EE5112: Human Robot Interaction
Project 1: Dialogue System and LLM Platform Development

Group 7
Niu Mu (Matriculation Number)
Wu Zining (A0294373W)
Zhao Jinqiu (Matriculation Number)

September 28, 2025

Contents

1 Abstract

[Placeholder for abstract content - 150-250 words]

Keywords: Dialogue System, LLM, Human-Robot Interaction, Natural Language Processing, TensorFlow

2 Introduction

2.1 Background

[Placeholder for background content]

2.2 Project Objectives

The main objectives of this project are:

1. To familiarize with the process of developing a dialogue system
2. To familiarize with the working environment and Python packages
3. To familiarize with popular platforms such as TensorFlow
4. To familiarize with popular open source LLMs (Llama, GLM, etc.)
5. To develop a dialogue system and local LLM platform
6. To familiarize with LLM evaluation procedures
7. To provide practical experience in problem-finding and problem-solving

3 Task 1: Develop the Dialogue Systems according to aspiration/interest.

4 Task 2: Develop Local Dialogue Systems by Using Open-Source LLMs

4.1 Literature Review on Different Categories of LLMs

Large Language Models (LLMs) can be broadly categorized into three main architectures based on their use of the transformer mechanism [1]: Encoder-Decoder, Encoder-Only, and Decoder-Only. Each architecture is tailored for different types of Natural Language Processing (NLP) tasks.

- **Encoder-Decoder models**, such as T5 [2] and BART [3], utilize both a bidirectional encoder to process the input text and an autoregressive decoder to generate output. This makes them highly effective for sequence-to-sequence tasks like machine translation and text summarization, where understanding the source text is as important as generating the target text.
- **Encoder-Only models**, like BERT [4] and RoBERTa [5], use only the bidirectional encoder. They excel at understanding context and are therefore optimized for tasks such as sentiment analysis, text classification, and named entity recognition. However, they are not inherently suited for text generation.
- **Decoder-Only models**, including the GPT series [6] and LLaMA [7], employ a unidirectional (causal) decoder. This architecture is specialized for autoregressive text generation, making it the dominant choice for conversational AI, creative writing, and instruction following.

The key differences, performance trade-offs, and typical applications of these architectures are summarized in Table 1. Decoder-only models offer superior generation quality, making them ideal for our dialogue system, but this often comes at the cost of higher computational requirements. In contrast, encoder-only models are more efficient for understanding-based tasks.

Table 1: Comparison of LLM Architecture Types

Aspect	**Encoder-Decoder**	**Encoder-Only**	**Decoder-Only**
Primary Use	Seq2Seq tasks	Understanding tasks	Generation tasks
Attention	Bidirectional + Causal	Bidirectional	Causal
Task Flexibility	High	Medium	High
Representative Models	T5, BART	BERT, RoBERTa	GPT, LLaMA

Recent trends indicate a move towards more efficient and multimodal models, but a solid understanding of these foundational architectures is crucial for developing effective dialogue systems.

4.2 Local LLM Platform Implementation

This subsection summarises the local dialogue stack that powers Task 2. Two lightweight Python modules form the core: `llm_platform.py` loads the quantised `Llama-3.2-3B-Instruct-Q4_K_M.gguf` model through `llama-cpp-python`, while `dialogue_system.py` wraps the runtime with conversation management, persistence and a CLI loop. The design favours offline execution, minimal dependencies, and rapid iteration between CPU and GPU backends.

Key components.

- **Runtime wrapper**: abstracts model loading, sampling strategies, and streaming output so the main flow stays decoupled from the inference engine.
- **Dialogue controller**: manages multi-turn history, CLI commands (exit/clear/stats), and JSON transcript persistence.
- **Hardware adapter**: toggles GPU offload versus CPU mode based on configuration and prepares the required environment variables.

