interactions.

4. Basic Player Mechanics: Standard player character controls including walking, running, and jumping. Functionality for entering, exiting, and driving a limited selection of car models for world traversal. Combat includes the ability to shoot NPCs or hit them with vehicles. A standard save and load game state mechanism is included.

5. System Components: Development and integration of all essential software modules required for the system's operation, including client-side Unity C# scripts for voice input, API communication, and orchestration, as well as a backend Python server responsible for AI model hosting and inference.

6. NPC & World Elements: Definition and placement of main storyline characters and generic ambient NPCs, each assigned specific roles to support AI context. The game world is structured through the inclusion of some modular building assets and key landmark locations such as the hospital, police station, and bank.

Exclusions

1. Multiplayer Functionality: The project is designed and implemented strictly as a single-player experience.

2. Multilingual Support: Voice input, AI processing, and voice output are limited to the English language only.

3. Cross-Platform Support: Compatibility is restricted to Windows 10/11 PC. No support for macOS, Linux, web browsers, or game consoles is included.

4. Extended Gameplay Systems: Complex gameplay loops beyond the core narrative and interaction mechanics are excluded. This includes elaborate side quests, branching narrative consequences driven by AI dialogue, player abilities like swimming, non-car vehicles such as bikes and boats etc., and extensive character animation variety beyond basic locomotion and interaction states.

5. Advanced AI/Dialogue Features: Mechanisms such as dialogue caching to reduce latency and minimize repeated API calls, persistent conversation memory enabling NPCs to recall past interactions across sessions, database integration for dialogue logging and complex AI state

management, and advanced NPC behaviors driven by dialogue outcomes are not included within the scope of this implementation.

6. Quest Integration with AI Voice: The real-time voice interaction feature specifically excludes direct NPC involvement in quest-giving or providing explicit help/solutions for quests through the dynamic dialogue system.

1.5 Tools and Technology

- **Unity engine** with **C#** – game client development (gameplay logic, physics, rendering).
- **Python** backend – real-time AI communication via **asyncio** and **websockets**.
- **SmolLM2-360M** – fine-tuned compact language model for AI.
- **PyTorch** and **Hugging Face** – model training and inference.
- **Deepgram API** – Speech-to-Text (STT) and Text-to-Speech (TTS).
- **Microsoft Visual Studio** – Unity development.
- **Visual Studio Code** – Python development.
- **Google Colab** and **Kaggle Notebooks** – cloud GPU resources.

1.6 Feasibility Study

A preliminary feasibility study concluded that the *Early Retirement* project, while ambitious, was viable within its defined scope and constraints. Technically, the core technologies required for game development, AI model implementation including a compact language model like SmolLM2 and efficient fine-tuning techniques like QLoRA, voice processing via cloud APIs such as Deepgram, and standard communication protocols were assessed as accessible and mature [4]. The necessary client-side audio preprocessing in C# was also deemed achievable within Unity.

From a resource perspective, standard development hardware was considered sufficient, with API usage costs managed within development tiers. A notable resource investment involved the creation of a large synthetic dataset for AI model training, which, though time-intensive for curation, proved feasible. Operationally, the two-person team structure with complementary skills in game development and AI engineering, supported by an Incremental and Agile methodology, provided a suitable framework for managing the project's complexity.

However, the study also highlighted key risks that shaped the project's scope and design. The performance of the compact AI model carried the inherent risk of occasional incoherence or failures in deep context understanding. Achieving consistent low latency, targeting a three-second interaction cycle, for the cloud-dependent pipeline was identified as a major challenge, susceptible to network variability and API performance. The complexity of integrating disparate technologies

such as C#, Python, and cloud services presented integration risks. Furthermore, dependencies on external APIs and inherent limitations in scaling the single-player focused architecture due to server resource constraints were acknowledged as significant factors. Despite these risks, the overall assessment confirmed the project’s feasibility, necessitating careful management and realistic expectations regarding performance and AI capabilities.

1.7 Project Plan and Timeline

The creation of "Early Retirement" was a structured endeavor spanning from June 2024 to May 2025. During this period, development efforts were strategically organized into distinct phases, encompassing initial research and planning, parallel development of core game and AI systems, critical integration points, and final stages of testing and refinement. Time was allocated to reflect both individual specializations as with Muhammad Asif’s contributions centering on game development within the Unity engine (see Figure 1.1) and Muhammad Raad’s focus on the AI system’s end-to-end creation (see Figure 1.2)and collaborative project milestones. These joint project tasks involved overarching planning, systems integration, quality assurance, and documentation (see Figure 1.3). The Gantt charts presented below offer a visual representation of how this time was distributed, detailing the specific schedules and progression of these key activities and illustrating the allocation of effort towards achieving the project’s objectives.

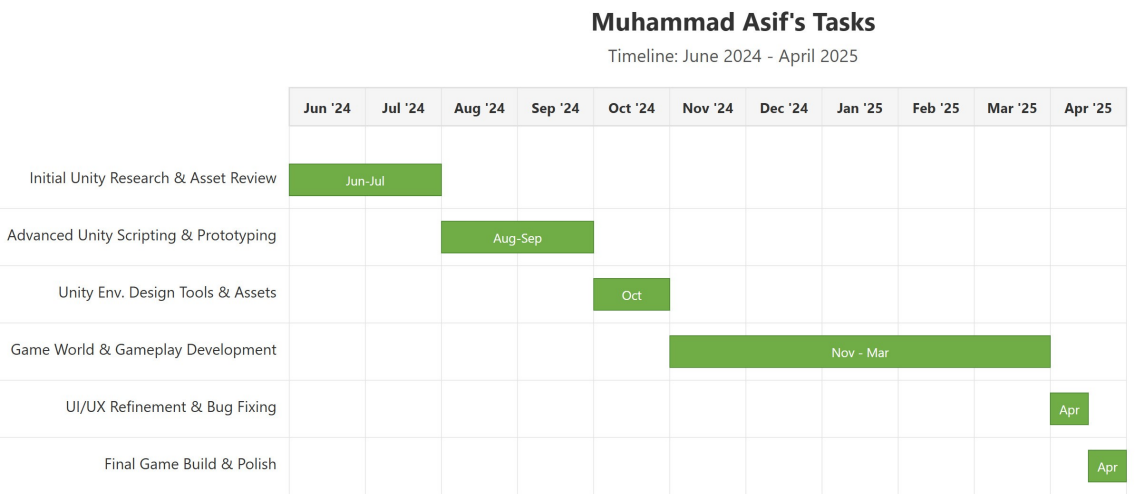


Figure 1.1: Timeline: Muhammad Asif’s Tasks

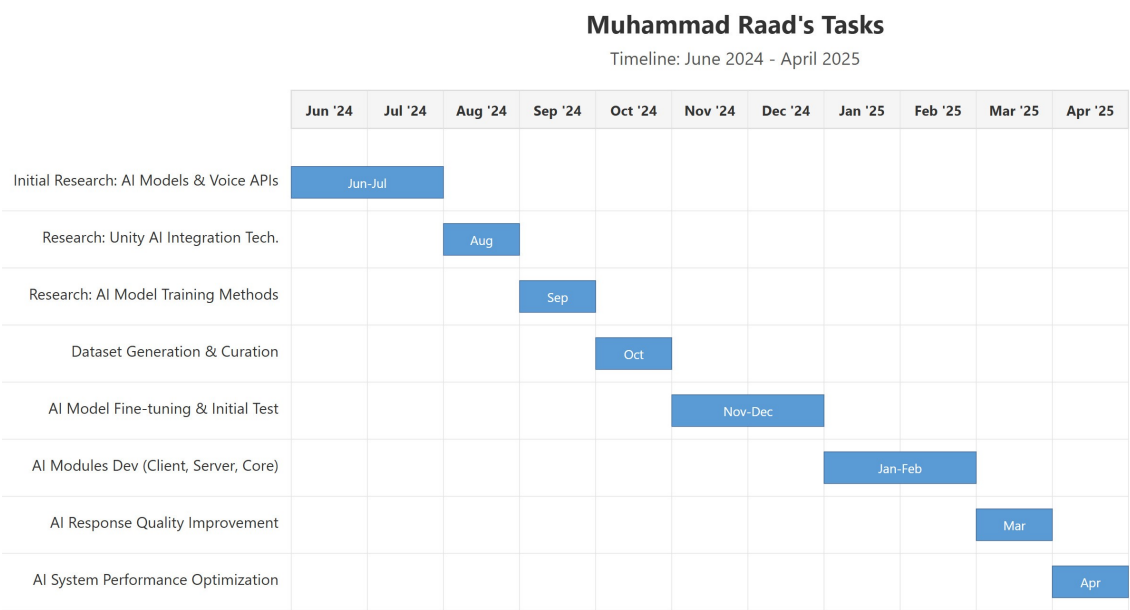


Figure 1.2: Timeline: Muhammad Raad’s Tasks

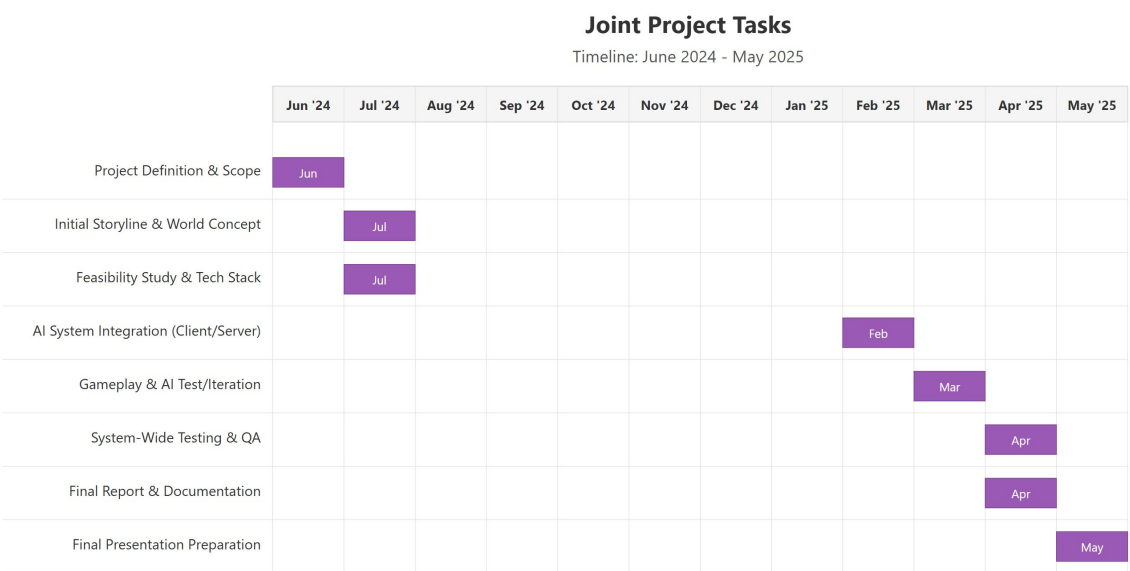


Figure 1.3: Timeline: Joint Project Tasks

1.8 Research Questions and Answers

This project investigated several key questions regarding the implementation of voice-driven AI NPCs in games:

RQ1: To what extent can a compact language model SmolLM2-360M, fine-tuned with QLoRA, generate coherent and contextually relevant real-time dialogue for NPCs? *Answer:*

The fine-tuned SmolLM2-360M demonstrated feasibility, often generating coherent and contextually relevant responses for immediate conversational turns based on NPC role, emotion, and player input. However, its compact nature limited its ability to grasp deeper conversational context or nuance, occasionally resulting in generic or slightly inaccurate responses and infrequent instances of incoherence such as gibberish, highlighting a trade-off between resource efficiency and dialogue sophistication [5].

