Conversational AI Module: Lab Handout

Songbo Hu[1] and Malak Sadek[2]

[1] Language Technology Lab, University of Cambridge
sh2091@cam.ac.uk
[2] Centre for Human Inspired Artificial Intelligence, University of Cambridge
mfzas2@cam.ac.uk

Abstract. This handout supports the lab sessions and course assignment for the *Conversational AI module* of the MPhil in Human-Inspired Artificial Intelligence, cohort 2025. The first two practical lab sessions will take place on the 13th and 20th, 1–2 PM, in the Syndics Room (17 Mill Lane). Its purpose is to guide you through the practical implementation of LLM-based conversational agents and to help you prepare for the course assignment.
Office hours are held every Friday, 9 AM–12 PM, in TR-17, English Faculty Building. You are welcome to drop in with questions about the lab, the assignment, or broader topics in NLP research.
For any further queries about this handout, please contact Songbo Hu at sh2091@cam.ac.uk.

1 Conversational Agents in a Nutshell

Conversational agents—also known as dialogue systems—have been central to research in natural language processing (NLP) and artificial intelligence (AI), dating back to Alan Turing's proposal of the imitation game [1]. Early examples include rule-based systems such as ELIZA [2], while today's systems are often powered by large language models (LLMs), for example ChatGPT [3]. In this handout, I will use the term *dialogue system*, which is the more common term in NLP research.

Broadly defined, a dialogue system is a computer program designed to maintain an intelligent, real-time conversation with a human. These systems are now so pervasive in daily life that their definition is often taken for granted rather than explicitly stated.

Dialogue systems are increasingly important in real-world applications, especially in domains such as healthcare and education. They serve as conversational interfaces that make language technologies accessible to non-experts. In practice, much of the impact of LLMs has been realised through dialogue systems, which translate research advances into tools that ordinary users can interact with directly.

In this lab, you will build a conversational system powered by state-of-the-art LLMs. We will begin with a simple but functional baseline, and then extend it through small research experiments.

2 Dialogue Systems: Concepts and Evaluation

In this section, we introduce the core concepts and NLP techniques behind dialogue systems. We do this by providing a brief historical overview that shows how advances in NLP have shaped system design, while at the same time highlighting the key concepts that remain relevant for building and understanding dialogue systems today. Then, we turn to evaluation methods, which are essential for analysing and comparing your own systems in the lab.

In NLP research, dialogue systems are usually grouped into two broad categories: **task-oriented dialogue systems** and **open-domain dialogue systems**. Task-oriented systems are designed to help users achieve specific goals (e.g., booking a train ticket or checking the weather). Open-domain systems—often called **chatbots**—aim to sustain longer, more free-form conversations that resemble human dialogue [4, Chapter 15]. Put simply, task-oriented systems are built to be *helpful*, while open-domain chatbots are built to be *engaging*.

Traditionally, task-oriented systems relied on structured representations and external databases or APIs to fetch information and perform actions, which made them quite different from open-domain chatbots. Since the advent of LLMs, this distinction has blurred. LLMs can follow natural-language instructions and even produce structured outputs (e.g., JSON) directly, reducing the need for specialised modules and making task-oriented systems much closer in design to open-domain ones.

In the next part, we will review how dialogue systems evolved—from early rule-based methods to today's LLM-based architectures—and introduce the core concepts that explain how each type of system works.

2.1 From Rules to LLMs

This section introduces the major paradigms in dialogue system development, showing how design choices have evolved alongside advances in NLP. At each stage, we highlight the key concepts that will be useful for your own lab work.

Rule-based systems. Rule-based dialogue systems represent the earliest paradigm in conversational AI. The most famous example is **ELIZA** [2], developed in the 1960s to simulate a Rogerian psychotherapist. These systems operate by matching user inputs against a set of hand-crafted rules, typically expressed as *patterns* (for recognising input) and *templates* (for generating responses).

For example, ELIZA might include a rule like this:

Pattern: `(0 YOU 0 ME)`
Transform: `WHAT MAKES YOU THINK I 3 YOU`

User input: `You love me`
System output: `WHAT MAKES YOU THINK I LOVE YOU`

Here, '0' denotes a wildcard matching any sequence of words, and the number '3' in the transform refers to the second wildcard segment in the user input.[3] Based on the above rule, ELIZA identifies key patterns (such as 'you' and 'me') in the user's input and applies a rule-based transformation to rephrase the statement as a system response.

Strengths: Rule-based systems are fully interpretable and controllable: developers know exactly how the system will respond. This makes them useful in domains where safety and predictability are important.

Limitations: Because every rule must be written by hand, these systems scale poorly to new domains or languages. They are also brittle: small changes in wording can break the match, leading to unnatural or repetitive conversations.

Retrieval-based systems. Retrieval-based dialogue systems are widely seen as the successors to rule-based systems. Rather than generating responses from scratch, they search within a fixed set of human-authored candidate responses and select the one that best matches the user's input. This marks a key shift from hand-written rules to data-driven methods.

How it works. The system encodes both the dialogue context (the user's query) and each candidate response into embedding vectors, and then selects the response with the highest similarity. Early systems relied on keyword overlap (e.g., TF–IDF or BM25), while modern systems use neural encoders such as `BERT` [5].

Language Encoder: A neural model that converts input text into a vector representation.
Embedding: The resulting continuous vector (often low-dimensional), where semantically similar texts are mapped to nearby points in the space.

In the following, we provide an illustrative example of how a retrieval-based system works. Here, both the user input and the candidate responses are encoded into embeddings, and the system ranks them by cosine similarity.

User input:

– `What are the best places to visit in Cambridge during the summer?`

Ranked candidate responses (cosine similarity:)

1. `What are the best places to see in Cambridge during the summer?`
   (paraphrase; $similarity = 0.99$)

[3] The number '2' in the transform would refer to the word 'you' in this example.

2. `You might enjoy punting on the River Cam or relaxing in the Botanic Garden.`
   (appropriate; $similarity = 0.90$)
3. `There is a nice restaurant in Oxford.`
   (irrelevant; $similarity = -0.40$)

Although the paraphrase is closest in wording, the second option is clearly the most appropriate response. This example illustrates why retrieval-based systems often require dialogue-specific encoders: they must capture not only surface similarity, but also conversational relevance. In practice, this usually means fine-tuning an encoder on your own dataset so that it learns what counts as a good response in your particular domain.

Strengths. Retrieval-based systems are safer than generation-based ones, since they can only output predefined responses. They are also easier to scale compared to hand-written rules.

Limitations. They cannot generate novel answers beyond the candidate set, and may sound repetitive or generic if the response pool is small. In modern systems, retrieval is often combined with generation.

Generation-based systems. A major shift in dialogue system development has been the move from *selecting* predefined responses to *generating* new ones with neural language models. This generation-based paradigm began with sequence-to-sequence (Seq2Seq) models, was advanced by pre-trained language models (PLMs), and today is dominated by large language models (LLMs). Each stage reflects the adoption of larger and more powerful neural models, leading to greater capability.

In this course, you will build a simple generation-based dialogue system as your baseline, and then extend it through research experiments.

Seq2Seq-based. Seq2Seq [6] models are a type of neural language model that generate responses token by token, conditioned on the user's input and all previously generated words. This marked the first major step toward fully generative dialogue systems. However, because these models had to be trained from scratch on relatively small datasets, their fluency and generalisation ability were limited.

Language Model: A machine learning model that assigns probabilities to sequences of words or tokens. Traditional language models (e.g., n-gram models) rely on statistical counts, while neural language models use neural networks to learn these probabilities from data.

Autoregressive Language Model: A type of language model that generates text one token at a time, predicting each new token based on

the dialogue context X and all previously generated tokens:

$$P(Y \mid X) = \prod_{t=1}^{T} P(y_t \mid X, y_{<t})$$

PLM-based. The next leap came with PLMs such as BERT, GPT-2 [7], and T5 [8], which are first trained on massive text corpora and then fine-tuned on smaller dialogue datasets. This separation of *pre-training* and *fine-tuning* allowed models to encode broad linguistic knowledge and adapt to new domains more effectively.

Pre-trained Language Model (PLM): A neural language model trained on very large unlabelled corpora, which captures general syntactic, semantic, and world knowledge. It can then be fine-tuned on smaller, task-specific datasets to adapt to dialogue applications.

LLM-based. Modern dialogue systems are powered by large language models (LLMs) such as GPT-4, Claude, and LLaMA. These models demonstrate *emergent abilities*—notably **instruction following** and **in-context learning (ICL)**—which allow them to perform new tasks without task-specific training. Instead of fine-tuning, developers now provide instructions and examples directly at inference time through prompts.

Large Language Model (LLM): A pre-trained language model trained on an extensive amount of data, capable of generalising to a wide range of tasks with minimal or no task-specific fine-tuning, and exhibiting emergent abilities.

Instruction Following: The ability of LLMs to interpret and execute a wide range of user instructions, typically without requiring additional fine-tuning. These instructions are expressed as natural language prompts.