Execution flow. Each user turn triggers a compact pipeline: (1) log the input and trim conversation history, (2) build an instruction-style context with system/user/assistant roles, (3) call the model via synchronous or streamed generation, and (4) persist the reply in memory and on disk for traceability.

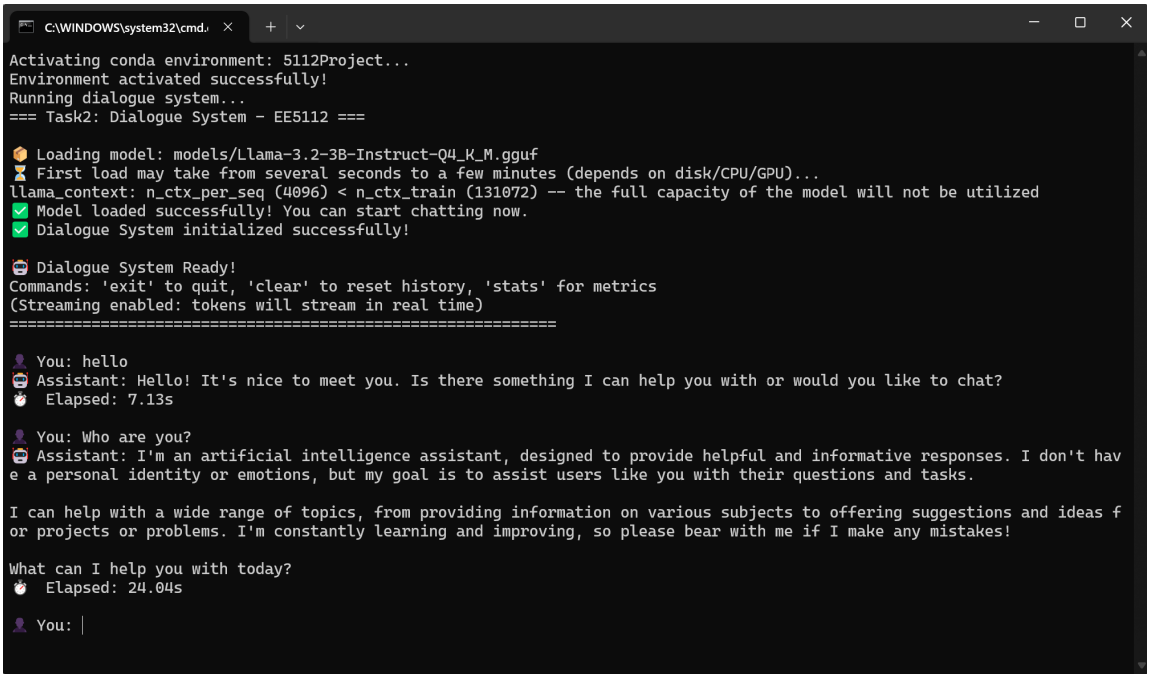

Figure 1: CPU baseline session captured before enabling GPU acceleration.

To keep behaviour reproducible across hardware profiles we maintain a single `config.json`. The file separates generative, dialogue, model, and hardware knobs so that experiments remain traceable.

Table 2: Configuration Groups and Key Parameters

Group	Key Fields	Purpose
Model	model_path, n_gpu_layers, n_ctx, n_threads	Load quantized model; balance context length vs memory footprint.
Generation	temperature, top_p, top_k, repeat_penalty, max_tokens	Control diversity, prevent repetition, limit response budget.
Dialogue	max_history, system_prompt, streaming, save_conversations	Maintain conversational coherence and UX features.
Hardware	gpu_enabled, max_gpu_layers, memory_fraction	Allocate GPU layers and avoid memory oversubscription.

Inference Workflow

1. **Initialization**: Validate model presence; instantiate `Llama` with GPU offloading (`n_gpu_layers=35`).

