CoDial: Interpretable Task-Oriented Dialogue Systems Through Dialogue Flow Alignment

Anonymous ACL submission

Abstract

It is often challenging to integrate specialized, unseen tasks to dialogue systems due to the high cost of expert knowledge, training data, and high technical difficulty. To facilitate cross-disciplinary collaboration, it is then essential to build frameworks with a low cost of specification to enable adoption by experts from other domains. In this work, we introduce CoDial (Code for Dialogue), a novel framework that converts structured expert knowledge, represented as a heterogeneous graph, into conversation logic. We create an implementation of our framework with Colang, a guardrailing language that aligns language models without additional training. Empirically, our framework achieves new state-of-the-art performance on the STAR dataset for inference-based models and is competitive with similar baselines on the well-known MultiWOZ dataset. We also demonstrate how to further improve experimental results with user feedback. CoDial is an interpretable and modifiable framework for task-oriented dialogue that can specify conversation logic in a strict zero-shot setting. [1]

1 Introduction

Task-Oriented Dialogue (TOD) is dialogue where a chatbot collaborates with the user to accomplish real-world tasks (Qin et al., 2023). Creating generalizable conversational systems is challenging due to the complexity of human conversation, but task-oriented systems allow humans to guide the chatbot to a desirable goal (Jacqmin et al., 2022). Recently, Large Language Models (LLMs) have revolutionized the field of Natural Language Processing (NLP), achieving remarkable results on a wide range of NLP benchmarks. As language models become more widely adopted in society, many experts in other disciplines and industries wish to adopt the technology to their specific tasks (Tian et al., 2024; Dahan et al., 2023). These experts might have limited knowledge of prompting and programming, which necessitates frameworks that enable generalization to a wide variety of tasks. Through these general frameworks, it is easier to build intelligent dialogue systems for tasks from various domains to enable efficient inter-disciplinary collaboration and quickly incorporate expert feedback.

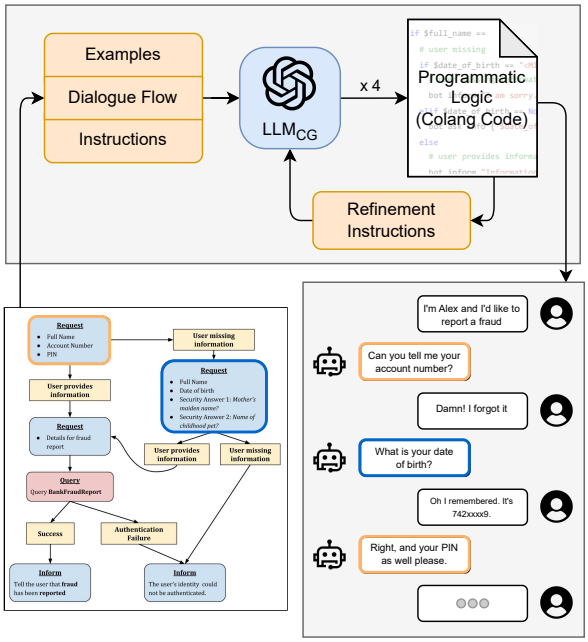


Figure 1: Overview of the proposed CoDial framework. An expert-curated dialogue flow (left) is transformed into executable programmatic logic using an LLM (top). The generated code is iteratively refined before producing the final program, which powers a conversational application (right), enabling the chatbot to follow the designer's requirements.

Many existing works utilize a schema-oriented framework to enforce more complex task structures (Zhang et al., 2023; Zhao et al., 2023; Mehri and Eskenazi, 2021). These systems convert tasks into a parsable *task schema*, allowing experts to specify and integrate new tasks with minimal effort. However, previous works do not satisfy two

[1] All of our code and data will be made publicly available at [GitHub Placeholder].

properties: i) **interpretability**, the ability to identify how the schema is being used by the LM to arrive at its outputs, and ii) the ability for an expert to **modify** the components after generating the task schema. Interpretability is generally desirable in multi-disciplinary systems — experts want to know when the model's outputs can be trusted even if they do not understand the mechanisms (Masud et al., 2024). The programmatic representation in AnyTOD, for example, is interpretable but difficult for a non-expert to modify. To facilitate inter-disciplinary collaboration, it is important to create a framework that is easy for an expert to specify, validate, and modify.