In-Context Learning (ICL): The ability of LLMs to perform new tasks by conditioning on a small number of examples given in the prompt at inference time, without updating model parameters.

Prompt: The text input provided to an LLM, which specifies the task, context, or examples. Prompts can be as simple as a direct instruction (e.g., `‘‘Write me a song’’`) or a more elaborate input with examples.

Prompt Engineering: The practice of designing and refining prompts to elicit desired behaviour from an LLM. This can involve rephrasing instructions, adding examples, or structuring the input in ways that improve output quality.

A key limitation of LLMs is their restricted **context window**, which limits how much information can be included in the prompt. To address this, many systems adopt **retrieval-augmented generation (RAG)**, where external knowledge is dynamically retrieved and injected into the prompt to ground the model's responses. For example:

User input:

– `What are the best places to visit in Cambridge during the summer?`

Retrieval step:
A dense retrieval model retrieves the most relevant entries from a knowledge base (e.g., a database of tourism information). The top retrieved entries are:

1. `Punting on the River Cam is a classic Cambridge summer activity.`
2. `The Cambridge University Botanic Garden offers beautiful grounds in the summer.`
3. `King's College Chapel is a popular attraction in Cambridge.`

Generation step:
The LLM then conditions its response on both the user input and the retrieved entries, producing an answer such as:

– `Some of the best places to visit in Cambridge during the summer include punting on the River Cam, visiting the Botanic Garden, and exploring King's College Chapel.`

Semantic parsing for action. In many real-world applications, dialogue systems need not only to *converse*, but also to *act*—for example, booking a table, setting an alarm, or checking a schedule. Since computers cannot directly execute free-form language, the request must be expressed as a structured query. Modern LLMs can generate such structured outputs directly when guided with a suitable prompt.

User: `Set an alarm for 6 am tomorrow.`

Prompt to LLM: `Extract the action from the user's request and output it as a JSON object. Use the format: { "action": ..., "time": ... }`

LLM Output: `{ "action": "set_alarm", "time": "06:00" }`

Here, the LLM is not only responding in natural language but also producing a structured query that a computer program can execute. This step is called **semantic parsing**.

In modern dialogue systems, semantic parsing is the step that turns "just talking" into useful actions, by transforming natural language into structured, machine-readable queries that can be sent to APIs or databases. With the right prompt (and sometimes a few examples), an LLM can produce these structured outputs, enabling the agent to do things like set alarms, check schedules, or book services—not just chat.

System architecture. Figure 1 illustrates how semantic parsing fits into a full dialogue pipeline. The user's spoken input is first transcribed by an automatic speech recognition (ASR) module. The text is then passed to the semantic parser, which produces a structured query that can be executed against a database or API. The result of this query, together with the user's original input, is used by the response generation module to create a natural-language reply. Finally, a text-to-speech (TTS) module can convert this reply back into speech. In this course, we will only work with text-based dialogue. The speech components (ASR and TTS) are included to illustrate how the same pipeline extends to spoken dialogue systems. You do not need to implement speech modules for the lab or assignment. You will only work with the text-based part of this pipeline.

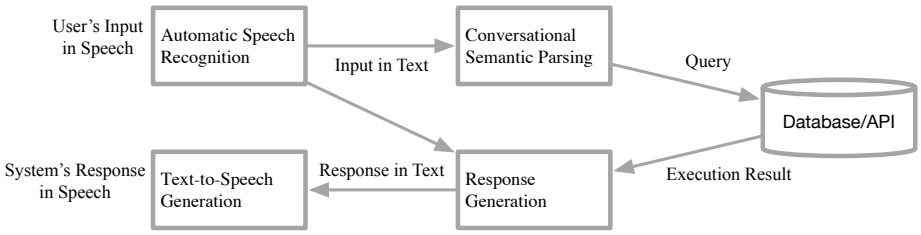

Fig. 1. System architecture for a spoken dialogue system.

This pipeline highlights how dialogue systems combine *conversation* with *action*: the semantic parsing step connects human language to machine operations, allowing the agent not only to respond, but also to perform useful tasks in the real world.

Strengths. Generation-based systems can produce novel and contextually appropriate responses, rather than being restricted to a fixed set of options. They are more flexible than retrieval-based approaches, and with LLMs, they can follow natural-language instructions, adapt to new tasks through prompts, and handle a wide variety of applications without task-specific training.

Limitations. These systems are less predictable than rule- or retrieval-based ones, and may sometimes generate incorrect, irrelevant, or unsafe responses (i.e.,

the issue of hallucination). Their effectiveness depends heavily on the availability and quality of training data, as well as on prompt design. Moreover, LLMs require substantial computational resources and operate as largely "black boxes", which makes them costly to run and difficult to control or explain.

2.2 Evaluating Dialogue Systems

Building a prototype dialogue system is only half the story: we also need to know *how good it is* and **whether it is ready for real-world use**. Evaluation plays a central role in NLP research, as it allows us to compare systems, refine design choices, and justify whether a proposed method is truly effective. In this section, we introduce the main approaches to evaluating dialogue systems and discuss how they can be applied in practice. Broadly, evaluation methods fall into two categories:

Automatic metrics. These are fast, cheap, and consistent. They are widely used for system components where outputs are structured (e.g., intent classification, slot filling, dialogue state tracking). Common metrics include **accuracy**, **precision/recall/F1**, or **exact match**.

For response generation, automatic metrics typically compare system outputs against reference responses (**reference-based evaluation**). The most widely used are **BLEU** [9], **ROUGE** [10], and **METEOR** [11], which measure n-gram overlap. While these metrics are useful for quick benchmarking and reproducibility, they often fail to capture whether a response is genuinely appropriate or engaging, because dialogue naturally allows for many valid continuations (*the one-to-many problem*). To address this limitation, recent research explores **reference-free metrics**, including approaches that use large language models themselves as evaluators (*LLM-as-a-judge*).

Human evaluation. In dialogue research, **human evaluation** is widely considered the gold standard, because only humans can judge the *holistic qualities* of a conversation. However, this approach comes with clear limitations: (i) Human judgments are often inconsistent—different people may rate the same output differently, which reduces reproducibility. (ii) Human evaluation is expensive and too slow for rapid system development. (iii) It requires careful task design, domain expertise, and quality control to ensure reliability. Despite these challenges, well-designed protocols (e.g., clear rating guidelines, multiple annotators) can improve consistency and make human evaluation more reliable.

One key reason for subjectivity and inconsistency in human evaluation is that dialogue quality is inherently multi-faceted. There is no universal definition of good dialogue, and the criteria often depend on the application, domain, and even individual users. To mitigate this, researchers often decompose dialogue quality into multiple fine-grained dimensions. This not only reduces variability in human judgments but also provides concrete guidelines for system development and analysis.

Dimension	Definition
Turn-Level Evaluation	
Grammaticality	Responses are free of grammatical and semantic errors
Relevance	Responses are on-topic with the immediate dialogue history
Informativeness	Responses provide unique and non-generic information specific to the dialogue context
Emotional Understanding	Responses demonstrate awareness of the user's current emotional state and respond appropriately
Engagingness	Responses are engaging and help fulfil the user's conversational goals
Consistency	Responses do not contradict information previously provided by the system
Proactivity	Responses actively and appropriately move the conversation forward
Quality	Overall quality and satisfaction with the individual response
Dialogue-Level Evaluation	
Coherence	The system maintains a logical and consistent flow throughout the dialogue
Error Recovery	The system is able to recover from its own errors during the conversation
Consistency	The system is consistent in the information it provides across the dialogue
Diversity	The system provides a diverse range of responses throughout the conversation
Topic Depth	The system discusses topics in depth, rather than superficially
Likeability	The system displays a likeable and engaging personality over multiple turns
Understanding	The system demonstrates sustained understanding of the user
Informativeness	The system provides unique and non-generic information throughout the dialogue
Adaptability	The system adapts flexibly to the user and their interests
Overall Impression	Overall quality and user satisfaction with the entire dialogue

Table 1. Evaluation dimensions for dialogue systems at both the turn and dialogue levels. Adapted from [12, 13].

These evaluation dimensions can be considered at two levels: (i) Turn level, where we judge the quality of individual responses. (ii) Dialogue level, where we judge the quality of the conversation as a whole. Table 1 summarises commonly used dimensions at both levels. In your own projects, you may not need all of them, but they can serve as a useful reference when designing human evaluation protocols. For the assignment, you should try at least one automatic metric and one small-scale human evaluation, so you can see how the two complement each other.

3 Implementing a Dialogue System: An Assignment Walkthrough

This section provides a walkthrough of building a baseline dialogue system for the course assignment. We show an example implementation in Python as a reference. You are welcome to follow it step by step, but you are not required to do so: you are free to design and implement your system in your own way.

