CS 224V Assignment 2

Due: October 13th, 2025

Instructions: Use this Colab Notebook in conjunction with this write-up. Make sure to "Save a copy in Drive" before running the notebook. Submit your answers through Gradescope and attach your Google Colab notebook. In red, we label how each question in this writeup corresponds to a Gradescope question

We expect heavy loads on the OVAL machine used in this assignment close to the deadline. **Thus, we highly recommend you start this assignment early and do not wait until the last minute.** This assignment is designed to be completed in **groups of 2**. Please submit as a group to Gradescope.

1 Introduction

This assignment is designed to give you hands-on experience with how we can leverage LLMs to create task-oriented dialogue agents grounded on knowledge corpus. You will learn:

- strategies for leveraging LLMs to develop task-oriented dialogue systems capable of performing complex tasks;
- key requirements and considerations for building effective dialogue agents;
- the importance of grounding task agents in a comprehensive knowledge corpus.
- a new framework for quickly creating dialogue agents with minimal steps.

2 Task Oriented Dialogue Agents

Researchers and industry practitioners have demonstrated significant interest in developing task-oriented dialogue agents. These agents are typically designed with a transactional focus, aiming to fill slot values based on user utterances to complete specific tasks (Budzianowski et al., 2018; Andreas et al., 2020; Rastogi et al., 2020). However, existing approaches have limitations in handling conditional logic, integrating knowledge sources, and consistently following instructions. Researchers and industry professionals often employ ad hoc pipelines to construct conversational agents. These pipelines aim to maintain context, address failure cases, and minimize hallucinations, yet frequently fail to achieve these objectives.

LLMs offer a promising opportunity to create more natural general-purpose dialogue agents such as ChatGPT, Claude, and Gemini (OpenAI, 2024; Anthropic, 2024; Gemini et al., 2023). However, task-oriented agents should provide reliable, grounded responses while letting developers exercise control over the conversation flow.

Here are the key challenges faced in creating reliable task-oriented dialogue agents:

1. Creation of effective, informative, and responsive informative agents, while letting developers exercise control without onerous efforts.
2. Support users' queries for information, which may be embedded in a task request.
3. Dialogue systems need to remember pertinent facts from the dialogue history.

3 GenieWorksheet

To tackle the aforementioned challenges, task-oriented conversational agents recent libraries, such as those offered by LangChain and Guidance, provide abstractions for developing LLM-based agents but still require developers to manually craft prompts and create reliable pipelines. This is particularly challenging for complex task-oriented agents.

Genie (Joshi et al., 2025) was designed to address this gap. Instead of asking developers to hand-craft prompt pipelines, Genie introduces a new abstraction called the GenieWorksheet. A worksheet is a declarative specification of how a task-oriented dialogue agent should behave.

With worksheets, the developer does not need to spell out every possible dialogue path (as in traditional dialogue trees). Instead, they declare the task requirements: what information the agent should collect, what conditions must be satisfied, and what actions should be taken once those requirements are met. Genie then handles the details of prompting, state tracking, and conversation flow. This is a novel way of defining task-oriented agents, GenieWorksheets, that provides explicit control to the TOD (Task Oriented Dialogue) agent developer.

Imperative vs. Declarative

GenieWorksheet enables programming policies in the declarative paradigm, in contrast to dialogue trees' much more complex imperative approach. Declarative programming is a paradigm that focuses on what the program should achieve without explicitly stating how to achieve it.

- **Imperative (dialogue tree):** `If the user provides a departure city, then ask for arrival; if they give arrival, then ask for time; if all three are present, call book_ride().`
- **Declarative (worksheet):** `I need fields {departure, arrival, time}.`

```
Once they're filled, call book_ride().
```

The developer can program what actions the agent should take based on the state of the conversations; these are called agent policies. This way the developer can manage dialogue flow and deliver high-level support for integrated knowledge assistants.