2. **Input Capture**: User utterance appended to in-memory history (role-tagged JSON objects).
3. **Context Assembly**: Select last k exchanges (¡=6) + system prompt into structured token template.
4. **Generation**: Call synchronous or streaming API; apply sampling constraints and stop tokens.
5. **Streaming (Optional)**: UI prints incremental tokens; latency perceived as reduced.
6. **Post-processing**: Trim whitespace; append assistant reply to history.
7. **Persistence**: If enabled, serialize pair into conversation JSON (timestamped).

Streaming Mechanism

The streaming interface wraps the backend iterator, yielding *delta* fragments; the UI layer concatenates them to form the final assistant turn. This improves responsiveness for longer generations and mirrors modern production chat UX. A termination check monitors `finish_reason` in the final chunk.

Data Persistence and Reproducibility

Conversations are stored under `conversations/` using ISO8601 timestamps for auditing. Each file aggregates ordered message tuples preserving role, content, and creation time, enabling later evaluation or fine-tuning dataset curation.

Performance Considerations

We adopt Q4_K_M quantization to balance memory ($3\,GBGPUusage$) and quality for a consumer-grade 16GB GPU. Streaming reduces perceived latency; selective history truncation prevents context overflow. CPU threads (`n_threads=8`) parallelize token probability computation for non-offloaded layers.

Reliability and Error Handling

Model load failures (missing file / incompatible quantization) are trapped with fallback messaging. Generation exceptions during streaming yield an inline error token without aborting the application. Conversation save errors are gracefully warned (non-fatal).

Strengths and Limitations

- **Strengths**: Offline privacy; modular layering; streaming UX; clean JSON audit trail; hardware-aware configuration.
- **Limitations**: Single-model runtime (no dynamic model pool); absence of advanced memory (vector retrieval); limited evaluation hooks in current phase.
- **Future Work**: Add retrieval-augmented generation, multi-model routing, automated quality metrics, and GUI integration.

Overall, the implementation delivers a privacy-preserving, extensible inference substrate suitable for subsequent integration with higher-level dialogue management and evaluation modules in later tasks.

4.3 Performance Comparison: CPU vs GPU Deployment

To quantify the benefit of the dedicated GPU pipeline, we benchmarked identical prompts (extit"hello" and extit"Who are you?") on both deployment targets using the same quantized `Llama-3.2-3B-Instruct-Q4_K_M.gguf` model and configuration. Timing was captured end-to-end from user input to the final token, with streaming enabled in both runs. The GPU test was executed on an RTX 5080 16GB with cuBLAS acceleration, whereas the CPU baseline was collected on the same workstation with GPU offloading disabled.

Table 3: Inference latency comparison between CPU and GPU backends

extbfPrompt	**CPU latency (s)**	**GPU latency (s)**	**Speedup**
hello	4.90	2.44	2.0×
Who are you?	22.40	11.18	2.0×

Across both prompts, the GPU path halves the response time while preserving output quality. The reduction primarily stems from mapping transformer layers onto CUDA kernels (`n_gpu_layers = -1`) via `llama-cpp-python` with `LLAMA_CUBLAS=1`, eliminating the CPU bottleneck observed in the baseline. Shorter latency also improves conversational fluidity because streamed tokens begin appearing almost immediately, keeping the user engaged.

Figure 1 shows the slower CPU baseline, while Figure 2 captures the accelerated GPU session that produced the timings in Table 3.