Rather than focusing on system design choices (which are covered in the lecture), this section walks you through the engineering steps of turning a design into a working prototype. We begin with a simple baseline system and then illustrate how it can be extended with features such as RAG or semantic parsing. Along the way, you will see code snippets and explanations of key implementation details. You are not required to use every technique introduced here—treat them as building blocks, like Lego, and select the ones that best support your assignment.

All code examples presented in this handout are available at: `https://github.com/songbohu/conversational_AI_module`

3.1 Prerequisites

To follow this walkthrough, you should be comfortable with at least one mainstream programming language, ideally **Python**, as all examples in this handout are written in it. You will also need access to a personal computer (a laptop or workstation is sufficient; no GPU is required).

In addition, you will receive an **OpenAI API key**, which is required to access the state-of-the-art GPT models provided by OpenAI. A typical API key looks like this:

```
sk-proj-xxxxxx-xxxxxxxxxxx
```

Important: Treat your API key as confidential information. Think of it as your **bank account and password**—anyone with access to it can use your credits. Do *not* share it with anyone, post it online (e.g., GitHub or public notebooks), or include it in your assignment submission.

For the assignment, the recommended languages are Python. If you intend to use a different language (e.g., JAVA or C++), please contact me in advance to ensure compatibility with grading and testing.

For most students, we recommend using **Jupyter Notebook** for experimentation and **conda** to manage dependencies. If you are new to these tools or run into setup issues, please come to my office hours or contact me via email at `sh2091@cam.ac.uk`.

3.2 Building a Skeleton for Dialogue Systems

In this section, we will construct the simplest possible skeleton of a dialogue system. For illustration, we call this prototype the **AI Parrot**. As the name suggests, this system is not designed to generate novel responses; it performs only one task: repeating back what has been said to it. Despite its simplicity, the AI Parrot contains all the essential components required for your assignment—except for the use of LLMs.

The code in Listing 1.1 defines an abstract base class, `DialogueSystem`, which serves as the foundation for all kinds of conversational agents in this handout. It provides the following functionalities:

- **Conversation management:** The system maintains a `conversation_history`, where each turn (user or assistant) is stored with its timestamp and optional metadata.

- **Extensibility:** The class defines an abstract method `chat()`, which must be implemented by subclasses. This design allows different dialogue systems (e.g., an LLM-based system) to reuse the same infrastructure while defining their own logic for generating responses.
- **Interaction loop:** The `start_a_chat()` method provides a simple text-based interface for multi-turn interaction. It handles user input, response display, and session termination.
- **Structured logging:** All dialogues can be saved automatically as JSON files using `save_history()`, ensuring reproducibility for evaluation and debugging.

```python
from abc import ABC, abstractmethod
from datetime import datetime
import json
import os

class DialogueSystem(ABC):
    """
    Abstract base class for dialogue systems.
    Handles essential logics for dialogue systems.
    """

    def __init__(self):
        self.reset()

    def reset(self):
        """Reset dialogue state and history."""
        self.conversation_history = []

    def append_turn(self, speaker: str, utterance: str, meta: dict = None
        ):
        """Add a turn to the dialogue history, with timestamp and optional
             metadata."""
        timestamp = datetime.now().strftime("%H:%M:%S") # e.g. "10:20:02"
        turn = {
            "timestamp": timestamp,
            "speaker": speaker,
            "utterance": utterance
        }
        if meta:
            turn["meta"] = meta
        self.conversation_history.append(turn)

    @abstractmethod
    def chat(self, utterance: str) -> dict:
        """
        Process a user utterance and return a structured result.
        Must include at least a 'text' field for the assistant reply.
        Example return:
```

```
        {
            "text": "Here is the weather forecast for tomorrow.",
            "action": {"type": "query_weather", "location": "Cambridge"}
        }
        """
        pass

    def start_a_chat(self):
        print("The system is ready. Type ‘bye‘ or ‘exit‘ to end the
            conversation.\n")
        while True:
            user_input = input("User: ")
            self.append_turn("user", user_input)

            if user_input.lower() in ["exit", "bye"]:
                print("Bot: Goodbye!")
                self.save_history()
                break

            result = self.chat(user_input)

            # Display assistant text
            print(f"\nBot: {result['text']}\n")

            # Append structured result
            self.append_turn("assistant", result["text"], meta={k: v for k
                , v in result.items() if k != "text"})

    def get_history(self):
        return self.conversation_history

    def save_history(self, filepath=None):
        """
        Save the current conversation history to a JSON file.
        If no path is given, automatically generate one with timestamp.
        """
        if filepath is None:
            os.makedirs("logs", exist_ok=True)
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            filepath = os.path.join("logs", f"conversation_{timestamp}.
                json")

        with open(filepath, "w", encoding="utf-8") as f:
            json.dump(self.conversation_history, f, indent=4, ensure_ascii
                =False)

        print(f"Conversation saved to {filepath}")
        return filepath
```

Listing 1.1. Implementation of the `DialogueSystem` abstract base class, which defines the essential structure and logic shared by all dialogue systems. It handles conversation history management, structured turn logging with timestamps, and file-based conversation saving.

Listing 1.2 illustrates how the `DialogueSystem` base class can be extended to implement a minimal working dialogue agent. The `ParrotBot` inherits all conversation management and logging functionalities from the base class, and only needs to define the `chat()` method, which specifies how the system generates responses. In this simple example, the bot simply echoes the user's utterance, acting as a "parrot" that repeats whatever it hears.

```python
from dialogue_system import DialogueSystem

class ParrotBot(DialogueSystem):

    def chat(self, utterance: str) -> dict:
        return {
            "text": utterance,
            "action": {"type": "echo"}
        }

if __name__ == "__main__":
    bot = ParrotBot()
    bot.start_a_chat()
```

Listing 1.2. Implementation of the `ParrotBot`.

3.3 Batch Replay for Reproducible Evaluation

So far, we have interacted with the `ParrotBot` manually. However, for research and evaluation purposes, it is important to have a **reproducible testing setup**. Instead of chatting with the bot in real time, we can evaluate it on a fixed set of pre-defined dialogues. This ensures that different system configurations or model versions can be compared *fairly and consistently*.

To achieve this, we prepare a small dataset of example dialogues, each represented as a list of turns. This dataset is provided purely for illustration in this handout.

> In your assignment, you will need to create your own testing dialogues based on the specific use case you choose to work on. In typical NLP research, evaluation datasets may contain hundreds or even thousands of dialogues. However, for this assignment, it is sufficient to prepare a smaller test set of around **10–50 dialogues**.

Every turn is a JSON object specifying the `speaker` (`user` or `assistant`) and their corresponding `utterance`. A sample input file (`example_input.json`) is shown below, where only one dialogue is displayed for illustration:

```
[
    [
    {"speaker": "user", "utterance": "Do colleges compete with each other ?"},
    {"speaker": "assistant", "utterance": "Yes, especially in sports like rowing. The May Bumps and Lent Bumps are famous races."},
    {"speaker": "user", "utterance": "Rowing sounds fun! Which college is the best at it?"},
    {"speaker": "assistant", "utterance": "It changes every year, but St John's, Caius, and Trinity are often at the top."}
    ],
...
]
```

Listing 1.3. Example input dialogues for batch replay testing (`example_input.json`).

Each inner list corresponds to one dialogue, and each element records a single conversational turn. The `user` turns will be fed into the system during batch replay, while the `assistant` turns serve as the **ground truth responses** for comparison.

To enable **reproducible evaluation**, we use a batch replay script that automatically runs the dialogue system on a fixed set of input dialogues and records both the system outputs and the ground-truth responses. Listing 1.4 shows an example implementation of such a script, which can be used to benchmark different system configurations under identical conditions.

```
import json
from datetime import datetime
import os
from parrot_bot import ParrotBot


def batch_replay(input_file: str, output_file: str = None):
    """
    Run a batch of multi-turn dialogues through the dialogue system.
    Keeps both system-generated responses and ground-truth assistant utterances.
    """

    # Load test dialogues
    with open(input_file, "r", encoding="utf-8") as f:
        test_dialogues = json.load(f)

    print(f"Running batch replay with {len(test_dialogues)} dialogues...\n")

```

```
    bot = ParrotBot()
    all_results = []

    for d_idx, dialogue in enumerate(test_dialogues, start=1):
        print(f"--- Start of Dialogue {d_idx} ---\n")
        bot.reset()
        dialogue_result = []

        for turn_idx, turn in enumerate(dialogue, start=1):
            speaker = turn["speaker"]
            utterance = turn["utterance"]

            if speaker.lower() == "user":
                # Record user utterance
                bot.append_turn("user", utterance)
                print(f"[{turn_idx}] User: {utterance}")

                # System response
                result = bot.chat(utterance)
                generated_reply = result["text"]

                # Find ground-truth reply (if exists next)
                gt_reply = None
                if turn_idx < len(dialogue) and dialogue[turn_idx]["speaker
                    "] == "assistant":
                    gt_reply = dialogue[turn_idx]["utterance"]

                # Append both generated and ground-truth responses
                dialogue_result.append({
                    "user_utterance": utterance,
                    "ground_truth": gt_reply,
                    "system_response": generated_reply,
                    "meta": {k: v for k, v in result.items() if k != "text"
                        },
                    "timestamp": datetime.now().strftime("%H:%M:%S")
                })

                bot.append_turn("assistant", generated_reply)
                print(f"    Bot: {generated_reply}\n")

        all_results.append({
            "dialogue_id": d_idx,
            "turns": dialogue_result
        })
        print(f"--- End of Dialogue {d_idx} ---\n")

    # Save to file
    os.makedirs("logs", exist_ok=True)
    if output_file is None:
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
```

```
        output_file = os.path.join("logs", f"batch_output_{timestamp}.json")

    with open(output_file, "w", encoding="utf-8") as f:
        json.dump(all_results, f, indent=4, ensure_ascii=False)

    print(f"Batch replay completed. Results saved to {output_file}\n")
    return output_file


if __name__ == "__main__":
    batch_replay("example_input.json")
```

Listing 1.4. Batch replay script for reproducible evaluation (`batch_replay.py`).