Further Reading: Dialogue Agent Frameworks

GenieWorksheets is one approach to designing task-oriented dialogue agents, but it sits in a broader ecosystem of tools. Each framework has its own strengths and weaknesses, and understanding the trade-offs can help you choose the right tool for a given application:

- **LangChain**[a]: The most widely used orchestration framework for LLM-powered apps. Provides modular components (chains, agents, memory) and strong integrations, but can be complex to maintain in large projects.
- **Guidance**[b]: A Python library for precise prompt programming with control flow and constraints. Useful when you need structured outputs or deterministic behaviors.
- **Haystack**[c]: A framework focused on retrieval-augmented generation (RAG). Particularly strong for document-grounded agents that need to search, retrieve, and cite knowledge sources.

Compared to these, Genie emphasizes a **declarative paradigm** with worksheets, which simplifies complex dialogue flows and makes it easier to ensure reliability in task-oriented contexts.

[a] https://www.langchain.com
[b] https://github.com/guidance-ai/guidance
[c] https://haystack.deepset.ai

4 Investment Agent with Genie Worksheets

Managing personal investments often requires navigating multiple platforms—checking account balances on Fidelity, comparing fund options, researching returns, and manually executing deposits or fund purchases. Investors face challenges such as finding the right fund that matches their risk tolerance, managing certificate of deposits, or tracking how their existing portfolio is performing relative to benchmarks. These processes are fragmented and require constant switching between dashboards, reports, and calculators.

The Investment agent streamlines the process of managing and growing investments by integrating key financial actions with data-driven recommendations. It enables users to:

- `GetRecommendation`: receive tailored fund or investment suggestions based on goals and risk appetite.
- `GetAccountBalance`: instantly retrieve their account balance for real-time financial awareness.
- `UsersInvestmentPortfolio`: view and analyze their current portfolio composition, including fund allocations and performance.
- `CertificateDepositInvestment`: explore and invest in certificate of deposit opportunities with clear terms and yield comparisons.
- `FundInvestment`: research, select, and execute investments into Fidelity-managed funds or ETFs.

The agent operates on top of a comprehensive database containing detailed information about Fidelity's funds—including historical performance, risk ratings, sector allocations, and expense ratios. This structured data allows the agent to both answer questions ("What are the top-performing Fidelity bond funds this quarter?") and execute actions ("Invest $2,000 into Fidelity Balanced Fund").

To get familiar with task-oriented dialogue agents, you will interact with the (Fidelity) investment agent. We use real-world fund data from Fidelity's public disclosures. This database includes tables covering funds, performance metrics, enabling realistic simulation of investment workflows.

Action Item 1

Learning Goal: Get familiar with task-oriented dialogue agents grounded in knowledge corpus.

Task 1 Interact with the (fidelity) investment agent for at least 8 turns. Hint: student input + agent response = 1 turn (Gradescope Q1)

Steps

1. Visit Investment Agent Web UI. There you can interact with the agent.
2. The bot will give you a unique User ID, paste it into Gradescope (Gradescope 1.1)
3. Now you can ask questions related to Fidelity's public investment courses, ask for recommendations, ask to invest etc.
4. You should view the intermediate steps by clicking on down arrow for each of the three steps: "Semantic Parser", "Agent Policy", and "Response Generator".
5. Download the conversation log by clicking on *conv_log.json*. This conversation log contains intermediate steps taken by the agent.
6. Upload the conversation log to Gradescope (Gradescope 1.2)
7. Do you think the agent was forgetting information that you had previously mentioned in the chat? (Gradescope 1.3)
8. How often does the agent make up information ("hallucinate")? Provide examples from the conversation that you had. (Gradescope 1.4)
9. What functionalities do you think such an agent have in addition to the existing ones? (Gradescope 1.5)
10. List other applications where such task-oriented dialogue agents can be useful. (Gradescope 1.6)

5 Creating your own task-oriented agent

In the previous section, we used the Investment Agent to learn more about courses offered and enroll in classes. Now, we will look into what it takes to create task-oriented agents with GenieWorksheets.