Our contributions can be summarized as follows:

- We propose CoDial, a novel framework to define and convert structured knowledge as heterogeneous graphs with conditional edges into programmatic logic. This framework enables effective model alignment with minimal human annotation effort, facilitating model guidance without requiring expertise in LM training.

- We implement our framework in the context of general-domain TOD systems by converting dialogue flows into Colang, a programmatic guardrailing language. We are the first to apply guardrailing for TOD. Our implementation provides a practical solution that enhances interpretability and modifiability compared to existing zero-shot TOD approaches.

- We demonstrate the effectiveness of our approach on the STAR and MultiWOZ datasets. Our method achieves new state-of-the-art (SOTA) results for STAR in zero-shot inference-based settings. Additionally, we show the efficacy of various code refinement strategies, using user feedback for manual and LLM-aided adjustments.

2 Related Work

Task-Oriented Dialogue Creating generalizable conversational systems is challenging due to the complexity of human conversation, especially when involving expertise from other disciplines (Chen et al., 2017). Louie et al. (2024) proposed a system that allows experts to create simulated patients for training mental health counselors via writing natural language principles, so experts can help direct the training without any technical background. There is also interest in using dialogue systems to assist in completing a task, also known as TOD (Jacqmin et al., 2022).

While Large Language Models (LLMs) have demonstrated impressive capability in a wide variety of domains, LLMs struggle with TOD and LLMs fall behind if not used properly (Hudeček and Dusek, 2023). Some works (Zhang et al., 2023; Zhao et al., 2023; Mehri and Eskenazi, 2021) use a schema-guided approach to generalize TOD systems to unseen tasks. Zhao et al. (2023) views the task schema as a program and adopts a neuro-symbolic approach to execute the policy program and control the dialogue flow.

Guardrails Guardrails aim to filter the inputs and outputs of LLMs to control their behaviour and mitigate harm (Rebedea et al., 2023; Dong et al., 2024a). Dong et al. (2024b) argues that a neuro-symbolic approach is required to guardrail LLMs, where a neural agent (e.g., an LLM) deals with frequently-seen cases and a symbolic agent can embed human-like cognition through structured knowledge for the rare cases. NeMo-Guardrails (Rebedea et al., 2023) is a toolkit to add programmable guardrails to LLM-based conversational applications without fine-tuning. NeMo-Guardrails employs Colang[2], an executable program language, to establish constraints and guide LLMs within the defined dialogical boundaries.

Code Generation and Prompt Optimization We use code generation strategies to convert our structured graphs into programmatic guardrails. Code generation has made remarkable progress with the introduction of LLMs (Le et al., 2022). While there are still many challenges with logical consistency and hallucinations (Liu et al., 2024), LLMs are proficient at generating code when provided in-context examples, documentation, and a plan (Jiang et al., 2024). There are many emerging methods to further optimize LLM generations — self-reflection, where the LLM is asked to update its own response, has been shown to reduce hallucinations and improve problem solving capabilities (Ji et al., 2023). There are also works to improve the output by rewriting the input prompt, referred to as prompt optimization (Yang et al., 2023; Yuksekgonul et al., 2024).

[2] https://docs.nvidia.com/nemo/guardrails/colang_2/overview.html

3 Methodology

We propose CoDial, a framework for building interpretable dialogue systems without requiring training data or programming expertise, illustrated in Figure 1. While we apply this framework for TOD, it can be extended to any dialogue. In CoDial, an expert defines the conversation logic through a novel task schema paradigm, called a *dialogue flow*. CoDial then generates an executable Colang code, a programming language for LLM guardrailing, based on the input dialogue flow.

3.1 Dialogue Flows Framework

We design a novel paradigm to represent a dialogue flow for conversational systems. We adopt a graph-based approach, similar to Mosig et al. (2020), but we are the first to frame TOD as a heterogeneous directed graph with conditional edges. We define three available node types:

Request This node type defines the variables, called slots, that CoDial tracks throughout the conversation (e.g., the departure location in a taxi booking task). When a conversation reaches this node, the system requests the required information specified by the node. Each slot is assigned a type (e.g., categorical) and accompanied by a few example values. Additionally, we include a free-form `rule` property to define the conditions under which a slot should be requested (e.g., in a taxi booking scenario, providing either a departure or arrival time is sufficient for booking). Since we leverage LLMs to build the TOD system, textual extensions can be easily incorporated.