This script loads a predefined set of dialogues from `example_input.json`, runs each user turn through the system, and records both the system-generated reply and the ground-truth response from the dataset. Each dialogue session is processed independently to prevent cross-session interference, and all interaction logs are saved in a structured JSON format for further analysis. Such a batch replay setup is widely used in dialogue system research for **automatic evaluation**, ensuring that different systems are tested on exactly the same inputs under identical conditions.

3.4 Evaluating Batch Replay Output

After running the batch replay script, the system automatically generates a structured log file (`batch_output_{timestamp}.json`) under the `logs/` directory. Each dialogue entry contains the full interaction history, including the user utterances, system-generated responses, ground-truth references, and associated metadata.

A sample excerpt of the generated output file is shown in Listing 1.5. Each `turn` entry contains both the system's generated output (`system_response`) and the reference answer (`ground_truth`). This structure enables easy comparison between predicted and expected responses in later evaluation steps.

```
{
    "dialogue_id": 8,
    "turns": [
        {
            "user_utterance": "Do colleges compete with each other?",
            "ground_truth": "Yes, especially in sports like rowing. The May Bumps and Lent Bumps are famous races.",
            "system_response": "Do colleges compete with each other?",
            "meta": {
                "action": {"type": "echo"}
            },
            "timestamp": "09:37:38"
        },
```

```
        {
            "user_utterance": "Rowing sounds fun! Which college is the
                best at it?",
            "ground_truth": "It changes every year, but St John's, Caius,
                and Trinity are often at the top.",
            "system_response": "Rowing sounds fun! Which college is the
                best at it?",
            "meta": {
               "action": {"type": "echo"}
            },
            "timestamp": "09:37:38"
        }
    ]
}
```

Listing 1.5. Excerpt of generated batch replay output (`batch_output_*.json`).

The next step is to evaluate how well our system performs compared with the reference (ground-truth) responses. Here, we make a simplifying assumption: if a system's output is more similar to the ground truth, it is considered to be of higher quality. This assumption clearly has limitations—particularly given the *one-to-many* nature of dialogue, where multiple valid responses may exist for the same input. Nevertheless, such automatic metrics provide a fast, objective, and reproducible way to compare different systems under identical test conditions.

One of the most widely used metrics in NLP is **BLEU** [9]. BLEU measures the degree of overlap between the n-grams in a system-generated response and those in a reference response. Although originally designed for machine translation, it has become a standard baseline for text generation tasks, including dialogue.

Listing 1.6 shows a simple Python script for computing the average BLEU score across all dialogues stored in the batch output file. The script reads the generated system responses and their corresponding ground-truth replies, computes a BLEU score for each turn, and averages the results.

```python
from nltk.translate.bleu_score import sentence_bleu, SmoothingFunction
import json

def evaluate_bleu(output_file: str):
    with open(output_file, "r", encoding="utf-8") as f:
        dialogues = json.load(f)

    smoothie = SmoothingFunction().method4
    scores = []

    for d in dialogues:
        for turn in d["turns"]:
            gt = turn["ground_truth"]
            sys = turn["system_response"]
            if gt and sys:
```

```
                score = sentence_bleu([gt.split()], sys.split(),
                    smoothing_function=smoothie)
                scores.append(score)

    avg_bleu = sum(scores) / len(scores) if scores else 0.0
    print(f"Average BLEU score: {avg_bleu:.4f}")

if __name__ == "__main__":
    evaluate_bleu("logs/batch_output_{timestamp}.json")
```

Listing 1.6. Example script for automatic evaluation using BLEU.

This simple evaluation provides a reproducible numerical score that can be used to compare different versions of your system. While BLEU offers a quick and quantitative measure of lexical overlap, it does not necessarily reflect whether a response is *appropriate*, *coherent*, or *engaging*. Therefore, in dialogue research, BLEU is often complemented by other automatic metrics or by human evaluation to achieve a more comprehensive assessment of dialogue quality.

3.5 Building an LLM-based Dialogue Agent

In the previous sections, we built a simple but fully functional dialogue system. However, as its name suggests, the **ParrotBot** is only capable of repeating what it hears, which makes it of limited use. In this section, we take a major step forward by connecting our dialogue framework to a state-of-the-art LLM. This integration will make our system substantially more intelligent.

Modern large language models (LLMs) are extremely large, often containing hundreds of billions of parameters. As a result, they cannot be efficiently executed on consumer hardware and typically require powerful GPU clusters to run. To make such models widely accessible, most providers (e.g., OpenAI's GPT) now offer them as **API-based services**. Through simple API calls, developers can send text prompts to these models and receive generated responses in return.

In this handout, we will demonstrate how to query OpenAI's **GPT-5** model using Python, and illustrate how it can be integrated into the dialogue system developed earlier in this handout.

Once you have saved your API key in a file named `openai.key`, you can confirm that everything is working by running the simple test script shown in Listing 1.7. It sends a short prompt to one of OpenAI's LLMs and prints the generated reply.

```
from openai import OpenAI
import os
import sys

def load_openai_key(key_path: str = "openai.key"):
    if not os.path.exists(key_path):
        sys.exit(f"Error: The API key file '{key_path}' was not found.")

```

```
    with open(key_path, "r", encoding="utf-8") as f:
        api_key = f.read().strip()

    os.environ["OPENAI_API_KEY"] = api_key
    print("OpenAI API key loaded successfully.")

def test_openai_api():
    client = OpenAI()
    response = client.responses.create(
        model="gpt-5-nano-2025-08-07",
        input="Write a one-sentence fun fact about Cambridge."
    )

    print("\n--- Model Response ---")
    print(response.output_text)
    print("----------------------\n")

if __name__ == "__main__":
    load_openai_key()
    test_openai_api()
```

Listing 1.7. Example script for sending queries to OpenAI

The script performs two essential steps:

- **Load the API key safely.** The function `load_openai_key()` reads your secret key from a local file (`openai.key`) and sets it as an environment variable. This is the recommended way to handle authentication securely, instead of hard-coding your key into the source code.
- **Send a simple test request.** The function `test_openai_api()` creates an OpenAI client and sends a short prompt ("*Write a one-sentence fun fact about Cambridge.*") to the model `gpt-5-nano-2025-08-07`. The model's reply is then printed to confirm that the connection works correctly. The `gpt-5-nano` model is the smallest member of the GPT-5 family—fast, lightweight, and inexpensive to use—while still offering strong capabilities. It is therefore the recommended model for this assignment, as it provides an optimal balance between performance and cost.

After testing the OpenAI API successfully, we can now connect it to our dialogue system. Listing 1.8 shows the implementation of `GPTBot`, a dialogue agent powered by OpenAI's **GPT-5 Nano** model. It replaces the simple echoing mechanism of the `ParrotBot` with a real large language model that can generate context-aware responses.

```
from dialogue_system import DialogueSystem
from openai import OpenAI
import os
import sys

class GPTBot(DialogueSystem):
```

```
    """
    A dialogue agent powered by OpenAI's GPT-5 Nano model.
    Extends the DialogueSystem base class to generate responses
    using the OpenAI API.
    """

    def __init__(self, model: str = "gpt-5-nano-2025-08-07", key_path:
        str = "openai.key"):
        super().__init__()
        self.model = model
        self._load_openai_key(key_path)
        self.client = OpenAI()

    def _load_openai_key(self, key_path: str):
        if not os.path.exists(key_path):
            sys.exit(f"Error: The API key file '{key_path}' was not found.
                ")
        with open(key_path, "r", encoding="utf-8") as f:
            api_key = f.read().strip()
        os.environ["OPENAI_API_KEY"] = api_key
        print("OpenAI API key loaded successfully.\n")

    def chat(self, utterance: str) -> dict:
        """
        Send the dialogue history and the latest user utterance to the LLM
            , and return the generated response.
        """
        # Combine dialogue context
        messages = []
        for turn in self.conversation_history:
            role = "assistant" if turn["speaker"] == "assistant" else "
                user"
            messages.append({"role": role, "content": turn["utterance"]})

        messages.append({"role": "user", "content": utterance})

        # Call the GPT model
        response = self.client.responses.create(
            model=self.model,
            input=messages
        )

        reply_text = response.output_text.strip()

        return {
            "text": reply_text
        }

if __name__ == "__main__":
    bot = GPTBot()
```

```
53    bot.start_a_chat()
```

Listing 1.8. Implementation of the `GPTBot` class, which connects the dialogue framework to OpenAI's GPT-5 Nano model.