5.1 Components of GenieWorksheets

GenieWorksheets has the following three components:

- **APIs:** External APIs that the agent will have access to, enabling it to perform designated actions.
- **Databases:** Knowledge sources that the agent will use to respond to user queries requiring external information.
- **Worksheets:** Worksheets specify how the agent should complete a particular task. Each worksheet represents one task or subtask, defining when it is activated, what information to collect from the user, and what actions to take based on that input (such as API calls, database lookups, or responses). The top-level worksheet always runs first to capture the main task, while additional worksheets are created for subtasks or alternative flows that are triggered only when their conditions are satisfied.

See an example of worksheets for the Investment Agent here: Investment Agent Worksheets. See an example of APIs defined for the Investment Agent in this file: `investment_agent.py`

Ticket Submission Worksheet

WS Predicate	WS Name	Predicate	Input	Type	Name	Enum Values	Description	Don't Ask	Required	Confirmation	Actions	WS Actions
	Main			**WS**	**submit_ticket**							
			input	Enum	student_task		The type of student requ...		TRUE			
						TroubleShoot						
						Leave of Abs.						
						Test Credits						
		self.student_task == "TroubleShoot"	input	Trouble Shoot	trouble_shoot		The enrollment issues that the student is facing		TRUE			
						...						
		self.student_task is not None and (self.trouble_shoot and self.leave_of_abs and self.test_credits)	input	str	extra_details		Ask for any other detail that the student wants to add		TRUE			
			input	confirm	confirm		Confirm the student w...		TRUE		if self.confirm == False: >say("Thank you, can I assist you in any other way?")	
self.student_task == "Trouble..."	**TroubleShoot**			**WS**								
						...						
	services_general_info			**db:free_text**								

Figure 1: Sample Worksheet. The Worksheet represents APIs. For example, the `Main` API is as follows: `Main(student_task: Enum["TroubleShoot", "Leave of Absence", "Test Credits"], troubleshoot: TroubleShoot, extra_details: str)`

5.2 Worksheets

A worksheet, as illustrated in Figure 1, has a name (WS Name), a predicate (WS Predicate) indicating when it is activated, and a set of fields. The top-level worksheet is unconditionally executed, and the rest of the worksheets are activated if their corresponding predicates are true. Each field has these attributes:

- **Predicate:** Python code for indicating whether the field should be active.
- **Input:** Whether it is an input or an internal value, the latter is computed rather than solicited from the user.
- **Type:** if the type is "Enum", the possible values are specified in the "Enum Values" field. Other types allowed are: str, int, bool, name of another worksheet, confirm.
- **Name:** the name of the field
- **Description:** a natural language description of the field.
- **Don't Ask:** if true, the agent saves the information if offered by the user, but does not solicit it. An example of such a field could be: "Is the user annoyed?". In this case, the system won't explicitly ask the user if they are annoyed; however, if the user mentions that they are, the value can be set to True.
- **Required:** if true, solicits the user for a value
- **Confirmation:** if true, confirms the value with the user, which is useful if an undesirable side effect can result from a mistaken value.
- **Actions:** code (in Python) to be executed whenever a value is assigned to the variable.

Design Guideline

When naming worksheets, fields, and enum values, use **clear and semantically meaningful labels**.

- Good: `departure_city`, `arrival_city`, `travel_date`
- Bad: `x1`, `argA`, `thing`

The performance of your agent depends heavily on these names and descriptions, since the LLM interprets them directly. Thoughtful naming makes the agent more reliable and easier to debug.

In the WS actions field, the developer can provide Python code to be executed when all the required fields are filled. Several built-in actions are provided to the developer: (1) `say (str)` responds to the user with the given string str. (2) `propose (ws, [fld,val]*)` instantiates a new worksheet `ws` with the given field value pairs as a Python dictionary.

Key Analogy

Think of a **worksheet** as a recipe:

- The **predicate** decides when to start cooking.
- The **fields** are your ingredients (some required, some optional).
- The **actions** are the steps that happen once you have everything ready.

Unlike an imperative recipe that says "chop onions, then heat oil, then add onions," a declarative worksheet just says "this dish requires onions, oil, and heat; once you have them, cook it." Genie fills in the details.