RQ2: What is the achievable end-to-end latency for a voice-driven NPC interaction pipeline involving cloud-based STT, a local-hosted custom AI model, and cloud-based TTS integrated into a Unity client? *Answer:* The system achieved the target latency of approximately 3 seconds under optimal network and server conditions. However, this latency was highly variable, significantly impacted by fluctuations in internet connectivity, Deepgram API responsiveness, and AI server inference time. While functional, this 3 second delay was perceptible and remains a challenge for achieving truly fluid, natural conversation pacing.

RQ3: What are the primary technical challenges in integrating real-time voice AI services across different platforms and languages such as Unity with C#, Python server, and external cloud APIs? *Answer:* Key integration challenges included selecting appropriate real-time communication protocols requiring WebSockets over HTTP for client-server interaction. Since most external tools are not tailored for Unity, it was a challenge to make everything work coherently. Managing data serialization across languages from C# to Python via JSON, implementing custom wrappers within Unity for reliable external API interaction with Deepgram STT and TTS due to potential SDK compatibility issues, handling client-side audio preprocessing from F32 to PCM16 conversion, and managing the deployment complexity of the containerized AI backend using Docker were all significant challenges.

RQ4: What are the practicalities and impact of large-scale synthetic dataset creation for fine-tuning an NPC dialogue model using existing language models? *Answer:* Creating the large 20,182 interactions synthetic dataset using generator models such as ChatGPT, Gemini, and others was practical for achieving data volume quickly but proved labor-intensive due to the need for extensive prompt engineering and manual curation and cleaning. This dataset was crucial for successfully adapting SmolLM2 to the specific domain and prompt structure. This technique has proven to improve the results of smaller models and is advised [6]. However, reliance on purely synthetic data limits conversational nuance compared to human-authored or real interaction data and risks inheriting biases from the source generator models [6].

Chapter 2

Literature Review

The integration of AI models in gaming represents a pivotal shift toward more interactive and engaging virtual environments [2][3][7]. AI models enable the development of advanced NPCs (non-player characters) and dynamic game agents, redefining traditional game mechanics. This review highlights the achievements and challenges associated with AI model-based implementations, drawing from recent research and examining their implications.

2.1 Large Language Models and Games a Survey and Roadmap

The CICERO model, as explored by Socher et al. (2023), underscores the role of LLMs in creating socially intelligent agents capable of negotiation and strategy formulation in Diplomacy [8]. The model's ability to process player interactions and craft contextually relevant responses showcases the depth LLMs can bring to social simulation. However, the study reveals limitations, particularly in real-time scalability, as generating nuanced responses requires substantial computational power. These challenges echo broader concerns in the field about balancing model complexity with performance, especially in multiplayer settings where speed and consistency are crucial [8].

2.2 Evaluating LLMs with Interactive Multi-Agent Negotiation Games

Similarly, the research by Dodge et al. (2023) on multi-agent negotiation games highlights the transformative capabilities of GPT-4 in fostering strategic reasoning [9][10]. While the structured interactions provided robust results, the reliance on text-based inputs may limit immersion in highly visual or action-driven games. This observation prompts discussions on how LLMs can be seamlessly integrated with multimodal systems, combining visual, auditory, and textual data to enhance user experience. Multimodal LLMs could bridge the gap between textual fluency and environmental responsiveness, a necessary evolution for next-generation gaming [9].

2.3 Ghost in the Minecraft: Generally Capable Agents for Open-World Environments via LLM

In sandbox environments like Minecraft, the VOYAGER system exemplifies the creative potential of LLMs by translating natural language commands into complex in-game actions [11]. By leveraging GPT-4 as both a planner and goal generator, the system facilitates innovative gameplay. However, the computational cost associated with real-time code generation remains a critical bottleneck. This issue points to a broader need for optimization in LLM deployment, particularly for games with open-ended mechanics. Lightweight architectures or hybrid systems combining traditional game logic with LLM-driven decision-making could mitigate these constraints while preserving creative flexibility [11].

2.4 PokéLLMon: A Human Agent for Pokémon Battles with LLM

The use of LLMs in competitive strategy games, as demonstrated by PokéLLMon in Pokémon battles, further highlights their adaptability [12]. The agent's ability to predict opponent moves and execute effective strategies aligns with the overarching goal of enhancing player interactivity. However, the domain-specific fine-tuning required for such applications raises questions about the generalizability of these models. To address this, researchers could explore transfer learning techniques, enabling LLMs trained on one game type to adapt more easily to others. This approach would reduce the development burden and broaden the applicability of LLMs across gaming genres [12].

The broader implications of these findings point to a paradigm shift in game design [1][7]. As LLMs become more integral to gaming, their role is not just to augment existing mechanics but to redefine what interactivity means in virtual environments. However, these advancements are not without ethical and technical concerns. Resource requirements, potential biases in model outputs, and the risk of over-reliance on AI systems pose significant challenges. Addressing these issues requires a collaborative effort between AI researchers, game developers, and ethicists to ensure that the integration of LLMs aligns with both technological and societal expectations.

The application of LLMs in gaming highlights remarkable advancements in creating immersive and intelligent systems, though challenges persist in scalability, domain adaptability, and ethical implementation. The research by Socher et al. (2023) demonstrates how LLMs can transform social interaction in games like Diplomacy, while Dodge et al. (2023) illustrate their capacity to simulate complex negotiations among agents, revealing the potential and limitations of current models [8][9]. Similarly, the VOYAGER system in Minecraft emphasizes the creative possibilities of integrating natural language into gameplay, though resource demands remain a critical concern [11]. Efforts like PokéLLMon further underscore the strategic capabilities of LLMs in competitive scenarios, while also raising questions about the transferability of such systems across diverse game genres [12]. These developments serve as a foundation for ongoing exploration into how LLMs can reshape interactivity and engagement in gaming environments while addressing technical and

ethical considerations.

Research on Dynamic NPC Dialogue: Directly relevant to *Early Retirement* is the growing body of research focused specifically on using language models to generate dynamic NPC dialogue. Hypothetical or emerging studies, such as one potentially titled "Large Language Models for Dynamic NPC Interaction in Role-Playing Games" [13][14], likely investigate specific technical approaches. These include novel model architectures, such as integrating retrieval mechanisms to pull relevant lore or character history, advanced context management strategies handling conversation history effectively within model limitations, prompt engineering techniques tailored for character consistency and role-playing, or methods for controlling dialogue style and emotional expression. Complementing this technical focus are studies examining the impact on player experience, such as the work by Karppi et al. published in *Behavior & Information Technology*, which likely analyzes how AI-driven dynamic dialogue systems affect metrics like player immersion, engagement, perceived realism, and emotional connection when compared directly against traditional scripted dialogue approaches in controlled settings [13][15]. Our project contributes to this area by implementing and evaluating a specific end-to-end system configuration using voice input, a compact fine-tuned model SmolLM2, and cloud APIs.

2.5 Existing Systems and Related Work

Recent research provides essential context for *Early Retirement* by demonstrating both the remarkable potential and the inherent challenges of integrating advanced AI language models into interactive entertainment [2][3][7]. Studies involving complex strategic games, such as Meta AI's CICERO in *Diplomacy* [16], have showcased the ability of AI to simulate sophisticated social reasoning, strategic communication, and negotiation skills using natural language [8]. However, these examples also frequently highlight significant hurdles, particularly the substantial computational resources and inference times required for such complex deliberation, which often conflict with the real-time demands of many game genres.

Evaluations of models like GPT-4 in multi-agent negotiation scenarios [9][10] further support the view that LLMs are capable of engaging in nuanced, goal-driven communication. These systems demonstrate promise for applications in multiplayer or socially complex gaming environments. Nevertheless, such experiments also underscore ongoing issues with scalability and responsiveness when deployed in live systems, posing major obstacles for real-time use.

Other research avenues, such as NVIDIA's VOYAGER agent using GPT-4 in *Minecraft* [11], illustrate the potential for AI to interpret high-level player goals and autonomously generate complex in-game actions, fostering emergent gameplay. This work highlights the capacity of language models to bridge player intent and executable action in open-ended environments. However, it also underscores the performance bottlenecks associated with real-time planning and code generation using current large models.

Projects like PokéLLMon [12] demonstrate that domain-specific fine-tuning can yield expert-level AI performance in particular game contexts. By tailoring large models to a constrained domain,

researchers achieved compelling in-game behavior that closely mirrors expert human players. This result, while impressive, highlights the considerable effort potentially needed to adapt models for each unique game environment and raises concerns about the generalizability of such systems across diverse games.

Directly relevant to this project is the specific research focusing on leveraging language models for dynamic NPC dialogue. Emerging studies, including conceptual work potentially explored under identifiers like [14] and player experience analyses such as those published in *Behavior & Information Technology* [13][15], investigate architectures and evaluate the impact of AI-generated dialogue systems compared to traditional methods. This line of inquiry aims to overcome the limitations of high-quality but ultimately finite and predictable scripted systems, exemplified by games like *Red Dead Redemption 2*. While RDR2 achieves impressive contextual reactivity and character consistency through extensive pre-authored content, it lacks the capacity for true improvisation or response to arbitrary player speech.

AI dialogue systems, like the one implemented in *Early Retirement* using SmolLM2 [5], pursue the goal of enabling dynamic conversational interaction and direct voice communication. This involves accepting significant trade-offs: sacrificing the guaranteed polish and narrative control of scripted dialogue for dynamism, while confronting challenges such as ensuring AI response coherence, managing noticeable interaction latency, maintaining consistent character voice and personality through AI generation, and mitigating the risk of nonsensical outputs.

Underpinning nearly all research in applied game AI, and particularly relevant to this project, is the critical challenge posed by scalability and resource constraints [3][7][8]. The immense computational power and data required to train and deploy state-of-the-art language models often place them beyond the reach of academic projects or smaller development teams [5]. Even inference with pre-trained models can be demanding for real-time, low-latency applications, especially when considering multiple AI-driven NPCs. These constraints necessitate pragmatic compromises, such as selecting smaller, more efficient models like SmolLM2 in this project [5], employing aggressive optimization techniques like 4-bit quantization and QLoRA fine-tuning [4], limiting the scope of AI integration *e.g.*, focusing on one-on-one interactions, and accepting certain limitations in performance or capability. Successfully navigating these resource limitations while still delivering meaningful AI-driven interactions remains a central challenge in the practical application of advanced AI within the games industry and related research [1][2][17].

Chapter 3

Requirement Specifications

3.1 Proposed Methodology

The system implemented in *Early Retirement* employs a novel methodology centered on real-time, voice-driven AI interactions to enhance NPC dynamism within a Unity-based open-world game [13]. This approach utilizes a distributed architecture connecting the game client to external AI services and a custom backend server, replacing traditional scripted dialogue with dynamically generated responses [7][8].

The interaction pipeline initiates within the Unity client. Player voice input is captured and preprocessed, e.g., raw audio samples converted to a suitable format for speech recognition as will be discussed in section 4.5.2. This audio data is then streamed in real time to a speech-to-text (STT) service for accurate transcription.

Once a transcript is available, the Unity client formulates a contextual prompt. This prompt combines the player’s transcribed text with relevant metadata about the target NPC, such as their role and current emotional state. This structured prompt is then transmitted to the backend AI server.

The backend server, a Python application, is responsible for processing this prompt. It utilizes a fine-tuned compact language model of SmolLM2-360M [5] to generate a text-based response. This model was adapted for the specific conversational context of the game using a large, custom-generated dataset [6][12]. The server applies necessary post-processing to the raw model output to refine it into a coherent conversational line.