External Action This node specifies a call to a Python function (referred to as Actions in Colang) within a dialogue flow. External actions enable the designer to execute complex logics through programming functions, interact with APIs, or invoke an LLM.

Inform (and Confirm) This node defines a template for providing information to the user (e.g., *Your taxi is booked with reference number [ref_no]*). The confirmation variant additionally allows the chatbot to ask a follow-up question (e.g., *Do you confirm the booking?*) and follow the appropriate predefined dialogue path based on the user's response.

Global Actions In addition to nodes, we support global actions that can be triggered from any point in the dialogue flow, independent of a specific node. These actions enable *fallback actions*, general responses such as *anything-else* or *out-of-scope*, that are not tied to a particular dialogue step.

The defined nodes logically connect to each other with **edges**. We add a textual `condition` property to edges to allow conditional branching in dialogue flows. Additionally, we support global actions that can be triggered from any point in the dialogue flow, independent of a specific node. These actions enable actions such as *anything-else* or *out-of-scope* actions that are not tied to a particular dialogue step. The crafted dialogue flow is the *only* thing that the designer needs to provide.

We represent the graphs as text in JSON format. The JSON representation consists of a list of nodes and a list of edges. The node list defines the dialogue flow nodes, specifying their types and assigning each a unique identifier (node ID). The edge list specifies the connections between nodes using their IDs (Figure 7). The JSON nodes, global actions, as well as functional specifications for API calls are translated into Colang code with our automatic code generation pipeline.

Algorithm 1 Our generated logic main flow.

for each v in $V^{(H)}$ **do**
  $v \leftarrow$ NULL or FALSE ▷ Initialize helpers
end for
while True **do**
  $h_{2i-1}^{(u-b)} \leftarrow \left(h_{2i-2}^{(u-b)}; \text{utt}_{2i-1}^{(u)}\right)$ ▷ User input
  `intent` $\leftarrow$ DetectIntent($h_{2i-1}^{(u-b)}$)
  if `intent` $\neq$ NULL **then**
    bot say IntentResponse(`intent`)
    continue
  end if
  for each v in $V^{(S)}$ **do** ▷ DST
    $v_{old} \leftarrow v$
    $v_j^{(s)} = \text{DST}\left(h_{2i-1}^{(u-b)}, \text{LLM}_\text{A}, p_j^{(s)}\right)$
    if $v \neq v_{old}$ **then**
      for each hv in v's dependents **do**
        $hv \leftarrow$ NULL or FALSE
      end for
    end if
  end for
  `state` $\leftarrow (V^{(S)}, V^{(H)})$ ▷ NAP
  `action`, $V^{(H)} \leftarrow$ NAP(`state`)
  `output` $\leftarrow \text{LLM}_\text{A}$(`action`, `state`)
  if `output` = NULL **then**
    `output` $\leftarrow \text{LLM}_\text{A}(V^{(H)})$
  end if
  bot say `output`
end while

3.2 Automatic Code Generation for Guardrails

We use LLM prompting to convert the input dialogue flow into Colang[3]. As a programmatic guardrailing tool, Colang allows us to adjust model behaviour without fine-tuning. The prompt for code generation ($\text{prompt}_{\text{CG}}$) instructs an LLM ($\text{LLM}_{\text{CG}}$) to generate Colang code with two components: Dialogue State Tracking (DST) and Next Action Prediction (NAP), which are explained below. Note that DST and NAP are combined and generated as a single Colang program. Figure 2 shows an overview of $\text{prompt}_{\text{CG}}$.

Algorithm 1 presents the general structure of the dialogue flows programmatic logic. The conversation runs within an indefinite while loop, where the chatbot waits for user input, detects the user's intent for global actions, predicts the slot variables defined in request nodes (DST component), and selects an action and generates a response (NAP component). Finally, if the NAP component does not generate a response based on the defined logic, the agent's LLM (LLM_{A}) chooses a fallback action instead. Figure 3 illustrates the execution life cycle of the agent.