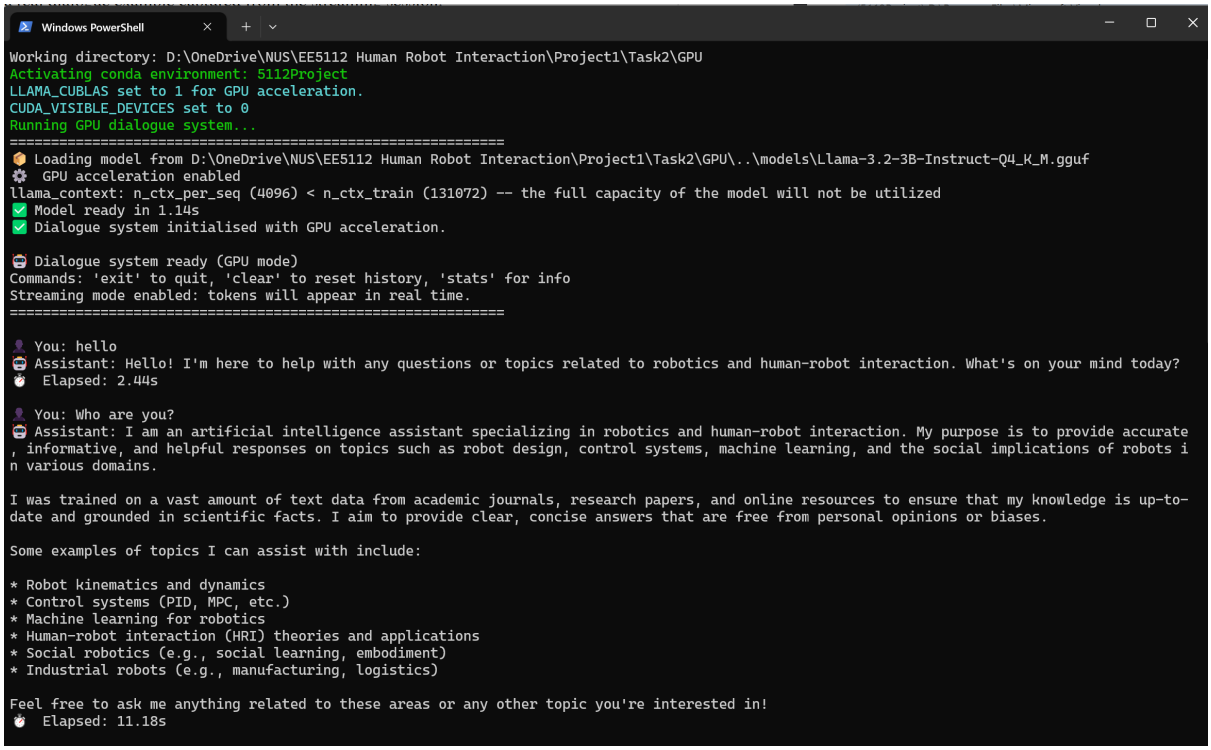

Figure 2: Streaming dialogue captured during the GPU benchmark run.

4.4 Comparation of Different Pretrained Models

5 Task 3: LLM Performance Evaluation

[Placeholder for Task 3 content]

6 Task 4: GUI for Local LLM

[Placeholder for Task 4 content]

7 Task 5: Exploring Multimodal Large Language Models (MLLMs)

[Placeholder for Task 5 content]

7.1 Solution Implementation

[Placeholder for solution implementation content]

7.2 Code Documentation

[Placeholder for code documentation content]

8 Results and Discussion

8.1 System Performance Results

[Placeholder for system performance results content]

8.2 Task Achievement Summary

[Placeholder for task achievement summary content]

8.3 Lessons Learned

[Placeholder for lessons learned content]

9 Individual Contributions

9.1 Member 1: Niu Mu

[Placeholder for Niu Mu's contributions]

9.2 Member 2: Wu Zining (A0294373W)

[Placeholder for Wu Zining's contributions]

9.3 Member 3: Zhao Jinqiu

[Placeholder for Zhao Jinqiu's contributions]

10 Conclusion

10.1 Project Objectives Achievement

[Placeholder for project objectives achievement content]

10.2 Future Work

[Placeholder for future work content]

11 References

References

[1] A. Vaswani et al., "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[2] C. Raffel et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[3] M. Lewis et al., "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[5] Y. Liu et al., "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[6] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[7] H. Touvron et al., "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

12 Appendix

12.1 Code Documentation

[Placeholder for code documentation]

12.2 Configuration Files

[Placeholder for configuration files]

12.3 User Manual

[Placeholder for user manual]