The cleaned text response is then sent back to the Unity client. Upon reception, the text is forwarded to a TTS service to synthesize it into audible speech. The resulting audio data is then played back in-game, delivering the NPC’s dynamically generated spoken response to the player.

This methodology leverages selected speech processing services, an efficiently fine-tuned language model, and appropriate communication protocols like WebSocket for low-latency text exchange and REST for other services to create a functional real-time AI dialogue pipeline. While the system aims for a responsive interaction targeting an average latency of ~ 3 -4 seconds, this can be influenced by various external factors.

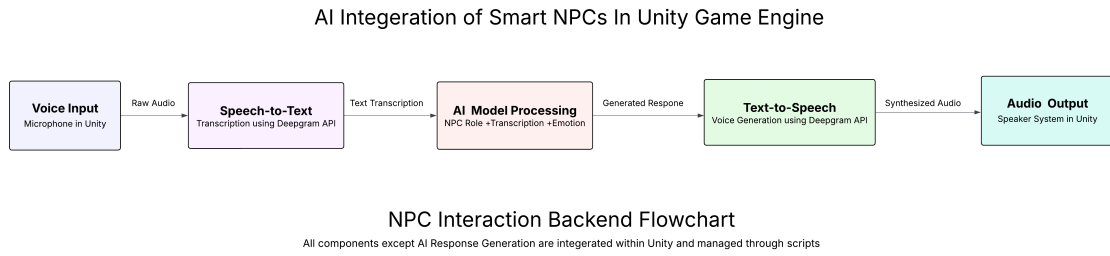


Figure 3.1: AI Integration Flowchart

3.2 Key Features

3.2.1 Real-Time NPC Interaction

A core feature of the system is its ability to facilitate real-time, voice-driven interactions between the player and Non-Player Characters (NPCs) [13]. This system enables NPCs to engage players in dynamic, contextually relevant, and emotionally resonant conversations [14]. Unlike traditional pre-scripted dialogues, which are often rigid and limited, the AI-driven approach processes player inputs in real time to generate personalized and meaningful responses tailored to the immediate game context.

The dialogue generation is designed to balance rapid interaction with emotional depth, aiming for low latency and seamless integration with gameplay mechanics. The underlying natural language understanding capabilities allow NPC responses to adapt to the player's tone, intent, and the evolving conversational context, thereby offering a more immersive and authentic interaction experience.

Furthermore, the system integrates speech-to-text and text-to-speech pipelines, enabling players to interact with NPCs using natural speech. This capability enhances accessibility and expands the range of possible interactions, aligning with the project's goal of fostering deep player engagement and realism. This approach to NPC dialogue aims to set a new standard for NPC intelligence and immersive gameplay [1][7].

3.3 Functional Requirements

3.3.1 AI Interaction Module

1. Voice Input Capture & Preprocessing – The system shall capture audio input from the player's designated microphone when interaction is active. It must preprocess the raw audio into 16-bit linear PCM format at the required sample rate for the STT API.

2. Contextual Prompt Formulation – Upon receiving a finalized text transcript, the system must construct a formatted prompt suitable for the fine-tuned SmolLM2 model, incorporating the transcript, the target NPC's predefined role, and a designated emotional context.

3. Speech-to-Text Conversion – The system shall accurately transcribe the player’s spoken English input into text using the integrated Deepgram STT API via a streaming WebSocket connection. It must handle session initiation and termination correctly.

4. AI Response Generation – The system shall transmit the formulated prompt via WebSocket to the deployed local server. The server must utilize the fine-tuned SmolLM2-360M model [5] to generate a single, coherent, and contextually relevant text response based on the received prompt.

5. Response Post-Processing – The backend server must apply cleaning procedures such as extracting the primary response line to the raw output of the language model before transmission.

6. Text-to-Speech Conversion – The system shall take the cleaned text response received from the AI server and convert it into audible English speech using the integrated Deepgram TTS API via a RESTful request, generating audio data in MP3 format.

7. Audio Response Delivery – The system shall load and play the received MP3 audio data through the game’s primary audio output device, synchronized appropriately with the interacting NPC.

3.3.2 Game Module

1. Player Locomotion: The player character must be controllable via standard keyboard/mouse, allowing movement including walking, running, and jumping within the game world.

2. Vehicle Interaction & Operation: The player must be able to enter designated car models found in the world, drive them using standard controls, and exit them.

3. NPC Interaction Initiation & Indication: The system must provide a clear visual indicator when the player is within the 3-meter interaction radius of an AI-enabled NPC.

3.4 Non-Functional Requirements

- **Performance:** The primary performance target for the end-to-end voice interaction cycle from the end of player speech to the start of NPC audio output is approximately 3 seconds. This target is achievable under optimal network conditions and when client or server resources are not heavily loaded. Actual latency may vary significantly due to external factors such as internet connection quality, Deepgram API response times, and AI server load. The system generally performs best when the host PC’s resources are idle and fully available for the game.
- **Resource Efficiency:** The system is designed to operate within typical mid-range gaming PC hardware specifications (Section 3.4). Resource efficiency is achieved through the use of the compact SmolLM2-360M language model [5], which is further optimized using 4-bit quantization and QLoRA fine-tuning [4] to minimize memory and computational

requirements during inference. Efficient client-side audio preprocessing implemented in C# within Unity also contributes to minimizing client load.

- **Robustness:** The system includes basic error handling for network communication and API interactions. However, it operates with known robustness limitations, particularly the potential for the AI model to generate irrelevant, nonsensical gibberish response, or contextually inaccurate dialogue [8]. Server-side post-processing such as first-line extraction and potential n-gram filtering aims to mitigate, but not eliminate, these occurrences.
- **Compatibility:** The game client is developed for and primarily tested on the Windows 10 (64-bit) operating system. While potentially functional on Windows 11 (64-bit), full compatibility is not guaranteed. Support for other operating systems is explicitly excluded.
- **Maintainability:** The software architecture promotes maintainability through modular code design. In Unity, functionality is divided among specific C# scripts. The backend AI server is a distinct Python application. This separation facilitates independent updates, testing, and debugging.
- **Scalability:** The system is designed and implemented for a single-player experience with one-on-one NPC interactions. Scaling the system to support multiple concurrent AI conversations or a large number of players simultaneously would present significant challenges due to resource constraints such as server computational power, API costs and network bandwidth [7][8], and is outside the current project scope.

3.5 Software Requirements

3.5.1 For Development

The development of *Early Retirement* necessitated a range of specific software tools and environments. The primary game client was built using the Unity Engine, specific version such as 2021.3 LTS or later, with C# scripting managed via Microsoft Visual Studio. Backend and AI model development relied on Python, version 3.12+ recommended, utilizing libraries such as *websockets* and *asyncio* for the server, alongside the Hugging Face ecosystem including *Transformers*, *PEFT*, *bitsandbytes*, *torch* for language model fine-tuning and inference, and *Pandas* for data handling. Visual Studio Code served as a key IDE for Python development. AI model fine-tuning experiments were conducted using web-based notebook environments like Google Colab and Kaggle Notebooks for access to GPU resources. Standard version control practices employed Git.

3.5.2 For End-Users (Players)

For end-users intending to play the game, the software requirements are minimal. The target operating system is Windows 10 (64-bit) or Windows 11 (64-bit), with the necessary Microsoft DirectX 12 runtime libraries installed. The game itself is distributed as a self-contained installation

package; therefore, players only need to obtain and install this package. As stated for simplicity, end-users can "just download this game and you are good to go," assuming their operating system and graphics drivers are reasonably up-to-date. However, an active and stable internet connection is a mandatory prerequisite during gameplay for the AI-driven voice interaction features to function correctly, as they rely on communication with external cloud services.

3.6 Hardware Requirements

The following hardware specifications outline the minimum and recommended configurations for running the "Early Retirement" game client application.

A Solid State Drive (SSD) is highly recommended for optimal loading times, even if not strictly listed as minimum. A stable internet connection is mandatory for the AI-driven NPC interactions to function.

Table 3.1: Hardware Requirements for "Early Retirement" Game Client

Component	Minimum Requirement	Recommended Requirement
Operating System	Windows 10 (64-bit)	Windows 10 / 11 (64-bit)
Processor (CPU)	Intel Core i5 / AMD Ryzen 5 series	Intel Core i7 / AMD Ryzen 7 series
Memory (RAM)	8 GB RAM	16 GB RAM
Graphics Card (GPU)	NVIDIA GeForce GTX 1060 / AMD Radeon RX 580	NVIDIA GeForce RTX 2060 / AMD Radeon RX 6700 XT
Storage	1.5 GB available space (SSD Recommended)	2 GB available space (SSD Strongly Recommended)
DirectX	Version 12	Version 12
Network	Broadband Internet Connection (Stable)	Broadband Internet Connection (Stable, Low Latency)
Input	Keyboard, Mouse, Functional Microphone	Keyboard, Mouse, Good Quality Microphone

3.7 Use Cases

This section outlines the primary ways a player interacts with the system, particularly focusing on the novel AI-driven features and the scenarios are described here.

Initiate and Conduct Voice Conversation: This is the core use case. The player approaches an AI-enabled NPC, activates voice input, speaks, and engages in a turn-based conversation where the NPC's responses are generated dynamically by the SmolLM2 model [5] and delivered via synthesized speech. This covers the full STT-AI-TTS pipeline.

Basic Game Actions: Standard use cases involving player control via keyboard, mouse, or gamepad for navigation such as walking, running, jumping, driving vehicles, and interacting with the game's save or load system. While separate from the AI dialogue, these are essential functional requirements for the game context.

Chapter 4

Design

The design of the *Early Retirement* system is centered on creating a dynamic and immersive interactive experience within its open-world Personal Computer (PC) game framework. The project, developed in the Unity engine, aims to move beyond traditional static NPC dialogue by implementing real-time, voice-driven interactions powered by AI [7][13]. This chapter details the architecture and design choices made to integrate the game client with AI services and a custom backend, which hosts the fine-tuned language model [5]. The fundamental design goal is to enable players to speak naturally to NPCs and receive contextually relevant, dynamically generated spoken responses, thereby fostering a greater sense of realism and engagement. The subsequent sections will elaborate on the system architecture, design constraints, methodologies, and distribution of responsibilities that guided the development of the functional pipeline for these interactions, all while operating within defined project constraints.

4.1 System Architecture

The system architecture for *Early Retirement* is designed to facilitate real-time, voice-driven NPC interactions. It integrates several key modules to manage the flow of information and processing between the player and the AI-driven NPCs. The core modules include:

1. **Input Processing Module:** Responsible for capturing player voice input and preparing it for the language model.
2. **Language Model Module:** At its heart is the SmoLLM2 model, which processes the contextualized player input and generates relevant dialogue responses [5].
3. **Game State Module:** This module maintains and provides the necessary game context, e.g., NPC roles, emotional cues, to ensure that NPC responses are appropriate to the ongoing interaction.

The interaction between these modules and external services is detailed in the data flow summary below.

4.1.1 Data Flow Summary

The interaction initiates with player voice input captured and preprocessed by the Unity Client. PCM audio bytes are streamed via WebSocket to Deepgram STT. Deepgram returns JSON transcripts via the same WebSocket. The Unity Client extracts the final transcript, formulates a JSON prompt including NPC context, and sends this via a separate WebSocket connection to the python-based AI Server that is locally deployed. The server runs inference using the SmolLM2 model, post-processes the result, and sends a JSON response containing the dialogue text back via WebSocket. The Unity Client receives this text, sends it in a JSON payload via REST (HTTP POST) to Deepgram TTS. Deepgram TTS returns MP3 audio data. The Unity Client receives this data, loads it into an `AudioClip` from a temporary file, and plays it through an `AudioSource` as the NPC's voice.