We denote a conversation between user u and chatbot b as a history of messages, Equation 1, where $\text{utt}_{2i-1}^{(u)}$ and $\text{utt}_{2i}^{(b)}$, show user's and chatbot's i-th utterance, respectively. Therefore, the total number of utterances in $h_{2i}^{(u-b)}$ is $2i$.

$$h_{2i}^{(u-b)} = \left(\text{utt}_1^{(u)}, \text{utt}_2^{(b)}, \ldots, \text{utt}_{2i-1}^{(u)}, \text{utt}_{2i}^{(b)}\right) \quad (1)$$

Moreover, we define a set of slot variables $V^{(S)}$ that track values for all of the slots defined in all request nodes, and helper variables $V^{(H)}$ that track the state for other types of nodes. $V^{(S)}$ and $V^{(H)}$ together form the state of conversation $s = \left(V^{(S)}; V^{(H)}\right)$ at each turn, which is used to determine the next action. Please refer to Appendix A.1 for more details on code generation.

Dialogue State Tracking (DST) As suggested by Feng et al. (2023), LLM prompting shows promising performance in DST. Therefore, we leverage a simple prompting approach using Colang's Natural Language Description (NLD) feature to set the value of each slot from the conversation history. For each slot, we instruct LLM_{CG} to write instructions about extracting the value for that slot, given the history. The NLD instructions are then converted into a prompt by Colang.

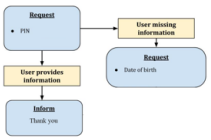
Formally, a slot variable $v_j^{(s)} \in V^{(S)}$ is predicted at bot's turn i as Equation 2, where $p_j^{(s)} \in P^{(s)}$ is the prompt generated by LLM_{CG} to extract the value for $v_j^{(s)}$.

$$v_j^{(s)} = \text{DST}\left(h_{2i-1}^{(u-b)}, \text{LLM}_{\text{A}}, p_j^{(s)}\right) \quad (2)$$

Since updating a slot may affect the state (e.g., in

=== DST in Colang Example

[Simple Dialogue Flow]



Equivalent DST Code

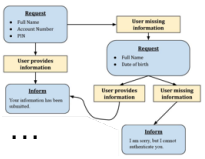
```
$pin = dst [...] $pin "extract user pin from conversation history ..."
$dob = dst [...] $dob "extract user date of birth from conversation history ..."
```

=== Colang Action Calling Definition

```
<ActionName>[(param=<value>[, param=<value>]…)]
```

=== NAP in Colang Example

[Simple Dialogue Flow]



Equivalent NAP Code

```
if $full_name == "<MISSING>" or ...
  # user missing information
  if $date_of_birth == "<MISSING>"
    # user missing information
    bot inform "I cannot authenticate you."
  elif ...
else
  # user provides information
  bot inform "Your information has been submitted"
```

=== Dialogue Flow

[Input Dialogue Flow]

=== Task Description

Define a variable for each request slot ...

Convert graph into nested if/else statements ...
For request nodes, ...
For external_action nodes, ...
For inform nodes, ...
For inform_and_confirm nodes, ...

Figure 2: Overview of $\text{prompt}_{\text{CG}}$. We provide two simplified dialogue flow examples alongside their corresponding Colang snippets, the Colang Python action calling definition (from the official documentation), the input dialogue flow, and instructions for converting a dialogue flow into a Colang program.

[3]We also experimented with i) retrieval-augmented generation using the Colang Language Reference documentation and ii) fine-tuning GPT-4o-mini on generation pairs of (programming task, Colang code), but found that prompting with examples works best.

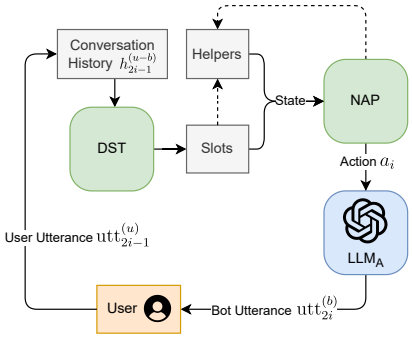


Figure 3: Execution life cycle of the generated agent

a search task, modifying the search criteria requires re-executing the search), LLM$_{CG}$ needs to identify the helper variables that need to be invalidated when each slot is updated. We instruct LLM$_{CG}$ to list the helper variables of nodes that are reachable from the updated slot in the graph (i.e., nodes that are direct or indirect children of the slot's request node). These variables are then reset to null or false, depending on their type.