5.2.1 The Ticket Submission Example

Figure 1 shows an example of ticket submission taken from the Genie Worksheet paper. This agent reproduces a subset of three tasks using seven pages from Stanford's ServiceNow ticket submission portal.

The **Main** worksheet is defined with the `WS` type. The value `sample_ticket` under Name specifies the backend API name, which is automatically invoked with the same parameters as those defined in the worksheet. In this case, the function signature is `submit_ticket(student_task, trouble_shoot, extra_details)`.

The first field in the `Main` worksheet is `student_task`. Since it has no predicate, the value can be filled in at any point in the conversation. Its input kind is `Input`, meaning the value must be provided by the user. The type is `Enum`, with allowed values `["TroubleShoot", "Leave of Absence", "Test Credits"]`. A description is also provided to help the parser identify what this field represents. Finally, it is marked as required by setting its value to `True`.

The second field is `trouble_shoot`, which is used when a student encounters issues enrolling in a class, as described under the description column. However, the agent can only use this field if the student's task is identified as Troubleshooting. This condition is specified by the predicate `self.student_task == "TroubleShoot"`. The type of this field is another worksheet, namely `TroubleShoot`, meaning that in order to fill `trouble_shoot`, the corresponding `TroubleShoot` worksheet must also be completed.

The last field in the `Main` worksheet is `confirm`, which has the type `confirm`. This type ensures the user is always asked for confirmation before the worksheet action is executed. The confirm field also specifies a field action: if the user does not confirm, the agent responds with "Thank you, can I assist you in any other way?"

Finally, the `services_general_info` worksheet is shown. This is a knowledge worksheet as defined by `db` type. The special `:free_text` tell the system that its a vector database containing free-text documents.

5.3 Building the Ride Booking Agent with GenieWorksheets

Action Item 2

Learning Goal: Get hands-on experience with building task-oriented agents using GenieWorksheets.

Task 2: Implement a Ride Booking Agent with Genie Worksheets by following the provided Google Colab Notebook. The Ride Book Agent should follow the conversation logic provided in Figure 2 or, equivalently, the conversation flow provided in section 5.4.

Before implementing the Ride Booking Agent, make sure to take the time to understand GenieWorksheets. Read Sections 5.1 and 5.2 in this handout, look at the example of the Plane Booking Agent in the Google Colab, and refer to the following resources:

- GenieWorksheets paper
- Starter Worksheet Template
- Example Worksheets: Course Enrollment Agent
- Example Worksheets: Fidelity Investment Agent

After implementing the agent, interact with the agent and download the `conversation_log.json`.

Setup Instructions: For this assignment, we will be using Azure OpenAI to access different models. To claim your compute credit and access your API keys, please follow the instructions provided here.

5.4 Conversation Flow for Ride Booking Agent

Here's the conversation flow represented by the logic diagram in Figure 2.

Step 1: Request and Confirm (Inputs)
The system begins by collecting essential information:

- Customer name
- Departure location
- Arrival location

This establishes the base request.

Step 2: Query (Check)

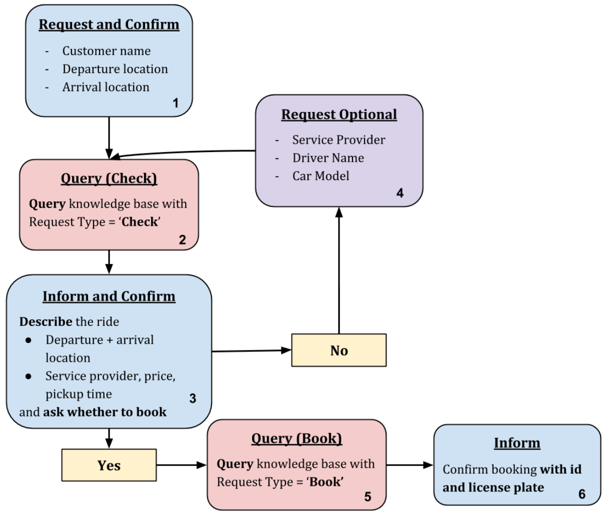

Figure 2: Conversation Logic for Ride Booking Agent

The system sends an api call to our `check_availability` api. This means it looks up possible ride options (service providers, driver name and car models).