4.1.2 Rationale for Architectural Choices

Offloading AI Inference: The primary reason for the client-server split is to move the computationally demanding language model inference off the player's machine [8]. This prevents the game's frame rate and responsiveness from being negatively impacted by AI processing, ensuring a smoother gameplay experience. While the local AI server deployment was lightweight at approximately 17MB, offloading AI inference generally, can be retained as it's a specific project finding influencing the perception of this choice.

Leveraging Cloud APIs: Using established cloud services like Deepgram for STT and TTS provides access to high-quality, specialized models without requiring the project team to develop these complex components from scratch.

WebSocket for Real-time Interaction: The choice of **WebSocket** for client-server communication and client-to-STT exchange helps reduce latency compared to using repeated HTTP requests. This is important for keeping the conversation close to real time.

REST for TTS Simplicity: Since Text-to-Speech usually returns a complete audio clip from a text input, the simpler REST API using HTTP POST is sufficient. It is also easier to use in Unity through *UnityWebRequest* instead of handling another streaming connection.

Modularity: Designing distinct components such as client modules, server application, and external services allows for independent development, testing, and potential future upgrades or replacements, such as swapping the language model or TTS provider.

4.2 Design Constraints

The design and subsequent implementation of *Early Retirement* were fundamentally influenced by a set of constraints that defined the boundaries of what could be achieved. These constraints necessitated specific technical choices and trade-offs throughout the development process.

- **Real-Time Processing:** The goal of creating a conversational interaction, even if not perfectly fluid, imposed a soft constraint on the acceptable end-to-end latency. The target of approximately 3 seconds was deemed borderline acceptable for turn-based interaction

but required careful optimization and acknowledgment of external dependencies (network, APIs). This constraint directly influenced choices against using much larger, potentially slower-inferencing language models [8] and necessitated the use of efficient communication protocols.

- **Resource Efficiency:** As an academic project with limited budget and access to high-performance computing, training and deploying large multi-billion parameter language models was infeasible [7][8]. This constraint directly led to the selection of a compact language model [5] that offered a compromise between capability and size. Furthermore, the need to fine-tune this model on accessible hardware mandated the use of parameter-efficient fine-tuning techniques such as QLoRA [4]. These techniques drastically reduce the memory footprint during training, for example by quantizing the base model and training smaller adapter layers [4], making the process viable on available GPU resources. This also implies that the deployed model for inference is relatively lightweight.
- **Language Coherence:** Generated responses must adhere to the narrative tone and context.
- **Scalability Limitation (Single Player Focus):** Resource constraints also dictated the system's scalability [8]. The architecture was designed and implemented with a single player interacting with one NPC at a time. Supporting multiple simultaneous AI conversations would require a significantly more robust backend, which was beyond the project's scope. This limitation was accepted early in the design phase.
- **AI Response Quality (Compact Model Trade-offs):** Opting for a compact language model inherently constrained the potential quality and complexity of the generated dialogue [5]. While fine-tuning aimed to improve its performance for the specific task, the design had to account for the model's limitations in understanding deep context, maintaining long-term consistency, and avoiding occasional undesired outputs [3][8]. The design included post-processing steps as a mitigation but acknowledged that achieving flawless human-like dialogue quality was not feasible with the chosen model class.
- **Platform Constraint (Windows 10/11):** Development resources were focused on delivering a functional prototype for the Windows PC platform. Targeting only Windows 10/11 simplified development by eliminating the need to handle cross-platform compatibility issues.
- **Network Dependency:** The reliance on cloud-based services for STT and TTS makes a stable, reasonably low-latency internet connection a strict requirement for the core AI features to function.
- **API Dependency:** The system's functionality is directly dependent on the continued availability, performance, and API specification stability of the chosen STT and TTS services.

4.3 Design Methodology

4.3.1 General Approach

The system follows an iterative and modular design methodology to ensure adaptability and scalability:

1. **Requirement Analysis:** Identify functional and non-functional requirements based on player interaction needs.
2. **Modular Development:** Design and develop independent modules for key functionalities like input processing, language modeling, and response delivery.
3. **Integration and Testing:** Gradually integrate designed modules and test for coherence, performance, and scalability.

4.3.2 Detailed Design Methodology

The development process for *Early Retirement* adopted an **Incremental Model** combined with **Agile Planning** principles. This hybrid methodology was strategically selected as the most suitable approach given the project's inherent complexities, the exploratory nature of integrating complex AI technologies within a game, and the specific dynamics of the two-person development team.

Workflow Description: This workflow, with its distinct phases, is visually represented in the project Gantt charts presented in Section 1.8[e.g., Figures 1.1, 1.2, and 1.3], which detail the specific timelines and task allocations. The key phases were:

- **Phase 1: Foundation & Feasibility (Joint Effort):** The project commenced with a feasibility study and intensive planning. Both team members collaborated on defining the vision, designing the world, identifying components, and outlining scope.
- **Phase 2: Parallel Research and Skill Development:** Asif focused on Unity development; Raad explored AI model training, fine-tuning, and voice APIs. They synchronized regularly.
- **Phase 3: Module Development:** Independent module development — Unity systems by Asif, AI backend and datasets by Raad.
- **Phase 4: Integration:** Joint milestone meetings, testing modules together and planning next sprints.
- **Phase 5: System Assembly and Refinement:** Modules were incrementally connected; performance, coherence, and quality were refined.
- **Phase 6: Final Testing and Documentation:** Comprehensive system testing and co-authoring of documentation.

Suitability: This approach was well-suited to the project for several reasons:

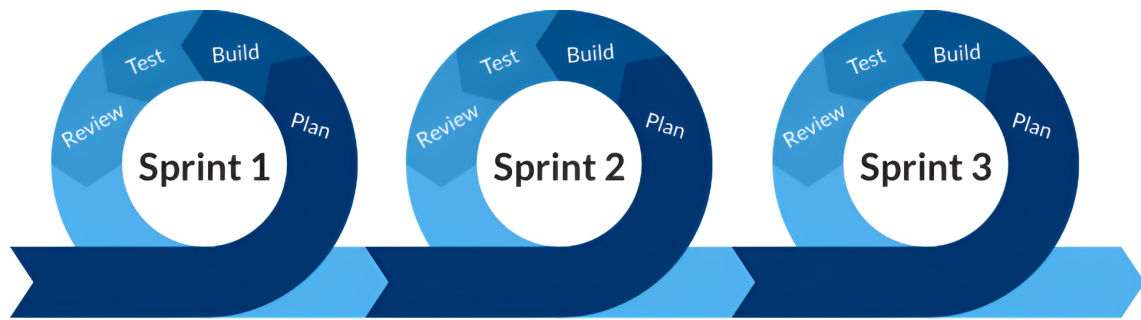


Figure 4.1: Agile-Incremental Development Methodology

- It allowed for flexibility and adaptation to emerging technical challenges and research findings.
- Breaking down the project into smaller, incremental modules helped in managing integration complexity.
- It encouraged specialization and ownership of tasks while ensuring collaboration.
- The iterative nature enabled continuous testing and feedback, reducing late-stage risks.
- It effectively supported small-team collaboration and coordination.

4.4 Distribution of Responsibilities

4.4.1 Project Direction and Joint Foundations

Initial design, feasibility analysis, narrative and world-building were collaborative. Raad led creative and technical vision; Asif led client-side development. Both contributed to planning and testing.

4.4.2 Game Development

Asif led the development of the *Early Retirement* game client using the Unity engine. His responsibilities encompassed the creation of the open-world environments, including the cities of Vata and Fehari, and the implementation of core gameplay systems in C#. This included programming player locomotion, vehicle mechanics, basic NPC movement systems, and developing the necessary user interface elements for game interaction.

4.4.3 AI System Development and Creative Oversight

Raad directed the AI system's end-to-end development and provided creative leadership for the project. His technical contributions spanned the entire AI pipeline, including dataset generation and curation, language model fine-tuning, and the implementation of the WebSocket-based Python backend server. He also managed the integration of external speech-to-text (STT) and text-to-speech

(TTS) APIs, along with developing response post-processing logic, while simultaneously guiding the narrative design and ensuring the quality and coherence of AI-generated dialogue.

4.5 System Architecture

The System architecture orchestrates a real-time voice interaction loop. Player voice input is captured by the Unity Game Client, which utilizes an external Speech-to-Text API of Deepgram to convert speech to text. This text, formulated as a JSON prompt, is then sent via WebSocket to the Python Backend Server where the NPCModel (SmoLLM2-360M) generates a response. The AI response is returned to the client, which employs an external Text-to-Speech API of Deepgram to convert it back into audio, delivered as the NPC's voice to the player.

Early Retirement: High-Level Architecture

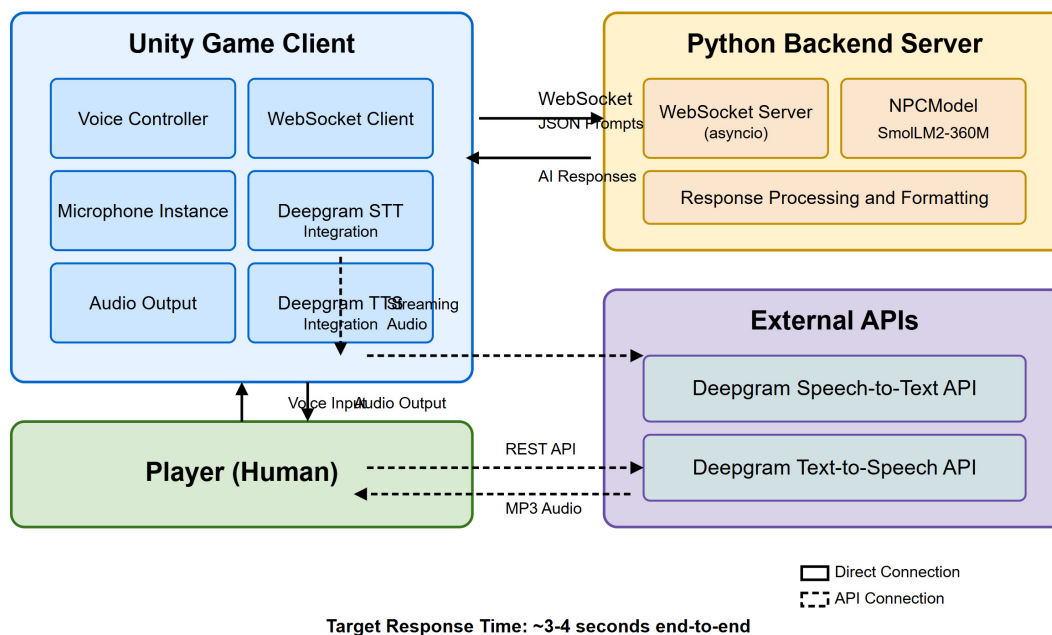


Figure 4.2: System Architecture Diagram

4.6 Sequence Diagram

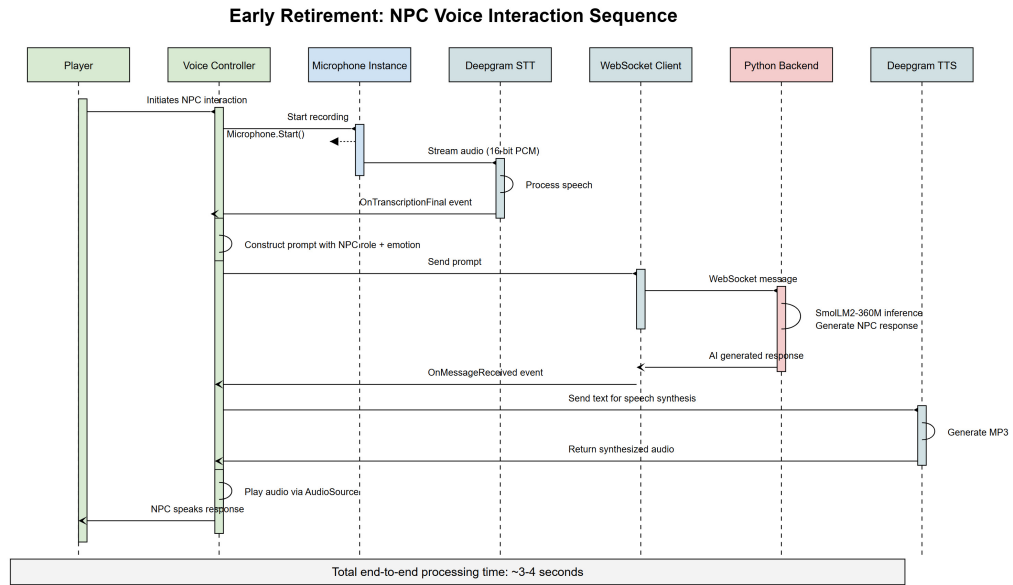


Figure 4.3: NPC Voice Interaction Sequence Diagram

4.6.1 Input Processing Module Design

- Uses `Microphone.Start()` for audio capture.
- Polls `Microphone.GetPosition()` to detect new data.
- Extracts and converts audio (F32 to I16 PCM).
- Sends processed bytes to STT module.