Next Action Prediction (NAP) We instruct LLM$_{CG}$ to convert the dialogue flow tree into a conditional logic, consisting of nested if/else statements, to generate a response given the current state. For each node n_j in the dialogue flow, an if statement is generated to check whether the conversation is in that node, using $v_j^{(s)}$ or $v_j^{(h)}$, depending on the type of the node. If the condition holds, the corresponding action for that node is executed; otherwise, the logic proceeds to check its child nodes. For each outgoing edge from a node, the dialogue logic checks whether there is a condition associated with the edge and evaluates whether or not the condition is met. If there is no condition, the edge is followed automatically. Formally, next action and next bot utterance are defined in Equations 3 and 4, respectively.

$$\left(a_i, V_{i+1}^{(H)}\right) = \text{NAP}(s_i) \tag{3}$$

$$\text{utt}_{2i}^{(b)} = \text{LLM}_\text{A}\left(a_i, s_i\right) \tag{4}$$

3.3 Modifications with Human Feedback

One advantage of our system is our ability to make modifications and improvements with human feedback. We experiment with two methods of improving the pipeline: 1) manual corrections to the dialogue flow and 2) LLM-aided corrections to the generated code. Our detailed manual modifications can be found in Appendix A.2.

During initial experiments, we observed that LLM$_{CG}$ frequently made errors in generating correct logic (i.e., if statement) for each node, DST initialization, and request node checks. To address this, we apply three hand-crafted **refinement instructions (RI)**—simpler prompts to iteratively improve the code—presented in Table 6. The ability to refine the generated code highlights the interpretability of the approach, and we refine the code with natural language feedback that could be written by a domain expert. Additionally, we attempt to refine the DST extraction prompts ($P^{(s)}$) with prompt optimization. Please refer to Appendix A.3 for more details.

4 Experimental Settings

Models For evaluation, we use GPT-4o[4], Claude 3.5 Sonnet[5], Gemini 2.0 Flash[6], and DeepSeek V3 (DSV3) (DeepSeek-AI et al., 2024) as LLM$_{CG}$, and GPT-4o-mini and DSV3 as LLM$_A$[7] . We generally use larger models for code generation, as it is a more complex task, and we found that smaller models often fail to fully adhere to instructions. For further details, please refer to Appendix A.4.

NeMo-Guardrails To implement the Colang guardrails, we use a fork from NeMo-Guardrails version 0.11, modified to inject our evaluator class. We use this class to evaluate on the ground truth history ($h_{2i-1}^{(u-w)}$), instead of the history of user-bot conversation ($h_{2i-1}^{(u-b)}$), similar to Nekvinda and Dušek (2021). For a detailed list of modifications, refer to Appendix A.5[8].

4.1 Datasets

STAR The STAR dataset (Mosig et al., 2020), collected in a Wizard-of-Oz setup (human-human conversations), provides **explicit task schemas** (i.e., dialogue flows) to ensure consistent and deterministic system actions. It serves as a benchmark for TOD systems, enabling evaluation across 24 tasks and 13 domains. STAR's structured collection aligns well with our objectives and CoDial's design choices. We also use silver state annotations created in STARv2 (Zhao et al., 2023) for

[4] https://openai.com/index/hello-gpt-4o/

[5] https://www.anthropic.com/news/claude-3-5-sonnet

[6] https://developers.googleblog.com/en/gemini-2-family-expands/

[7] All models were accessed in January and February 2025.

[8] The modified NeMo-Guardrails version that we used for the experiments is available at [GitHub Placeholder].

ablation studies. Refer to Appendix A.2 for more implementation details on STAR.

MultiWOZ MultiWOZ (Budzianowski et al., 2018) is a large-scale, multi-domain TOD dataset consisting of human-human conversations, with most domains involving booking subtasks such as hotel reservations and taxi services. Given the impracticality of crafting dialogue flows for every possible domain combination, as mentioned in previous work (Zhang et al., 2023), we report results in a naive oracle domain setting. We preprocess MultiWOZ 2.2 using the code from Li et al. (2024) to annotate each conversation turn with its active domains. For each turn i, we use the dialogue flow(s) of the corresponding domain(s) to predict the output and merge all turns at the end. Please refer to Appendix A.3 for more details.