The `chat()` method now sends the accumulated dialogue history and the latest user input to the GPT model through the OpenAI API. The model then generates a response that takes the full conversational context into account. At this stage, our system has evolved from a rule-based echo bot into a true LLM-based conversational agent.

In the next section, we will explore how to control the behaviour of LLMs through **prompt engineering** and by fine-tuning their generation parameters.

3.6 Prompt Engineering and Generation Parameters

When generating dialogue responses, the behaviour of LLMs can be controlled in two complementary ways: (i) by designing effective **prompts** that define the model's role, objectives, and behaviour; and (ii) by tuning **generation parameters** that determine the randomness, diversity, and length of its outputs.

System Prompts and Role Definition. The **system prompt** defines the LLM's initial instruction. The same model can act as a friendly assistant, a formal tutor, or a domain expert, depending on how the prompt is written. In our `GPTBot`, we can add a system-level instruction before sending the conversation history to the model:

```
1  def chat(self, utterance: str) -> dict:
2      messages = [{"role": "system",
3                   "content": "You are a friendly and knowledgeable Cambridge
                                student who enjoys helping others learn about
                               college life."}]
4
5      for turn in self.conversation_history:
6          role = "assistant" if turn["speaker"] == "assistant" else "user"
7          messages.append({"role": role, "content": turn["utterance"]})
8
9      messages.append({"role": "user", "content": utterance})
10
11     response = self.client.responses.create(
12         model=self.model,
13         input=messages
14     )
15
16     reply_text = response.output_text.strip()
17     return {"text": reply_text}
```

Listing 1.9. Adding a system prompt to define the model's role and style.

This simple modification changes the system's personality and linguistic style. By altering the system prompt, we can directly influence how the model behaves

and responds to users. The process of designing clear and effective prompts to achieve the desired behaviour is known as **prompt engineering**.

Generation Parameters. Beyond the prompt, LLMs have several **generation parameters** that control the behaviour of text generation. The most common parameters include:

temperature
: Controls the randomness of generation (range 0–2). Higher values make outputs more creative but less stable.

max_tokens
: Sets the maximum number of tokens in the response. Useful for limiting overly long outputs.

top_p
: Enables *nucleus sampling*, where the model samples only from the top p cumulative probability mass. Lower values produce safer, more deterministic responses.

presence_penalty
: Penalises reusing previously mentioned topics, encouraging the model to explore new ideas.

frequency_penalty
: Penalises repeated words or phrases, reducing redundancy and promoting lexical variety.

For example, we can make the model more deterministic by setting a lower temperature and a small top-p, or make it more imaginative by increasing these values.

```
response = self.client.responses.create(
    model=self.model,
    input=messages,
    temperature=0.3,     # More deterministic and focused
    max_output_tokens=150, # Limit response length
    top_p=0.9            # Sample from top 90% probability mass
)
```

Listing 1.10. Using generation parameters to control model behaviour.

By adjusting these parameters, you can make the same model behave very differently—from a concise, factual assistant to a creative storyteller.

Not all OpenAI models support adjustable generation parameters such as `temperature` or `top_p`. Always check the official model documentation before using these options. In particular, `gpt-5-nano-2025-08-07` (and other models in the GPT-5 family) *do not* support these parameters. If

you wish to experiment with generation settings, please switch to a model from the GPT-4 family (e.g., `gpt-4o`).

3.7 Retrieval-Augmented Generation for Dialogue Systems

RAG combines a language model's generative capabilities with an external knowledge base. It is particularly effective for incorporating large volumes of domain-specific information that cannot be directly included in the model's input due to context length limitations. Moreover, RAG offers a cost-efficient way to extend a model's knowledge, since retrieving and inserting only the most relevant passages is substantially cheaper than processing long prompts.

The implementation below demonstrates how to build a retrieval-augmented dialogue system using the OpenAI API. The system stores a local knowledge base, computes text embeddings for retrieval, and dynamically fetches relevant snippets at runtime. The retrieved context is then appended to the system prompt so that the model can ground its response in factual knowledge.

```python
from dialogue_system import DialogueSystem
from openai import OpenAI
import numpy as np
import json
import os
import sys

class RAGBot(DialogueSystem):
    """
    A retrieval-augmented dialogue agent.
    It retrieves relevant knowledge snippets from a local knowledge base
    using OpenAI's embedding API before generating responses.
    """

    def __init__(self,
                 model: str = "gpt-5-nano-2025-08-07",
                 embedding_model: str = "text-embedding-3-small",
                 key_path: str = "openai.key",
                 kb_path: str = "cambridge_knowledge_list.json"):
        super().__init__()
        self.model = model
        self.embedding_model = embedding_model
        self.kb_path = kb_path
        self._load_openai_key(key_path)
        self.client = OpenAI()

        # --- Load Knowledge Base ---
        if not os.path.exists(kb_path):
            sys.exit(f"Error: The knowledge base '{kb_path}' was not found
                .")
```

```
        with open(kb_path, "r", encoding="utf-8") as f:
            self.knowledge_base = json.load(f)
        self.doc_texts = [d["text"] for d in self.knowledge_base]
        print(f"Knowledge base loaded with {len(self.doc_texts)} entries.\n")

        # --- Embedding Cache ---
        self.embedding_cache_path = os.path.splitext(kb_path)[0] + "_embeddings.npy"
        if os.path.exists(self.embedding_cache_path):
            print(f"Loading cached embeddings from {self.embedding_cache_path} ...")
            self.doc_embeddings = np.load(self.embedding_cache_path)
        else:
            print("Computing embeddings for knowledge base ...")
            self.doc_embeddings = self._embed_texts(self.doc_texts)
            np.save(self.embedding_cache_path, self.doc_embeddings)
            print(f"Embeddings saved to cache: {self.embedding_cache_path}\n")

    def _load_openai_key(self, key_path: str):
        if not os.path.exists(key_path):
            sys.exit(f"Error: The API key file '{key_path}' was not found.")
        with open(key_path, "r", encoding="utf-8") as f:
            api_key = f.read().strip()
        os.environ["OPENAI_API_KEY"] = api_key
        print("OpenAI API key loaded successfully.\n")

    def _embed_texts(self, texts):
        """Generate embeddings for a list of texts using OpenAI embedding API."""
        response = self.client.embeddings.create(
            model=self.embedding_model,
            input=texts
        )
        embeddings = [item.embedding for item in response.data]
        return np.array(embeddings)

    def _embed_query(self, query):
        """Generate embedding for a single query."""
        response = self.client.embeddings.create(
            model=self.embedding_model,
            input=[query]
        )
        return np.array(response.data[0].embedding)

    def retrieve_context(self, query, top_k: int = 3):
        """Retrieve top-k most relevant snippets using cosine similarity."""
```

```
        query_emb = self._embed_query(query)
        scores = np.dot(self.doc_embeddings, query_emb.T) / (
            np.linalg.norm(self.doc_embeddings, axis=1) * np.linalg.norm(
                query_emb)
        )
        top_indices = np.argsort(scores)[::-1][:top_k]
        retrieved_texts = [self.doc_texts[i] for i in top_indices]

        print("\nRetrieved Knowledge Snippets:")
        for i, text in enumerate(retrieved_texts, 1):
            print(f"[{i}] {text}")
        print("
            ----------------------------------------------------------\n
            ")

        return "\n".join(retrieved_texts)

    def chat(self, utterance: str) -> dict:
        """Main chat logic: retrieve context to generate answer."""
        retrieved_context = self.retrieve_context(utterance)
        system_prompt = (
            "You are a friendly and knowledgeable Cambridge student who
                helps "
            "others learn about university life. "
            "Use the retrieved context below to answer accurately and
                naturally. "
            "If you don't know, say so politely.\n\n"
            f"--- Retrieved context ---\n{retrieved_context}\n--- End
                context ---"
        )

        messages = [{"role": "system", "content": system_prompt}]
        for turn in self.conversation_history:
            role = "assistant" if turn["speaker"] == "assistant" else "
                user"
            messages.append({"role": role, "content": turn["utterance"]})
        messages.append({"role": "user", "content": utterance})

        response = self.client.responses.create(model=self.model, input=
            messages)
        reply_text = response.output_text.strip()

        return {"text": reply_text, "retrieved_context": retrieved_context
            }

if __name__ == "__main__":
    bot = RAGBot()
    bot.start_a_chat()
```

Listing 1.11. A retrieval-augmented dialogue agent built with OpenAI embeddings.