4.6.2 Language Model Module Design

- Uses compact quantized base model (e.g., SmolLM2) [5].
- Fine-tuned using PEFT (e.g., QLoRA) [4].
- Dataset structured as: role, input, emotion, response [6].
- Custom collator masks tokens for loss computation.
- Saves LoRA adapter weights and tokenizer.

4.6.3 Communication Module Design

- Uses `ClientWebSocket` in Unity.

- Asynchronous listener for incoming AI responses.
- Serializes/deserializes JSON messages.
- Sends prompts; manages lifecycle and errors.

4.6.4 Response Delivery Module Design

- Sends text to Deepgram TTS API.
- Receives MP3; saves to temp file.
- Loads via `UnityWebRequest` and plays via `AudioSource`.

4.6.5 Voice Controller Module Design

- Coordinates transcription, model querying, and TTS.
- Manages context like NPC role and emotion.
- Operates via event-driven architecture.

4.6.6 Backend Server Module Design

- Uses `asyncio` and `websockets` in Python.
- Loads model and tokenizer with adapters.
- Formats and validates prompts.
- Uses sampling methods in `generate()`.
- Sends back processed response over `WebSocket`.

4.7 Game World and Inhabitants

4.7.1 Game Setting: Vata and Fehari

The game unfolds across two primary world zones, Vata and Fehari. These areas are designed with distinct visual atmospheres and environmental features, contributing to the overall immersive experience. The urban landscape is constructed using some modular buildings, allowing for diverse and realistic city environments. Key locations a player encounters include:

- **Urban Zones (Vata & Fehari):** These are the main regions where gameplay occurs, each promising a unique environmental design and atmosphere.
- **Tyler's Residence:** The primary setting for the initial dramatic events, including Tyler's traumatic home invasion. This location serves as the catalyst for Tyler's descent into vengeance and his subsequent actions throughout the narrative.

- **Baez's House:** A recurring sanctuary and meeting point where Tyler often seeks out his best friend Baez for support, counsel, and to initiate new plans or missions. It acts as a familiar touchstone for Tyler, representing loyalty and a connection to his past life as he navigates his perilous new path.
- **Roman's Pub/Base:** This location serves as Roman's operational hub for his illicit dealings and a key meeting point for Tyler and Baez when seeking information or undertaking missions for him. It's depicted as a place where crucial mission briefings, tense negotiations, and tests of loyalty unfold.
- **Robbers' Hideout (Building Top Floor):** The site of a climactic confrontation where Tyler tracks down Cobbler and the other robbers responsible for his family's murder. This location is pivotal as Tyler exacts his revenge and uncovers devastating truths that further fuel his actions.

4.8 GUI Design

The Graphical User Interface (GUI) of *Early Retirement* is designed for intuitive access to essential game functions while maintaining an immersive third-person narrative experience. The design philosophy emphasizes minimalism and clarity, ensuring UI elements support, rather than distract from, gameplay. The following figures showcase key elements of the game's Graphical User Interface (GUI) and visual design.



Figure 4.4: Initial Start Menu Interface



Figure 4.5: In-Game Main Menu Options

The above two figures show the menu buttons of a game. The first displays the initial start menu with PLAY and QUIT options, while the second shows the in-game main menu featuring options like Resume, New Game, Map, and Quit.

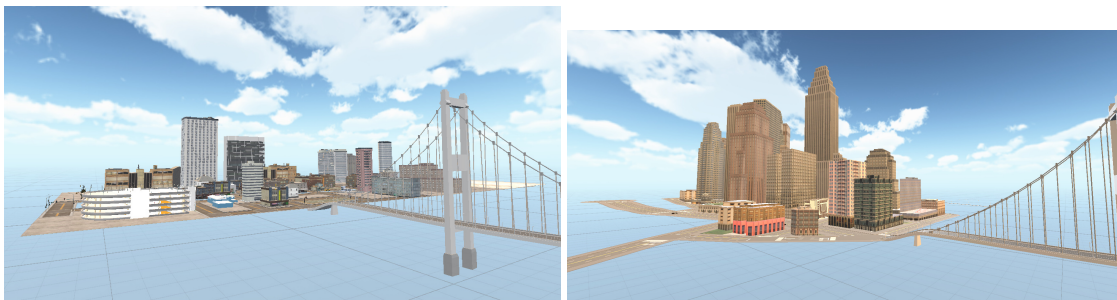


Figure 4.6: Bird eye view of Fehari & Vata

The images present the game world's urban landscapes, highlighting the distinct city layouts of Fehari and Vata with detailed structures and waterway crossings. A complementary map view outlines strategic zones and spatial connections, offering both visual and navigational context.



Figure 4.7: Map Overview of Game World Zones (Vata & Fehari)



Figure 4.8: Tyler Wayne (Protagonist)

Figure 4.8 visually presents Tyler, a 34-year-old American with a good physique and a strong financial background. Tyler is methodical, determined, and often bold. However, he is currently battling depression, which manifests as cynicism and sensitivity despite his natural eloquence and charisma. His outlook is tainted by deep-seated pessimism.



Figure 4.9: Netan (Antagonist)

Figure 4.9 presents a 40-year-old individual with a good physique, who is smart, highly educated, and extremely rich. Netan is analytical and competitive but also egotistical, critical, hostile, and condescending. While possessing wisdom and resourcefulness, he is marked by arrogance, stubbornness, and dishonesty.



Figure 4.10: Anne & Qash (Tyler's Wife & son)

Figure 4.10 presents portraits of Anne, Tyler's empathetic and expressive wife dressed in a dark, stylized outfit, and their 7-year-old son Qash, shown wearing a colorful shirt and jeans. Anne is described as emotionally deep and overthinking, while Qash is characterized as witty, hyperactive, and joyful. The images capture both the maturity of Anne and the lively energy of her young son.



Figure 4.11: Figure 4.11: Baez (Tyler's Best Friend)

Figure 4.11 presents Baez, Tyler's best friend and mentor, depicted as a full-body portrait. He is described as an Afro-American former wrestler turned cook, characterized by his objective.



Figure 4.12: Samuel (Gun Supplier/Mechanic) & Roman(Drug Dealer)

Figure 4.12 shows Samuel (left), a bald, bespectacled mechanic and gun supplier with a confrontational demeanor, and Roman (right), a formally dressed drug dealer described as analytical, energetic, yet doubtful, potentially serving as a sidekick.



Figure 4.13: Dalas(Junkie) & Cobbler(Robber)

Figure 4.13 introduces Dalas, a sociable but impulsive "Junkie" depicted as a Black man in a green jacket with a cigarette, and Cobbler, a "Robber" shown wearing sunglasses and a gold chain, described as daring, combative, and impulsive.



Figure 4.14: Tob & Todd (Twins)

Figure 4.14 introduces Tob and Todd, identical twin brothers from Baez's neighborhood, depicted in separate panels showing their nearly identical appearance and matching attire of button-up shirts and dark trousers. Described as a "lethal combination," they are portrayed as serious, minimal-speaking, and stern individuals.



Figure 4.15: Example Generic NPCs in the Game World

This showcases examples of generic Non-Player Characters (NPCs) that populate the game world of "Early Retirement" to create an ambient environment. It displays a police officer in a T-pose, a worker with a broom, and a pedestrian with a shoulder bag, representing the diverse ambient NPCs assigned roles to support in-game AI context.



Figure 4.16: First mission with & without torch View

compares two views of the game's first mission, demonstrating the significantly reduced visibility in a dark indoor environment without the torch (top panel).



Figure 4.17: Second Mission View

This video game screenshot shows a third-person view of a player character in a dark jacket and jeans, standing on a sidewalk in a modern urban setting with multi-story buildings and a street. A mini-map in the lower-left corner indicates active gameplay from what the report identifies as the "Second Mission View" of "Early Retirement".

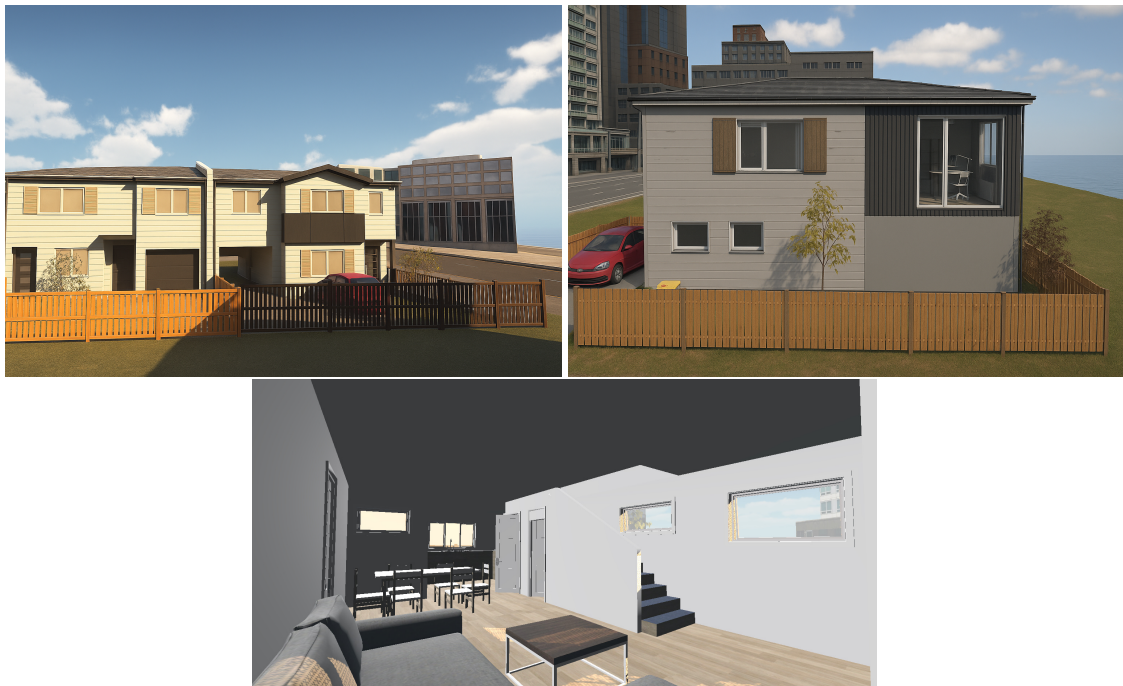


Figure 4.18: Tyler's Safehouse

This set of images displays "Tyler's Safehouse," a key location in the game "Early Retirement". Two panels provide exterior views of a suburban-style house with a wooden fence, set against a backdrop of city buildings, while a third panel reveals a sparsely furnished interior with a staircase, couch, and table, illustrating the safehouse's appearance.