4.2 Metrics

For the STAR dataset, we compute BLEU-4 score (Papineni et al., 2002) using SacreBLEU (Post, 2018). We also follow Mosig et al. (2020) to compute F1 and accuracy. Since we applied STAR's response templates for response generation, we use regex patterns to match generated responses with actual values to a template. For the MultiWOZ dataset, we compute BLEU, Inform and Success rates, and Joint Goal Accuracy (JGA) using the official evaluation script (Nekvinda and Dušek, 2021). We report the mean result of three runs.

4.3 Baselines

For a complete list of compared methods, please refer to Appendix A.6. Our most comparable baselines are as follows:

- *AnyTOD* (Zhao et al., 2023) pretrains and finetunes T5-XXL for DST and response generation. It views task schema as a Python program to enforce the complicated conversation logic to guide the LM decisions.
- *IG-TOD* (Hudeček and Dusek, 2023) is a prompting-based approach using ChatGPT to track dialogue states via slot descriptions, retrieve database entries, and generate responses without fine-tuning.
- *SGP-TOD* (Zhang et al., 2023) is a zero-shot approach that uses two-stage prompting to track dialogue state and generate response. It employs graph-based dialogue flows to steer LLM actions with out-of-domain formatting examples.

5 Results

5.1 Results on STAR

SOTA Performance Without Training. Table 1 summarizes our results on the STAR dataset. CoDial achieves strong performance, surpassing all training-based approaches except AnyTOD PROG+SGD XXL, and sets the new SOTA among inference-based methods. Our framework improves F1 by **+5** and accuracy by **+6.9** points over the previous SOTA. While AnyTOD PROG+SGD XXL achieves higher scores, it requires extensive pretraining with task-specific data and manually written dialogue logic programs, making it less accessible to non-programming experts. In contrast, CoDial operates in a strict zero-shot setting, eliminating the need for training and manual programming.

Without modifications, the original STAR dialogue flows result in lower performance (F1: 52.8), underscoring the need for dialogue flows to precisely reflect the intended TOD behaviour. We apply manual refinements to the dialogue flow and LLM-aided modifications to the generated code, which significantly enhance performance. We further explore the impact of LLM-aided corrections in Section 5.3.

Impact of Model Selection We experience with different model choices for the (LLM_{CG}, LLM_{A}) pairing. Better instruction following and more robust code generation often translate to higher overall performance. Because CoDial's approach requires generating Colang, an unfamiliar language for most models, LLMs must accurately interpret prompts and produce syntactically correct code. When the chosen LLM struggles with instruction following, code generation can fail, leading to incorrect or incomplete programs. Among the tested configurations, CoDial (4O, 4O-MINI) achieves the highest performance in all metrics.

Additionally, we try a setting with DSV3 as LLM_{A}. Although we see a drop in performance, we observe that in some tasks, such as *Bank Fraud Report*, DSV3 performs better. Therefore, we also report results in an oracle voting setting (Table 4) between GPT-4o-mini and DSV3 as LLM_{A}, where for each task, we take the best-performing LLM_{A} by F1. This results in an increase of $+1.7$ and $+1.5$ points in F1 and accuracy, respectively.

Action Prediction and API Calls To gain deeper insights into CoDial's performance, we conduct a

Model	Approach	LLM_{CG}	LLM_A	F1	Acc.	BLEU
Training-based Approaches				*Domain Transfer*		
BERT + Schema	Fine-tuning	-	-	29.7	32.4	-
SAM	Fine-tuning	-	-	51.2	49.8	-
AnyTOD NOREC BASE	Fine-tuning	-	-	55.8	56.1	32.4
AnyTOD PROG+SGD XXL	Pretraining	-	-	70.7	70.8	44.2
Inference-based Approaches				*Domain Transfer*		
SGP-TOD-GPT3.5-E2E	Zero-shot	-	-	53.5	53.2	-
CoDial (Ours)				*Strict Zero-Shot*		
CoDial ORIGINAL DFS	Zero-shot	4o	4o-mini	51.9	51.1	38.9
CoDial − RI	Zero-shot	4o	4o-mini	56.1	57.3	38.4
CoDial − RI	Zero-shot	Sonnet	4o-mini	57.0	58.4	39.2
CoDial	Zero-shot	DSV3	4o-mini	46.1	48.0	28.0
CoDial*	Zero-shot	Gem. 2 Fl.	4o-mini	50.5	52.1	32.9
CoDial	Zero-shot	Sonnet	4o-mini	57.7	58.5	39.3
CoDial	Zero-shot	4o	DSV3	55.6	56.8	44.2
CoDial	Zero-shot	4o	4o-mini	**58.5**	**60.1**	**45.2**