In this implementation, the agent first loads a local knowledge base in JSON format, where each entry contains a short text snippet about Cambridge. To enable efficient retrieval, the system computes embeddings for all entries using OpenAI's `text-embedding-3-small` model. These embeddings are stored in a NumPy array and cached locally in a file so that subsequent runs do not require recomputation. At runtime, when the user submits a query, the system performs the following steps:

1. **Embed the query.** The user's input is converted into an embedding vector using the same model as the knowledge base.
2. **Retrieve context.** The query embedding is compared with all document embeddings using cosine similarity. The top-k most similar snippets are selected as the *retrieved context*.
3. **Construct prompt.** The retrieved context is appended to a system prompt. This ensures that the model's generation is grounded in relevant factual information.
4. **Generate response.** The full conversation history, together with the updated system prompt, is sent to the LLM to generate a reply.

The embedding cache is an important practical optimisation. Computing embeddings for a large knowledge base can be slow and costly, but these vectors are static—once computed, they do not change. By saving them locally, the system avoids recomputing embeddings every time it runs, which reduces both latency and API cost while ensuring results remain reproducible across sessions.

> For the assignment, you can adapt this system to other domains by replacing the knowledge base file (`cambridge_knowledge_list.json`) with your own knowledge dataset. Each entry should contain a short, self-contained text passage relevant to your chosen use case.

3.8 Semantic Parsing for Dialogue Systems

Semantic parsing maps user utterances into structured queries that can be executed by other computer programs or APIs. This allows dialogue systems to interact with databases, services, or external APIs, enabling them to perform actions in the real world. Such systems typically include a separate module—known as the *semantic parser*—that translates natural language inputs into structured representations (e.g., JSON, SQL, or API calls) which downstream components can process or execute.

The design of this module usually depends on the structure and functionality of the APIs it interacts with. In this handout, we will illustrate this idea using a small example: a dialogue system for **Jack's Gelato**, which allows users to interact with the shop's ordering and information APIs. This mock API takes a structured query and generates a corresponding picture of the ice cream order.

```python
from PIL import Image, ImageDraw
import difflib

flavor_colors = {
    'Baked Vanilla': '#F3E5AB',
    'Manuka Honey & Fig': '#D2B48C',
    'Roasted Banana': '#FFE135',
    'Caramel': '#A67B5B',
    'Kings College Lavender': '#E6E6FA',
    'Gianduja (vegan)': '#6B4423',
    'Passion Fruit Sorbet (vegan)': '#FFD700',
    'House Yoghurt': '#FFFFF0',
    'Dark Chocolate & Sea Salt (vegan)': '#8B4513',
    'Treacle Cake': '#664229',
    'Strawberries & Cream': '#FFC0CB',
    'Coconut, Raspberry Ripple (vegan)': '#FF3366',
    'Organic Whisky; Ncnean': '#DAA520',
    'coconut and ube': '#7D26CD'
}

def draw_ice_cream(flavours, size, container):
    width, height = 400, 500
    img = Image.new('RGB', (width, height), 'lightblue')
    draw = ImageDraw.Draw(img)
    sizes = {'Single Scoop': 1, 'Double Scoop': 2, 'Triple Scoop': 3}
    num_scoops = sizes[size]
    scoop_y = 300 - (100 * num_scoops)
    for i in range(num_scoops):
        color = flavor_colors.get(flavours[i], 'lightblue')
        draw.ellipse([150, scoop_y, 250, scoop_y + 100], fill=color)
        scoop_y += 100
    if container == 'Normal Cone':
        draw.polygon([(160, 300), (240, 300), (200, 400)], fill='#D2B48C')
    elif container == 'Paper Cup':
        draw.rectangle([150, 300, 250, 400], fill='#F5F5F5')
    elif container == 'Chocolate Dipped Waffle Cone':
        draw.polygon([(160, 300), (240, 300), (200, 400)], fill='#8B4513')
    img_path = './ice_cream.png'
    img.save(img_path)
    return img_path

def find_most_similar(available, inputs):
    matches = []
    for value in inputs:
        best = difflib.get_close_matches(value, available, n=1, cutoff
            =0.0)
        matches.append(best[0] if best else None)
    return matches

```

```python
def get_gelato(state):
    flavours = find_most_similar(flavor_colors.keys(), state["flavours"])
    size = find_most_similar(["Single Scoop", "Double Scoop", "Triple
        Scoop"], [state["size"]])[0]
    container = find_most_similar(["Normal Cone", "Paper Cup", "Chocolate
         Dipped Waffle Cone"], [state["container"]])[0]
    return draw_ice_cream(flavours, size, container)

if __name__ == '__main__':
    state = {
        "flavours": ["House Yoghurt", "ube"],
        "size": "Double Scoops",
        "container": "cone"
    }
    image_path = get_gelato(state)
    print(f"Image saved to {image_path}")
```

Listing 1.12. Example API implementation for Jack's Gelato.

In this example, the system generates an image of an ice cream order based on three structured attributes: `flavours`, `size`, and `container`. When the example query above is executed, the API produces the following output image, illustrating the result of semantic parsing and action execution.

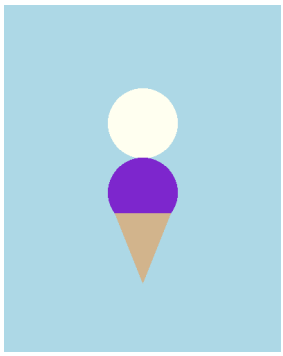

Fig. 2. Example output of the Jack's Gelato API, generated from a structured query.

To complete the pipeline, we can use a language model to automatically generate the structured query from natural language dialogue. In this setup, the model plays the role of a *semantic parser*: it extracts structured fields such as `flavours`, `size`, and `container` from the user's utterances. Below is an implementation of a simple semantic parser that uses OpenAI's GPT models to perform this extraction.

```python
from openai import OpenAI
import os, json
```

```python

def parse_gelato_order(conversation, model="gpt-5-nano-2025-08-07", key_path="openai.key"):
    """
    Use OpenAI LLM to parse a conversation into a structured gelato order. Any missing fields are returned as empty strings.
    """

    # --- Load API key ---
    if not os.path.exists(key_path):
        raise FileNotFoundError(f"API key file not found at {key_path}")
    with open(key_path, "r") as f:
        os.environ["OPENAI_API_KEY"] = f.read().strip()

    client = OpenAI()

    system_prompt = (
        "You are a semantic parser for an ice cream shop called Jack's Gelato. "
        "Given a conversation between a user and assistant, extract the user's final order "
        "as a JSON object with the following fields: "
        "flavours (list of strings), size (string), and container (string). "
        "If any information is missing, use an empty string (''). "
        "Return only valid JSON and nothing else."
    )

    messages = [{"role": "system", "content": system_prompt},
                {"role": "user", "content": conversation}]

    response = client.responses.create(model=model, input=messages)
    parsed_json = response.output_text.strip()

    try:
        order = json.loads(parsed_json)
    except json.JSONDecodeError:
        print("Model output not valid JSON, returning empty fields.")
        order = {"flavours": [], "size": "", "container": ""}

    return order

if __name__ == "__main__":
    conversation = """
    User: Hi! What flavours do you have today?
    Assistant: We have House Yoghurt, Coconut and Ube, and Dark Chocolate & Sea Salt.
    User: Great, can I get a double scoop of yoghurt and ube in a cone please?
    Assistant: Sure! One double scoop of House Yoghurt and Ube in a cone.
```

```
    """
    parsed = parse_gelato_order(conversation)
    print(json.dumps(parsed, indent=4))
```

Listing 1.13. LLM-based semantic parser for Jack's Gelato.

The core of building this semantic parser lies in the design of the **system prompt**. Since LLMs operate primarily by following natural language instructions, they require a carefully and clearly designed prompt to perform structured information extraction reliably. In this case, the prompt explicitly defines:

- **The role and task:** The model is instructed to act as a *semantic parser* for an ice cream shop, extracting order information from dialogue.
- **The expected input:** A conversation between a user and an assistant, written in plain text.
- **The required output format:** A strict JSON object containing three fields—`flavours`, `size`, and `container`. The prompt also specifies that if information is missing, the model should return empty strings instead of guessing.

With both the semantic parser and the API ready, we can now integrate them into a full dialogue system. The system should collect user inputs, use the LLM-based parser to convert the dialogue history into a structured order, and then pass that structured query to the API function to produce the final output.