Figure 4.19: Baez's House Ext. & Int.

These two images showcase Baez's house, a recurring sanctuary in "Early Retirement" where Tyler seeks support. The left image displays the house's gabled exterior with an attached garage, set against a backdrop of taller city buildings. The right image reveals a modern interior space, likely a kitchen or open-plan living area, featuring contemporary cabinetry and an island.



Figure 4.20: AI Enhanced aerial View of Fehari & Vata

These two AI-enhanced aerial views showcase the distinct urban layouts of Fehari and Vata, the primary game world zones in "Early Retirement". The top image displays an island city, likely Fehari, characterized by its irregular road network, varied building structures, and surrounding water. The bottom image presents a more grid-like city layout, presumably Vata, with uniform blocks, taller buildings, and an adjacent barren landscape.



Figure 4.21: Player-Accessible Vehicles

These image displays three distinct player-accessible vehicles available in "Early Retirement" a blue classic sedan, a silver modern sports sedan, and a yellow muscle car—each shown in an urban street setting.

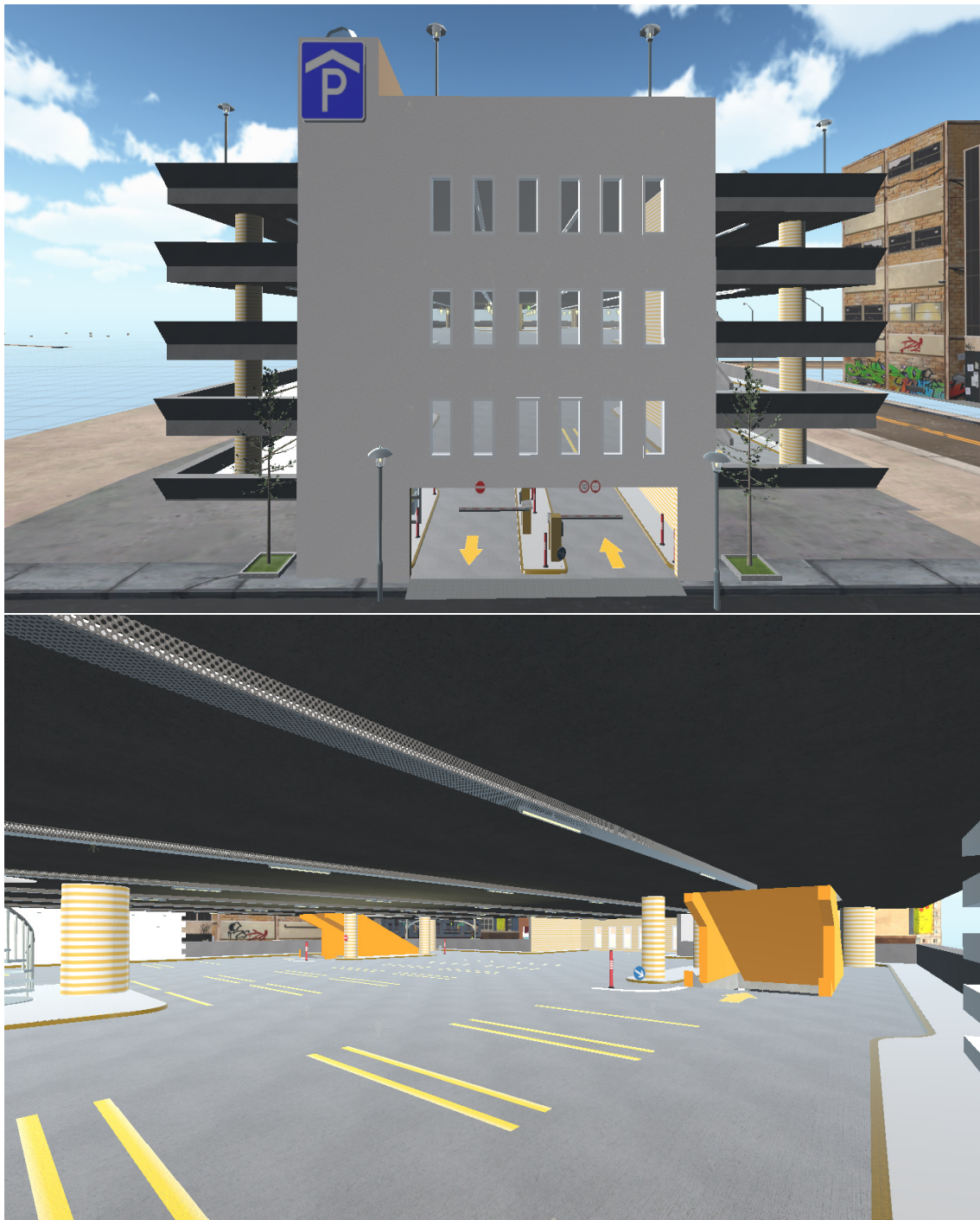


Figure 4.22: Car Parking Ext. & Int.

These images depict a multi-level car parking structure from the game "Early Retirement," showcasing its modern exterior design with multiple tiers (top panel) and a view of an interior parking level with marked bays and structural columns (bottom panel).



Figure 4.23: Police Station Ext, Int. & Jail

These images show "Early Retirement's" police station—a key landmark—detailing its exterior, interior reception/lobby, and a jail cell.



Figure 4.24: Hospital & park

These images display "Early Retirement's" hospital exterior, a key landmark, alongside an adjacent open park area within the game's urban environment.



Figure 4.25: Bank Ext. & Int.

These images illustrate "Early Retirement's" bank, a key landmark, presenting its formal exterior and the spacious interior lobby with teller counters and waiting areas..

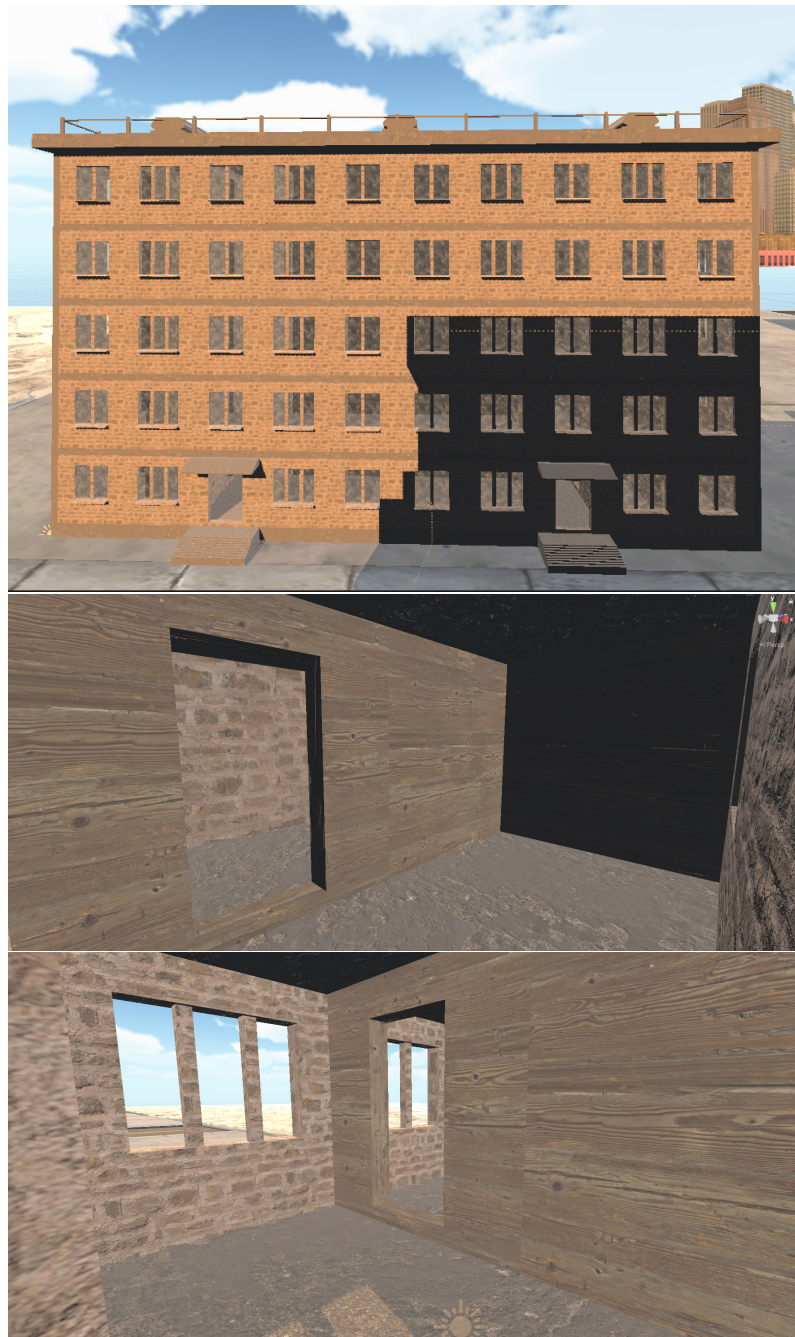


Figure 4.26: Netan's building for final mission Ext. & Int.

These images show Netan's building for the final mission in "Early Retirement," presenting its multi-story exterior and sparsely detailed interior rooms. The visuals establish this key antagonist location, setting the scene for the game's climactic encounter.

Chapter 5

System Implementation

5.1 System Architecture Implementation

The implementation of *Early Retirement* involved the construction and interconnection of the core components as specified in the system architecture design (Section 4.2). The development of the Unity game client proceeded using the Unity Engine (e.g., 2021.3 LTS), where C# scripts were written to bring to life the designed gameplay logic, user interface elements, and the client-side aspects of the speech interaction pipeline. The Unity editor served as the environment for scene construction, script attachment, and iterative testing. The outcome of this client-side development was a compiled Windows PC application, bundling all necessary assets and executable code.

Concurrently, the backend AI server, designed to host the fine-tuned SmolLM2 model [5], was implemented in Python. This implementation utilized the `websockets` library and `asyncio` to realize the designed WebSocket-based communication. The `NPCModel` class, as designed, was implemented to handle model loading, inference execution, and response formatting. This server was configured for local execution, either on the primary development machine or another device on the local network.

The integration of external Deepgram services was actualized through C# scripts within the Unity client. The designed flow of capturing audio via the Unity Microphone API, transmitting it to Deepgram’s STT service over WebSocket, and sending AI-generated text to the TTS service via HTTP POST was implemented using Unity’s `UnityWebRequest` and .NET’s `ClientWebSocket` classes. API key management was implemented through client-side configuration, with the understanding that more robust security measures would be necessary for a production environment. This methodical implementation process successfully translated the designed modular architecture into an operational real-time voice interaction system.

5.2 Architecture and Component Integration

The operational success of *Early Retirement* was contingent upon the effective integration of its distinctly developed architectural components: the Unity game client, the Python-based AI backend

server, and the external Deepgram APIs, as detailed in the system design (Chapter 4). This section focuses on the practical processes and key considerations that enabled these components to function cohesively.