Step 3: Inform and Confirm
The system describes the ride to the customer:

- Departure + arrival location
- Service provider, price, pickup time

Then it explicitly asks whether the customer wants to book.

Decision Point

- If the customer says Yes → proceed to booking (Step 5).
- If the customer says No → request optional details (Step 4).

Step 4: Request Optional
If the initial offer isn't accepted, the system may ask for optional preferences:

- Service Provider
- Driver Name
- Car Model

Then it loops back to Step 2 (Query Check) to refine ride options.

Step 5: Query (Book)

```
User: Hi
Agent: Hi! Could you please provide me with the name of the customer who wants to book the ride?
User: Its John Doe
Agent: Hi John! Could you please provide me with the departure location for the ride booking?
User: I am going from LA to SF
Agent: Thank you for providing the details, John. Since you are traveling from LA to SF,
can you let me know the date and time you would like to book the ride?
User: maybe tomorrow around 10am
Agent: Thank you for the information, Rishab. Let me confirm, you would like to book a ride from LA to SF
for tomorrow, October 1st, at around 10:00 AM?
User: Yes thats true
Agent: Your ride has been successfully booked. Here are the details:

- Status: Success
- Booking ID: 7a906279-cc47-4a36-b871-1ec119e80657

Thank you for choosing our service, Rishab! If you have any further questions or need assistance, feel free to ask.
Safe travels!
User: exit()
```

Table 1: A sample conversation with the Agent. Note that this simple conversation follows the "Happy Path". For your assignment, you will interact with the bot and change information midway, such as after selecting LA as the departure location, you can say "I am actually traveling from San Jose"

If the customer accepts, the system executes the `book_ride` api finalize the booking.

Step 6: Inform (Final Confirmation)
The system confirms the booking details back to the customer, including:

- Booking ID
- License plate

This concludes the flow.

In summary: The system first checks ride options (Step 1 → 2 → 3). If the customer refuses, it gathers more optional info and retries (Step 4 → 2). If the customer accepts, it proceeds to finalize booking (Step 5 → 6).

5.5 Evaluating the Ride Booking Agent

Action Item 3

Learning Goal: Get hands-on experience with evaluating a task-oriented agent.

Task 3: Read about evaluation in the GenieWorksheets paper and answer Gradescope Q2.1. Then, evaluate all of the turns in your `conversation_log.json` file and share your thoughts by updating the `conversation_log.json` file as specified in the Google Colab.

Once you finish the evaluation, complete the following:

- Upload the evaluated conversation_log.json file to (Gradescope Q2.2).
- Upload the Google Colab notebook as a PDF. In Google Colab, click File → Print → "Save as PDF" and upload the downloaded PDF file to (Gradescope Q2.3)
- Paste the link to the google spread sheet you created as a worksheet (Gradescope Q2.4)

References

Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H. Lin, Ilya Lintsbakh, Andy McGovern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. 2020. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.

Anthropic. 2024. Claude.

Paweł Budzianowski, Tsung-Hsien Wen, Bo-Hsiang Tseng, Iñigo Casanueva, Stefan Ultes, Osman Ramadan, and Milica Gašić. 2018. MultiWOZ - a large-scale multi-domain Wizard-of-Oz dataset for task-oriented dialogue modelling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5016–5026, Brussels, Belgium. Association for Computational Linguistics.

Gemini, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Harshit Joshi, Shicheng Liu, James Chen, Larsen Weigle, and Monica Lam. 2025. Controllable and reliable knowledge-intensive task-oriented conversational agents with declarative genie worksheets. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 27264–27308, Vienna, Austria. Association for Computational Linguistics.

OpenAI. 2024. Chatgpt.

Abhinav Rastogi, Xiaoxue Zang, Srinivas Sunkara, Raghav Gupta, and Pranav Khaitan. 2020. Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 8689–8696.