The following example shows how to combine the `parse_gelato_order()` function with the `get_gelato()` API call to create an end-to-end conversational agent called `GelatoBot`.

```
from openai import OpenAI
from gelato_api import get_gelato
from gelato_semantic_parser import parse_gelato_order
import os, json, sys


class GelatoBot:
    """
    A conversational agent that understands user utterances about gelato,
        parses them into structured orders, and uses an LLM to generate
        natural responses. When the order is complete, it calls the
        Gelato API to generate the ice cream image.
    """

    def __init__(self, model="gpt-5-nano-2025-08-07", key_path="openai.
        key"):
        self._load_openai_key(key_path)
        self.client = OpenAI()
        self.model = model
        self.conversation_history = []

```

```
    def _load_openai_key(self, key_path: str):
        if not os.path.exists(key_path):
            sys.exit(f"Error: The API key file '{key_path}' was not found.")
        with open(key_path, "r", encoding="utf-8") as f:
            api_key = f.read().strip()
        os.environ["OPENAI_API_KEY"] = api_key
        print("OpenAI API key loaded successfully.\n")

    def chat(self, user_input: str):
        self.conversation_history.append({"role": "user", "content": user_input})

        # Combine full conversation into one text block
        conversation = "\n".join(
            [f"{m['role'].capitalize()}: {m['content']}" for m in self.conversation_history]
        )

        # --- Step 1: Semantic parsing ---
        order = parse_gelato_order(conversation)
        print("\nParsed Order:\n", json.dumps(order, indent=4))

        # --- Step 2: Check if order complete ---
        order_complete = bool(order["flavours"] and order["size"] and order["container"])

        # --- Step 3: Use LLM to generate response ---
        system_prompt = (
            "You are GelatoBot, a friendly and polite assistant working at Jack's Gelato. "
            "You help customers place ice cream orders and make small talk if needed. "
            "When the order is complete, confirm the details cheerfully. "
            "If something is missing (flavours, size, or container), ask naturally for clarification."
        )

        user_prompt = (
            f"Here is the current parsed order:\n{json.dumps(order, indent=4)}\n\n"
            f"Conversation so far:\n{conversation}\n\n"
            f"Please write the next assistant message."
        )

        response = self.client.responses.create(
            model=self.model,
            input=[{"role": "system", "content": system_prompt},
                   {"role": "user", "content": user_prompt}]
        )
```

```
        reply_text = response.output_text.strip()

        print("\nAssistant:", reply_text)

        # --- Step 4: Generate gelato image if ready ---
        if order_complete:
            image_path = get_gelato(order)
            print(f"Gelato image saved to: {image_path}\n")

        self.conversation_history.append({"role": "assistant", "content":
            reply_text})

    def start(self):
        print("Welcome to GelatoBot! Type 'bye' or 'exit' to exit.\n")
        while True:
            user_input = input("You: ")
            if user_input.lower() in {"bye", "exit"}:
                print("Goodbye!")
                break
            self.chat(user_input)


if __name__ == "__main__":
    bot = GelatoBot()
    bot.start()
```

Listing 1.14. A dialogue system combining semantic parsing and API execution.

The `GelatoBot` implementation illustrates how to integrate a semantic parser into the dialogue system pipeline, following the architecture shown in Figure 1. At the core of this system lies the **system prompt**, which defines the assistant's role, behaviour, and conversational goals, serving as the guiding specification for the model. When designing prompts for such agents, it is essential to:

- **Clearly specify the task:** describe what the assistant is expected to do (e.g., help users place gelato orders politely and efficiently).
- **Define the input and output formats:** the model receives the conversation history and the parsed order as input, and should produce a single, well-formed assistant response as output.
- **Constrain style and tone:** specify that the assistant should remain friendly, concise, and natural.

Although the current version of `GelatoBot` serves as a baseline, its behaviour can be substantially improved through *prompt engineering*. Beyond simple task specification, effective prompt design can embed domain knowledge directly into the model's behaviour and guide it to manage dialogue flow more naturally and coherently. Several strategies are particularly useful in this context. Here are two examples:

– **Describe the expected conversation flow.** Clearly outline how the assistant should progress through stages such as greeting, information gathering, order confirmation, and closure.
– **Use ICL.** Include a small number of short example dialogues that illustrate the desired style and structure of conversation, especially how the assistant asks clarification questions or confirms details.

In the assignment, you are encouraged to design your own use case and adapt this pipeline to your chosen scenario. By experimenting with different prompt designs and dialogue strategies, you can observe how subtle variations influence system behaviour, consistency, and user experience.

4 Other Tools for Building Conversational AI

In addition to the OpenAI API, a number of open-source tools and frameworks are widely used for building, training, and deploying conversational agents. This section briefly introduces several key libraries that you may find useful when prototyping or experimenting with your own system. You are welcome to use any of these tools in your assignment, although doing so is not required.

4.1 Hugging Face Transformers

While API-based LLMs such as OpenAI's GPT models provide convenient access to state-of-the-art capabilities, they typically do not allow much flexibility for fine-tuning or model customisation.[4] In contrast, the `Transformers` library by Hugging Face offers a fully open-source framework for developing and experimenting with transformer-based models—the architecture underlying most modern LLMs.

The `Transformers` library (`https://huggingface.co/docs/transformers/en/index`) provides a user-friendly toolkit for loading, fine-tuning, and running a wide range of pretrained models, including BERT, T5, GPT-2, and LLaMA. With sufficient computational resources, it also supports fine-tuning very large models (e.g., `deepseek-ai/DeepSeek-R1`). This makes it a powerful framework for both research and system development with open models, enabling rapid prototyping and evaluation of customised conversational agents. In the example below (Listing 1.15), we demonstrate how to use this package to implement the same Gelato order parser introduced earlier in Listing 1.13.

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM,
    AutoModelForCausalLM
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"

```

[4] Some API providers do support fine-tuning, but the process is generally less transparent and less flexible than using Hugging Face Transformers.

```python
class ParsingModel():
    def __init__(self):
        pass

    def predict(self, history):
        raise NotImplementedError()

class GelatoParsingModel(ParsingModel):
    def __init__(self, model_path="google-t5/t5-base"):
        super().__init__()
        self.model_path = model_path
        self.tokenizer = AutoTokenizer.from_pretrained(self.model_path)
        self.model = AutoModelForSeq2SeqLM.from_pretrained(self.model_path
            ).to(device)

    def history_to_string(self, history):
        assert isinstance(history, list)
        processed_history = " ".join(list(map(lambda x: x["speaker"] + ":
            " + x["utterance"], history)))
        return processed_history

    def state_to_string(self, state):
        state_st = "flavours: " + ", ".join(state["flavours"]) + "# size:
            " + state["size"] + "# container: " + state[
           "container"]
        return state_st

    def string_to_state(self, state_string):

        pairs = state_string.split("# ")
        state = {}
        for pair in pairs:
            key, value = pair.split(": ", 1)
            if key == "flavours":
                value = value.split(", ")
            state[key] = value
        return state

    def predict(self, history):

        prefix = "dialogue state tracking"
        context_text = self.history_to_string(history)
        inputs = prefix + " : " + context_text
        model_inputs = self.tokenizer([inputs], return_tensors="pt").to(
            device)
        generated_ids = self.model.generate(**model_inputs, max_new_tokens
            =512)
        output = self.tokenizer.batch_decode(generated_ids,
            skip_special_tokens=True, clean_up_tokenization_spaces=True)
            [0]
```

```
        try:
            state = self.string_to_state(output)
        except:
            state = {"flavours": [], "size": "", "container": ""}
        return state
```

Listing 1.15. Example semantic parsing model using Hugging Face Transformers.

While pretrained models such as `t5-base` demonstrate strong general-purpose language understanding, they are not specialised for structured tasks such as semantic parsing. In this section, we focus on relatively small models rather than truly large language models; with carefully designed prompts, such models can still achieve decent performance. However, as shown below, if we directly use a pretrained model without any task-specific fine-tuning, the model tends to produce generic or irrelevant text outputs instead of the structured state representations required by our dialogue system.

```
if __name__ == "__main__":
    dst_model = GelatoParsingModel(model_path="google-t5/t5-base")

    history = [
        {"speaker": "customer", "utterance": "Hello! I am looking for an
            ice cream?"},
        {"speaker": "assistant", "utterance": "Yes, we have several
            options."},
        {"speaker": "customer", "utterance": "Can I have a double scoop
            with coconut and ube and Dark Chocolate & Sea Salt, in a
            normal cone."}
    ]

    print(dst_model.predict(history))
```

Listing 1.16. Running a base T5 model without fine-tuning.

Without fine-tuning, the output is unstructured and unrelated to the intended task:

```
{'customer': 'Hello! I am looking for an icecream? assistant: Hello!
    customer: Hello!'}
```

After fine-tuning the model on a small dataset of example dialogues and their corresponding queries, the same code produces structured and meaningful outputs that align with the API format required by our system:

```
{'flavours': ['coconut and ube', 'Dark Chocolate & Sea Salt'], 'size': '
    Double Scoop', 'container': 'Normal Cone'}
```

The following script provides a reference implementation for fine-tuning a T5 model for the gelato order parsing task. It uses a dataset file (`dst_data.json`) containing input–output pairs of source dialogues and corresponding structured

queries. The script tokenises the data, trains the model for a small number of steps, and produces a fine-tuned model suitable for the semantic parsing component in our dialogue system.