Table 1: Comparison of models on the STAR dataset. Our CoDial model achieves the SOTA in a "Strict Zero-Shot" setting, where we do not require any training samples or sample conversations. SAM results are cited from Zhao et al. (2023). The generated code for the model with an asterisk (*) has been manually fixed and is not directly comparable. DF stands for "dialogue flow."

Actions	F1	Acc.
Intent Detection (Global Actions)		
All	96.3	92.8
LLM Generated Actions		
Fallbacks	51.4	57.8
Excluding Fallbacks	38.7	39.0
All	49.1	52.1

Table 2: Individual action prediction performance of intent detection and LLM_A in CoDial. Fallback actions include `goodbye`, `out_of_scope`, and `anything_else`. All entries are micro-averaged.

Model	JGA	Inform	Success	BLEU	Combined
Training-based Approaches					
SOLOIST	35.9	81.7	67.1	13.6	88.0
MARS	35.5	88.9	78.0	19.6	103.0
AnyTOD XXL	30.8	76.9	47.6	3.4	65.6
Inference-based Approaches					
IG-TOD (fs)	27	-	44	6.8	-
SGP-TOD	-	82.0	72.5	9.2	86.5
CoDial	27.8	76.3	53.9	3.6	68.7

Table 3: Comparison of models on the MultiWOZ dataset. SOLOIST and MARS results are cited from Zhao et al. (2023). (*fs*) indicates few-shot.

detailed analysis of the results. We find that NeMo Guardrails' intent detection performs strongly, achieving an F1 score of 96.3 (Table 2). Additionally, we observe that STAR's API calling precision—measured as the ratio of correct API calls to the total number of API calls—stands at 74.9. Table 2 also summarizes the performance of the actions that are generated by LLM_A (i.e., when NAP component does not generate an output). LLM-generated actions account for 25% of all predicted actions, with 70% of them belonging to three fallback actions: `goodbye`, `out_of_scope`, and `anything_else`. Excluding fallback actions, LLM-generated actions only account for 9.2% of the predicted actions, indicating that the NAP logic is generally effective at generating outputs based on the representative state for dialogue. Since predicting fallback actions is a simple 3-way classification, we would expect high performance. However, LLM_A achieves an F1 score of only 51.4. We attribute this to the lack of an explicit schema for fallback actions in the STAR dataset, leading to inconsistencies in how wizards annotate them. Additionally, we observe a significant performance drop from fallback to non-fallback actions in both F1 ($51.4 \rightarrow 38.7$) and accuracy. This suggests that despite having an explicit schema, LLMs struggle to capture the more complex logic required for correctly predicting non-fallback actions. Our findings align with Dong et al. (2024b), reinforcing the need for a neural-symbolic approach.

5.2 Results on MultiWOZ

Our results on the MultiWOZ dataset are summarized in Table 3. Unlike STAR, where wizards were provided structured guidance for system responses, MultiWOZ lacks a predefined dialogue flow, making interactions less consistent. This variability in MultiWOZ poses additional challenges for rule-based and programmatic approaches like CoDial. Our most comparable system is AnyTOD, which also has a strict programmatic schema and

Model	F1	Acc	BLEU
Refinement Instructions (RI)			
CoDial	58.5	60.1	**45.2**
– RI 3	56.1	58.0	41.6
– RI 3 & 2	55.9	57.6	41.7
– RI 3 & 2 & 1	56.1	57.3	38.4
Oracle Vote			
CoDial (DST 4o-mini + DSV3)	60.2	61.6	46.8
Silver Label DST			
CoDial + STARv2 States	**60.7**	**62.9**	44.3

Table 4: Ablations on the STAR dataset.