A primary integration task was the establishment of robust communication channels as per the design. For the main AI interaction loop—transmitting transcribed player input to the AI server and retrieving generated dialogue—the designed WebSocket connection was implemented between the `WebsocketClient.cs` script in Unity and the Python backend. This involved ensuring that JSON payloads were correctly serialized and deserialized across the C# and Python environments. The Python server’s implementation, leveraging the `websockets` library with `asyncio`, was specifically developed to handle concurrent client connections, manage asynchronous message loops, and route data to the AI model inference logic as designed.

The integration with Deepgram’s STT service also involved implementing the designed WebSocket-based audio streaming from `DeepgramTranscription.cs` in Unity and handling asynchronous transcription results. For Deepgram’s TTS service, the designed RESTful HTTP POST approach was implemented in `DeepgramTTSRestAPI.cs`.

5.3 Implementation Details

5.3.1 Unity Client Implementation

Within the Unity client, the interconnection of core C# scripts—`Microphone_Instance.cs`, `DeepgramTranscription.cs`, `WebsocketClient.cs`, `DeepgramTTSRestAPI.cs`, and `VoiceController.cs`—was implemented in accordance with their respective low-level designs (Section 4.5.4). The `VoiceController.cs` script serves as the central orchestrator, following an event-driven architecture. For instance, the `OnTranscriptionFinal` event from `DeepgramTranscription.cs` triggers prompt construction and dispatch through `WebsocketClient.cs`. Likewise, upon receiving a response from the backend AI server, `WebsocketClient.cs` raises an event that initiates speech synthesis via script of `DeepgramTTSRestAPI`. This modular and asynchronous event-driven approach proved effective in managing real-time network communication and control flow within Unity.

5.3.2 Integration Challenges Encountered

Several technical challenges, anticipated in the design phase or emerging during development, were addressed during the component integration process.

Achieving reliable, low-latency real-time communication was a significant focus. The designed shift from potential HTTP request-response cycles to WebSockets for the primary AI dialogue with the custom Python server was implemented. This required careful implementation of asynchronous WebSocket handling in both the C# client (using `System.Net.WebSockets`, often with `async/await`) and the Python server (using the `websockets` library with `asyncio`) to ensure non-blocking I/O and manage concurrent connections effectively.

Implementing the direct API communication with Deepgram from Unity, using `UnityWebRequest` and `.NET WebSocket` classes as planned, required meticulous attention to header construction, JSON data serialization, and the management of asynchronous operations (Unity Coroutines and `C# async/await`). This direct implementation provided fine-grained control and helped avoid potential SDK compatibility issues within Unity's runtime environment. Handling binary audio data for STT and MP3 data for TTS also necessitated careful byte-level manipulation during implementation.

Ensuring consistent data formatting (primarily JSON) between C# and Python required rigorous testing of serialization and deserialization logic on both client and server sides to prevent runtime errors.

Finally, managing multiple concurrent asynchronous operations (STT streaming, AI server I/O, TTS calls, audio loading) within Unity's main game loop was implemented using a combination of Unity's Coroutines and C#'s Task-based Asynchronous Pattern. This was crucial to prevent UI freezes and ensure that network requests and responses were processed smoothly in the background. Debugging issues across this distributed setup also demanded a systematic approach to tracing data flow and isolating problems.

5.4 Tools and Technologies Implementation

The development of the *Early Retirement* prototype was accomplished by employing the specific tools and technologies identified in Section 1.5. This section briefly outlines how these were practically applied during the implementation phase. The Unity Engine, with C# scripting in Microsoft Visual Studio, served as the primary environment for all client-side development, including gameplay mechanics, UI, audio processing, and network communication initiation. For the AI backend development and the implementation of model fine-tuning scripts, Python was utilized extensively, with Visual Studio Code as the main editor, supported by libraries from the Hugging Face ecosystem and PyTorch.

The Deepgram API was directly integrated for its STT and TTS capabilities as per the design specifications. Communication protocols, namely `WebSocket` for AI server interaction and STT and REST for TTS, were implemented using the respective `.NET` and Python libraries. While Docker was used during certain development stages for ensuring consistent Python environments, particularly for AI experimentation, the final iteration of the system described for this project phase was run with a directly hosted local Python server to simplify the operational setup.

5.5 Development Environment

The diverse nature of the project, spanning game client development, backend AI services, and computationally intensive model training, necessitated the use of multiple specialized development environments, each chosen for its strengths in particular domains. For all C# scripting within the Unity project, Microsoft Visual Studio served as the primary IDE. Its robust debugging capabilities,

seamless integration with the Unity editor, and comprehensive code completion features were invaluable for developing and iterating on client-side game logic and AI interaction scripts.

The Python backend server, which utilizes the `websockets` library and `asyncio` for handling real-time communication and AI model inference, was developed primarily using Visual Studio Code. VS Code's lightweight nature, powerful Python support through extensions, integrated terminal for running scripts and managing dependencies, and Git integration made it an efficient choice for this component. It was also suitable for creating and managing auxiliary Python scripts used in data processing or offline testing.

For the AI model fine-tuning phase, specifically the training of the SmolLM2 model using QLoRA, web-based notebook environments like Google Colab and Kaggle Notebooks were indispensable. These platforms provided crucial free-tier access to GPU resources, which are essential for training language models [5][4]. The interactive notebook interface facilitated experimentation with the fine-tuning script, managing datasets, installing required libraries, executing training runs, monitoring progress, and saving the resultant model adapters. This multi-IDE workflow allowed each development task to be performed using the most appropriate and efficient toolset available to the team.

5.6 AI Model Implementation

The NPC dialogue in *Early Retirement* is powered by the compact SmolLM2-360M model [5], chosen for its balance of conversational ability and resource efficiency. We quantized the base model to 4-bit using `bitsandbytes` and applied QLoRA adapters via the Hugging Face PEFT library [4]. A custom dataset of 20,182 prompt–response pairs was tokenized [6] (with input tokens masked for loss calculation), collated for padding, and fine-tuned using mixed-precision (bf16) training under the Hugging Face `Trainer` [4]. Only the LoRA adapter weights (in `safetensors` format), plus the model's config and tokenizer, were saved for deployment.

At runtime, our Python backend (built on `asyncio` and `websockets`) loads the quantized model and LoRA adapters in an `NPCModel` class. Incoming prompts are parsed, tokenized, and truncated to fit the context window; `model.generate()` is then called with sampling parameters to produce dialogue. The output tokens are decoded, post-processed to extract the first coherent line, and returned as JSON over the WebSocket connection.

5.7 Voice Interaction Walkthrough

The practical implementation of a single, complete voice interaction turn within *Early Retirement* demonstrates the orchestrated use of the selected tools and technologies across the Unity client and the Python backend server.

The sequence begins with audio input in the Unity client. The `Microphone_Instance.cs` script, leveraging `UnityEngine.Microphone`, captures raw audio. This script then performs essential preprocessing in C#, converting the 32-bit float samples to 16-bit PCM byte arrays. These bytes are

passed to `DeepgramTranscription.cs`, which manages a `System.Net.WebSockets` connection to stream this audio data to the Deepgram STT service. Upon receiving a final transcription in JSON format, an event is triggered.

This event is handled by `VoiceController.cs`, which formulates a JSON prompt incorporating the transcribed text, the specific NPC's role, and a contextual emotion. This prompt is then handed to `WebsocketClient.cs`. Utilizing another `ClientWebSocket` instance, `WebsocketClient.cs` sends this JSON payload asynchronously to the locally running Python AI server.

The Python server, built with the `websockets` library and `asyncio`, has an `async def` handler function awaiting incoming messages on the established `WebSocket` connection. Upon receiving the prompt JSON, this handler parses the data and invokes the `NPCModel.generate_response` method. Inside this method, the fine-tuned `SmolLM2` model [5] loaded using Hugging Face Transformers and PEFT adapters [4] performs inference based on the prompt and configured sampling parameters. The generated text undergoes post-processing to produce the final dialogue line. This response is then packaged into a JSON object and sent back to the Unity client via `await websocket.send`. Back in Unity, `WebsocketClient.cs` receives the AI's response and triggers another event. `VoiceController.cs` handles this, taking the textual response and passing it to `DeepgramTTSTestAPI`. This script constructs a JSON payload with the text and uses `Networking` to send an HTTP POST request to the Deepgram TTS REST API. Deepgram responds with MP3 audio data. `DeepgramTTSTestAPI.cs` then manages the retrieval of this audio data, its conversion into a `Unity AudioClip`, and finally, initiates playback using a `UnityEngine.AudioSource`, allowing the player to hear the NPC's dynamically generated response. This entire multi-step process illustrates the practical application of the chosen technologies for each conversational turn.

Chapter 6

System Testing and Evaluation

6.1 Reason For Testing

Systematic testing and thorough evaluation are indispensable stages in the software development lifecycle, critical for ensuring the delivery of a high-quality, reliable, and effective software product. For the *Early Retirement* project, with its intricate blend of game mechanics and advanced AI-driven voice interaction, a structured testing approach was vital to identify and mitigate potential issues early, build confidence in the system's functionality, and ultimately determine its success in achieving its innovative goals. The testing process was broadly guided by two distinct yet complementary objectives: Verification and Validation.

6.1.1 Verification

Verification addresses the fundamental question: *Was the system built correctly?* Its primary purpose is to ensure that the developed software components and the integrated system meticulously adhere to their predefined design specifications (as detailed in Chapter 4), architectural plans, and explicitly stated functional and non-functional requirements (Chapter 3). In the context of *Early Retirement*, verification activities involved detailed checks to confirm that:

- Individual C# scripts and modules within the Unity client executed their logic as designed.
- The Python backend server components correctly processed AI inference requests and managed data according to their specifications.
- Data was accurately exchanged between the client and server via the designed WebSocket communication protocol.
- External API integrations, such as those with Deepgram STT and TTS services, performed according to their documented protocols and the integration design.

The aim of verification is the methodical detection and elimination of faults, errors or defects introduced during the design or implementation phases, thereby ensuring the internal consistency, correctness, and technical soundness of the system relative to its blueprint.

6.1.2 Validation

Validation, on the other hand, seeks to answer the question: *Was the right system built?* Its focus is on evaluating the final, integrated product to ascertain whether it successfully meets the intended user needs and fulfills its overarching purpose within the target operational context. For *Early Retirement*, validation activities concentrated on assessing if the core innovation, the AI-driven voice interaction system, provided a demonstrably enhanced, more immersive, or engaging experience for the player compared to traditional NPC interaction paradigms. This included evaluating the perceived quality, relevance, and coherence of the AI-generated dialogue, the player's tolerance for the system's inherent latency, and the overall intuitiveness and effectiveness of the voice-based interaction mechanism. Validation is therefore concerned with the external correctness and utility of the system, ensuring that the final prototype effectively addresses the problem it was designed to solve and aligns with the project's primary objectives regarding player experience [2][8][13]

6.2 Unit Testing

Unit testing formed a critical early phase in ensuring code reliability for *Early Retirement*. Tests were systematically applied to verify the correctness of discrete software units before integration. This approach was followed across both AI components written in Python and game logic developed in Unity using C#.

For AI, Python scripts in the backend were tested independently. Functions handling JSON parsing, prompt construction for SmoLLM2 [5], post-processing (e.g., first-line extraction), and JSON response formatting were validated using mock data. Unit testing tools like `unittest` or `pytest` enabled automation. Key utility functions for data loading, prompt formatting, tokenization with masking, and the custom data collator were tested to ensure correct structure and padding with `-100` labels [6].

In Unity, C# methods were tested in isolation. Audio conversion in `Microphone_Instance.cs` was checked with boundary values. Scripts responsible for JSON serialization, deserialization, and API payload construction were verified using sample strings. Core mechanics like player movement were tested in sandbox scenes within Unity's Play Mode. This bottom-up strategy ensured stable, reliable modules ahead of integration.