```
import pandas as pd
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM, \
    DataCollatorForSeq2Seq, Seq2SeqTrainer, Seq2SeqTrainingArguments
from transformers import set_seed
from datasets import Dataset, DatasetDict
import json

def load_json_from_file(filename):
    with open(filename, 'r') as file:
        return json.load(file)

def run_experiment():
    model_name = "google-t5/t5-base"
    set_seed(10086)

    tokenizer = AutoTokenizer.from_pretrained(model_name, max_length=512)
    data_json = load_json_from_file("dst_data.json")

    for k, v in data_json.items():
        v = Dataset.from_pandas(pd.DataFrame.from_dict(v))
        data_json[k] = v
    data_dic = DatasetDict(data_json)

    prefix = "dialogue state tracking"
    model = AutoModelForSeq2SeqLM.from_pretrained(model_name, max_length
        =512)

    def preprocess_function(examples):
        inputs = [prefix + " : " + s for s in examples["source"]]
        targets = examples["target"]
        return tokenizer(inputs, text_target=targets, max_length=512)

    tokenized_dataset = data_dic.map(preprocess_function, batched=True)
    data_collator = DataCollatorForSeq2Seq(tokenizer=tokenizer, model=
        model)

    training_args = Seq2SeqTrainingArguments(
        output_dir="./output/dst_model",
        learning_rate=1e-3,
        per_device_train_batch_size=32,
        weight_decay=0.01,
        predict_with_generate=True,
        max_steps=5000,
        save_total_limit=1
    )

```

```
    trainer = Seq2SeqTrainer(
        model=model,
        args=training_args,
        train_dataset=tokenized_dataset["train"],
        eval_dataset=tokenized_dataset["val"],
        tokenizer=tokenizer,
        data_collator=data_collator
    )

    trainer.train()
    trainer.save_model("./output/dst_model/checkpoint-best")

if __name__ == '__main__':
    run_experiment()
```

Listing 1.17. Reference fine-tuning script for dialogue state tracking with Hugging Face.

For this assignment, **you are not required to train or fine-tune any models.** All examples provided in this handout are for illustration purposes only. If you are interested in fine-tuning your own model as part of the assignment, please contact me in advance. Fine-tuning requires access to a GPU and a sufficiently large dataset. The purpose of this section is to help you understand the two dominant paradigms in NLP development:

- **Fine-tuning:** adapting a smaller pretrained model to a specific task using supervised data;
- **In-context learning (ICL):** guiding large language models to perform new tasks through well-designed prompts, without additional training.

Although ICL with powerful LLMs often achieves good performance, such models are typically large and expensive to run. In many practical scenarios, fine-tuning smaller open models can be a more cost-effective and scalable alternative.

4.2 LangChain and High-Level Frameworks

While implementing dialogue systems from scratch helps you understand their internal mechanisms and offers maximum flexibility, modern high-level frameworks such as `LangChain` (`https://github.com/langchain-ai/langchain`) provide convenient abstractions that significantly simplify the development process. These frameworks are typically engineered for performance and often run faster due to their optimised implementations.

LangChain integrates LLMs, retrievers, vector stores, and prompt templates into a modular pipeline, allowing developers to prototype RAG systems with

minimal code. Although such frameworks are generally less flexible than fully custom implementations, they are highly optimised for scalability, efficiency, and interoperability, making them well suited for production-scale applications. Listing 1.18 demonstrates how the same RAGBot introduced in Listing 1.11 can be implemented using LangChain.

```
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_community.vectorstores import Chroma
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_core.prompts import PromptTemplate
import os, json, sys


class LangChainRAGBot:

    def __init__(self,
                 model_name="gpt-4o-mini",
                 embedding_model="text-embedding-3-small",
                 key_path="openai.key",
                 kb_path="cambridge_knowledge_list.json"):
        self._load_openai_key(key_path)

        if not os.path.exists(kb_path):
            sys.exit(f"Error: The knowledge base '{kb_path}' was not found
                .")
        with open(kb_path, "r", encoding="utf-8") as f:
            knowledge_base = json.load(f)
        texts = [doc["text"] for doc in knowledge_base]
        print(f"Knowledge base loaded with {len(texts)} entries.\n")

        embeddings = OpenAIEmbeddings(model=embedding_model)
        splitter = RecursiveCharacterTextSplitter(chunk_size=500,
            chunk_overlap=50)
        docs = splitter.create_documents(texts)
        self.vectorstore = Chroma.from_documents(docs, embeddings)
        self.retriever = self.vectorstore.as_retriever(search_kwargs={"k":
             3})

        self.llm = ChatOpenAI(model=model_name)
        self.prompt = PromptTemplate.from_template(
            "You are a helpful Cambridge student. "
            "Answer the question using the context below. "
            "If unsure, say you don't know.\n\n"
            "Conversation History:\n{history}\n\n"
            "Context:\n{context}\n\n"
            "Question:\n{question}"
        )

        self.history = []

```

```
        print("LangChain RAGBot initialised successfully.\n")

    def _load_openai_key(self, key_path: str):
        if not os.path.exists(key_path):
            sys.exit(f"Error: The API key file '{key_path}' was not found.
                ")
        with open(key_path, "r", encoding="utf-8") as f:
            api_key = f.read().strip()
        os.environ["OPENAI_API_KEY"] = api_key
        print("OpenAI API key loaded successfully.\n")

    def chat(self, user_input: str):
        docs = self.retriever.get_relevant_documents(user_input)
        context = "\n".join([d.page_content for d in docs])

        history_text = ""
        for user, assistant in self.history[-3:]:
            history_text += f"User: {user}\nAssistant: {assistant}\n"
        history_text += f"User: {user_input}\n"

        final_prompt = self.prompt.format(context=context, history=
            history_text, question=user_input)

        response = self.llm.invoke(final_prompt)
        reply = response.content.strip()

        print(f"\nAssistant: {reply}\n")

        self.history.append((user_input, reply))

    def start(self):
        print("Welcome to LangChain RAGBot! Type 'bye' or 'exit' to quit.\
            n")
        while True:
            user_input = input("You: ")
            if user_input.lower() in {"exit", "bye"}:
                print("Goodbye!")
                break
            self.chat(user_input)


if __name__ == "__main__":
    bot = LangChainRAGBot()
    bot.start()
```

Listing 1.18. A retrieval-augmented dialogue system implemented with LangChain.

5 Conclusion and Submission Guidelines

This handout introduced the key concepts and practical implementations involved in building conversational AI systems, including RAG, semantic parsing, and dialogue management with LLMs. For the assignment, you are required to submit the following materials:

1. **Your report** (maximum 1,500 words), summarising your system design, implementation, and key findings.
2. **Your system implementation**, including all source code, configuration files, and any supporting resources (e.g., RAG knowledge base or external API code).
3. **A list of testing dialogues** (at least 10), demonstrating how your system handles different user inputs and scenarios.
4. **System outputs** corresponding to each test dialogue, showing the system's responses.

If you have any questions or encounter technical difficulties, please contact me at `sh2091@cam.ac.uk`.

Bibliography

[1] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, no. 236, pp. 433–460, 1950. [Online]. Available: https://doi.org/10.1093/mind/LIX.236.433

[2] J. Weizenbaum, "Eliza—a computer program for the study of natural language communication between man and machine," *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, 1966.

[3] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[4] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*, 3rd ed., 2025, online manuscript released January 12, 2025. [Online]. Available: https://web.stanford.edu/~jurafsky/slp3/

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423/

[6] O. Vinyals and Q. V. Le, "A neural conversational model," in *ICML Deep Learning Workshop*, 2015. [Online]. Available: http://arxiv.org/pdf/1506.05869v3.pdf

[7] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[8] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, Jan. 2020.

[9] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[10] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Ann Arbor, Michigan: Association for Computational Linguistics, Jun. 2005, pp. 65–72. [Online]. Available: https://aclanthology.org/W05-0909

[11] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association

for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013

[12] S. E. Finch and J. D. Choi, "Towards unified dialogue system evaluation: A comprehensive analysis of current evaluation protocols," in *Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, O. Pietquin, S. Muresan, V. Chen, C. Kennington, D. Vandyke, N. Dethlefs, K. Inoue, E. Ekstedt, and S. Ultes, Eds. 1st virtual meeting: Association for Computational Linguistics, Jul. 2020, pp. 236–245. [Online]. Available: https://aclanthology.org/2020.sigdial-1.29/

[13] S. Mehri and M. Eskenazi, "Unsupervised evaluation of interactive dialog with DialoGPT," in *Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, O. Pietquin, S. Muresan, V. Chen, C. Kennington, D. Vandyke, N. Dethlefs, K. Inoue, E. Ekstedt, and S. Ultes, Eds. 1st virtual meeting: Association for Computational Linguistics, Jul. 2020, pp. 225–235. [Online]. Available: https://aclanthology.org/2020.sigdial-1.28/