6.3 System Testing

System testing evaluated "Early Retirement" as a complete, integrated system within a deployment-like environment. This black-box phase validated system behavior against the functional requirements from Chapter 3, using the final Windows build of the Unity game connected to the internet and the local Python AI server. Testers simulated player behavior including new game starts, navigating the Vata and Fehari cities, and using locomotion and vehicle mechanics, focusing on stability and core functionality.

A major focus was AI-NPC voice interactions, tested across roles (e.g., Police Officer, Shopkeeper) using varied prompts, evaluating the performance of the fine-tuned SmolLM2 model [5][12]. Ambient noise testing was minimal. Player controls, and vehicle mechanics were tested for reliability during and after AI interactions. Informal endurance testing ran the game for extended periods to detect memory leaks, crashes, or performance degradation. This phase served as the final validation against system-wide requirements and assessed system stability before prototype completion.

6.4 Usability Testing

While large-scale usability studies (e.g., SUS surveys) were out of scope, informal testing and heuristic evaluations occurred iteratively. Developers and peer observers simulated users to assess ease of use, intuitiveness, and overall player experience for the core voice interaction feature.

Testing evaluated voice interaction discoverability, activation via key press. Observers noted the 3s average latency between input and response, assessing its effect on conversational flow, a common challenge highlighted in LLM game integrations [8], and whether users repeated speech prematurely. Error recovery handling and user tolerance to nonsensical AI outputs, another known challenge with LLMs [3][8], were also assessed. Despite the latency, users found the voice interface engaging and intuitive, which aligns with positive reception of similar voice-based systems [13]. However, reducing delay was identified as the most critical usability improvement, with AI errors viewed as less immersion-breaking than latency-induced pacing issues, a finding that contributes to understanding player priorities in AI-driven game experiences [2]

6.5 Software Performance Testing

Performance testing focused on responsiveness, resource use, and system stability, particularly the AI-driven voice pipeline. Key metric: end-to-end conversational latency. It includes STT received, AI request/response, TTS request/playback enabled segmented latency analysis. Under good conditions, average latency met the 3s target, demonstrating the real-world efficacy of using optimized small language models [5][4]. Variability occurred under network congestion or high system load.

Unity Profiler and OS tools such as Task Manager monitored CPU, RAM, and network usage. Key observations included CPU load from audio capture via `Microphone_Instance.cs`, networking via `WebsocketClient.cs`, and rendering. RAM use was monitored for leaks. Bandwidth for STT, AI, and TTS was within typical broadband limits. The Python AI server's inference ran stably under single-user load. Performance was optimal when the client was idle; heavy concurrent application use increased latency variability. Testing confirmed feasibility of the 3s latency goal but highlighted variability as a concern for future optimization. The stable inference performance on the server further underscores the benefits of the chosen model and optimization strategy [5, 4].

6.6 Compatibility Testing

Compatibility testing targeted Windows 10 (64-bit), with most testing on multiple machines using this OS. Basic functionality checks on Windows 11 identified no critical incompatibilities, though coverage was limited. Other OS platforms (e.g., macOS, Linux) and consoles were excluded.

Hardware testing covered two PC classes: one meeting minimum specs (Intel Core i5, 8GB RAM, NVIDIA GeForce GTX 1060), and one exceeding recommended specs (Intel Core i7, 32GB RAM, NVIDIA GeForce RTX 3060). On both, AI interaction and gameplay functioned, with smoother experience and more consistent latency on higher-spec hardware. Tests included multiple USB microphone types and 1080p display resolutions. Software prerequisites such as DirectX 12 were validated. Compatibility testing confirmed that the game would function reliably on most target configurations.

6.7 Exception Handling Testing

Exception handling tests assessed system stability under error conditions. Scenarios included internet loss during STT, AI WebSocket, or TTS operations. Network exceptions (e.g., `WebSocketException`, `UnityWebRequestException`) were caught, logged, and handled without crashes. Timeout handling was evaluated.

Invalid JSON payloads tested parsing resilience in both client (C#) and server (Python). Across these tests, error handling mechanisms (try-catch in C#, try-except in Python) effectively maintained application stability. Logs aided debugging, though some user feedback could be improved for clarity in specific failure modes.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

The *Early Retirement* project successfully demonstrated the feasibility of real-time, voice-activated NPC interactions within a prototype open-world game developed in Unity. The system integrated player voice input captured and preprocessed in C#, cloud-based Deepgram STT/TTS services, and a locally hosted Python-based backend running a fine-tuned SmolLM2-360M language model for dialogue generation. Using QLoRA for parameter-efficient fine-tuning and a custom dataset of over 20,000 examples, the project illustrated the practicality of deploying small language models for immersive interaction in constrained development environments [5][4][6].

Key achievements include establishing a complete speech-to-AI response pipeline, achieving average interaction latencies of around 3 seconds—a significant challenge in LLM game integration [8]—and generating contextually appropriate NPC dialogue. Despite some limitations in deep contextual retention and occasional response genericity due to the compact model size, which are common trade-offs with SLMs [5][8], the system delivered coherent interactions and showcased a novel approach to game dialogue. Integration of disparate technologies and asynchronous real-time processing proved both challenging and educational. Overall, the project validates the potential of voice-driven AI systems to enhance immersion in narrative-driven games and sets the groundwork for further innovations in this space.

7.2 Future Work

Future development could focus on several areas to improve performance, realism, and gameplay integration, aligning with ongoing research avenues in AI and gaming [1].

7.2.1 Latency Reduction

Reducing interaction latency is paramount. Future work could leverage edge AI techniques or perform local inference with smaller, optimized models [5] depending on client hardware. Server-side performance may benefit from faster inference frameworks like ONNX Runtime or TensorRT,

along with more efficient asynchronous code. Additionally, switching or optimizing STT/TTS services or exploring on-device voice processing [13] might reduce network-induced delays.

7.2.2 Dialogue Quality and Model Enhancements

To improve dialogue depth and contextual relevance, future iterations might involve larger or more sophisticated models, context windows, or Retrieval-Augmented Generation (RAG) [3][8]. Summarization techniques and persistent context tracking could enhance continuity [3]. With access to better compute, applying Reinforcement Learning from Human Feedback (RLHF) [8] and improving dataset quality [6] would refine both safety and coherence. More advanced prompt engineering could help enforce tone, emotional cues, and character consistency.

7.2.3 Narrative and Gameplay Expansion

Expanding the world with new storylines, side quests, and character arcs would enhance immersion. Game mechanics such as new traversal options (e.g., swimming, bikes), expanded interiors, and NPC behaviors that dynamically respond to dialogue outcomes would make the environment more interactive. A richer narrative that evolves based on AI conversations could further highlight the advantages of this interaction model.

7.2.4 Interaction System Improvements

Caching common phrases and using persistent NPC memory via lightweight databases could improve responsiveness and continuity across sessions [3]. This would support relationship development and long-term player-NPC engagement. These improvements would also help reduce server load for repeated interactions.

7.2.5 Robustness and Evaluation

Improving fault tolerance throughout the STT-AI-TTS pipeline is essential. Future builds should include better error handling and recovery mechanisms. Broader usability testing with diverse players would provide critical insights into responsiveness and satisfaction, particularly for voice interactions [13]. Comprehensive platform testing, especially on various Windows 10/11 configurations, would ensure broader compatibility.

Overall, these enhancements could elevate *Early Retirement* from a prototype into a compelling, real-world demonstration of the future of AI-driven narrative games [1][7].

References

- [1] Markus Dablander. Future research avenues for artificial intelligence in digital gaming: An exploratory report. <https://arxiv.org/abs/2412.14085>, 2024. Accessed December 24, 2024. Cited on pp. 1, 9, 11, 13, 51, and 52.
- [2] Penny Sweetser. Large language models and video games: A preliminary scoping review. <https://arxiv.org/abs/2403.02613>, 2024. Accessed December 24, 2024. Cited on pp. 1, 8, 10, 11, 48, and 49.
- [3] Sihao Hu et al. A survey on large language model-based game agents. <https://arxiv.org/abs/2404.05983>, 2024. Accessed December 24, 2024. Cited on pp. 1, 8, 10, 11, 19, 49, and 52.
- [4] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, May 2023. Extended NeurIPS submission. Cited on pp. 4, 11, 14, 19, 23, 45, 46, 49, and 51.
- [5] Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, TzuHao Mo, Qiuhao Lu, Wanjiang Wang, Rui Li, Junjie Xu, Xianfeng Tang, Qi He, Yao Ma, Ming Huang, and Suhang Wang. A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with llms, and trustworthiness. *arXiv preprint arXiv:2411.03350*, November 2024. Submitted on 4 Nov 2024. Cited on pp. 7, 11, 12, 14, 16, 17, 19, 23, 42, 45, 46, 48, 49, and 51.
- [6] Curtis Savage. Should you train your model on generated data?, April 2023. Published in *AI for PRODUCT PEOPLE* on Medium. Cited on pp. 7, 12, 23, 45, 48, 51, and 52.
- [7] Roberto Gallotta et al. Large language models and games: A survey and roadmap. <https://arxiv.org/abs/2402.18659>, 2024. Accessed December 24, 2024. Cited on pp. 8, 9, 10, 11, 12, 13, 15, 17, 19, and 52.
- [8] Richard Socher, Bryan McCann, David Kiela, and Jacob Devlin. Large language models and games: A survey and roadmap. <https://arxiv.org/>, 2023. Accessed December 24, 2024. Cited on pp. 8, 9, 10, 11, 12, 15, 18, 19, 48, 49, 51, and 52.
- [9] Jesse Dodge, Mohit Iyyer, and Yejin Choi. Llm-deliberation: Evaluating llms with interactive multi-agent negotiation games. <https://montrealetics.ai/>, 2023. Accessed December 24, 2024. Cited on pp. 8, 9, and 10.
- [10] Nathan Herr et al. Are large language models strategic decision makers? a study of performance and bias in two-player non-zero-sum games. <https://arxiv.org/abs/2404.10862>, 2024. Accessed December 24, 2024. Cited on pp. 8 and 10.

- [11] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, May 2023. Accessed: 2024-12-24. Cited on pp. 9 and 10.
- [12] PokéLLMon Team. Pokéllmon: A human-parity agent for pokémon battles with large language models. <https://arxiv.org/>, 2024. Accessed December 24, 2024. Cited on pp. 9, 10, 12, and 49.
- [13] Nima Zargham. Let’s talk games: An expert exploration of speech interaction with npcs. <https://www.tandfonline.com/doi/full/10.1080/10447318.2024.2338666#abstract>, 2024. Accessed April 19, 2025. Cited on pp. 10, 11, 12, 13, 17, 48, 49, and 52.
- [14] Li Song. Llm-driven npcs: Cross-platform dialogue system for games and social platforms. <https://arxiv.org/abs/2504.13928>, 2025. Accessed April 14, 2025. Cited on pp. 10, 11, and 13.
- [15] Takehiro Takayanagi, Hiroya Takamura, Kiyoshi Izumi, and Chung-Chi Chen. Beyond turing test: Can gpt-4 sway experts’ decisions? *arXiv preprint arXiv:2409.16710*, September 2024. Version 2, submitted on 25 Nov 2024. Cited on pp. 10 and 11.
- [16] Meta AI. Cicero: An ai agent that negotiates, persuades, and cooperates with people, November 2022. Meta AI Blog. Cited on p. 10.
- [17] Chengpeng Hu, Yunlong Zhao, and Jialin Liu. Game generation via large language models. <https://ieeexplore.ieee.org/document/10645597>, 2024. Accessed December 24, 2024. Cited on p. 11.