Model	JGA	Inform	Success	BLEU	Combined
Predicted Belief State					
IG-TOD (fs)	27	-	44	6.8	-
CoDial SINGLE*	46.2	91.5	77.6	3.2	87.7
CoDial	27.1	74.3	54.4	3.6	67.9
Oracle Belief State					
IG-TOD (fs)	-	-	68	6.8	-
CoDial SINGLE*	-	94.6	90.6	3.5	96.1
CoDial	-	93.1	75.3	4.0	88.2
DST Prompt Optimization					
CoDial	27.1	74.3	54.4	3.6	67.9

Table 5: Ablations on MultiWOZ. (fs) indicates few-shot.

generates output text with templates. We achieve comparable performance without pre-training.

Most of the MultiWOZ test set consists of multi-domain conversations, where a user may, for example, book both a taxi and a restaurant in the same dialogue. Since CoDial is designed for single-domain interactions, we report its performance on single-domain dialogues in Table 5, where it performs well. However, when applied naively to multi-domain settings by switching dialogue flows based on the active domain, performance drops significantly. We suspect this is due to compounded errors from DST to NAP.

5.3 Ablations

Oracle DST Performance To assess the impact of DST, we evaluate CoDial under an Oracle DST setting. Since STAR does not provide gold DST labels, we simulate an oracle setting by replacing CoDial's DST component with the silver-annotations from STARv2 (Table 4). This results in a performance gain of $+2.2$ F1 and $+2.8$ accuracy. We do the same for MultiWOZ, where we replace our predictions with the gold belief state. As shown in Table 5, this leads to a substantial performance improvement, making CoDial competitive with other baselines. A similar trend was seen in previous works like IG-TOD (Hudeček and Dusek, 2023). These findings suggest exploring more advanced DST approaches could be a promising direction to improve performance, regardless of the dataset. While we experiment briefly with prompt optimization for our framework, described below, we also urge further exploration in this direction.

Code Optimization We use LLMs to perform iterative code refinement as well as automatic prompt optimization for the DST prompts. Refining the code with RIs consistently enhances CoDial's performance, demonstrating the benefits of integrating user feedback into the generation process. By prompting the LLM to iteratively refine its outputs, CoDial achieves better accuracy and fluency compared to not using RI (CoDial − RI in Table 1). This highlights the system's ability to improve through human feedback. We also conduct an ablation study to examine the effect of the used RIs individually, summarized in Table 4. Although all RIs are helpful, most of the performance improvements can be attributed to the third RI, which refines the conditional logic of request nodes.

After obtaining the Oracle DST results, we attempt prompt optimization to improve DST. As shown in Table 5, the performance minimally decreases for every metric except Success. This indicates improving the DST is a non-trivial task that could be impacted by our naive multi-domain evaluation setup. For example, part of the DST prompt instructs to update the slots "based on the last user input" in case the user's requirements change throughout the conversation, but the update could be found in an earlier message corresponding to a different domain.

6 Conclusion

In this work, we introduced CoDial, a novel framework for building interpretable TOD systems with minimal human effort, eliminating the need for programming expertise. To create a dialogue system for a specific task, an expert simply designs a dialogue flow, and CoDial converts it into a fully functional conversational application. This approach ensures modifiability and interpretability, allowing experts to build TOD systems effortlessly while maintaining control over system behaviour. Our evaluations on STAR and MultiWOZ demonstrate CoDial's effectiveness, achieving SOTA performance among inference-based methods on STAR and competitive results on MultiWOZ.

Limitations

While CoDial offers an interpretable and modifiable approach to TOD systems, it has certain limitations. First, scalability remains a challenge. For large and complex dialogue flows, CoDial requeries all slots every turn, which may increase latency and computational cost. In general, the DST is the biggest challenge of the system and we leave improvements to future work. Second, CoDial is less effective for multi-domain dialogues, as it operates on a single dialogue flow at a time. Handling seamless transitions between multiple domains would require additional mechanisms beyond the current framework, and structured logic on how different domains flow into each other, which we leave to future work. In general, dialogue flows assume there is a logical progression to conversations, which isn't the case for all dialogue structures. Finally, reproducibility could be a concern when relied on API-based LLMs. Since these models are periodically updated, responses may vary across different API versions, making consistent evaluation challenging.

Ethics Statement

This work adheres to ethical research practices by ensuring that all models, codebases, and datasets used comply with their respective licenses and terms of use. The STAR and MultiWOZ datasets employed in our experiments do not contain personally identifiable information or offensive content.

As with any system leveraging LLMs, CoDial inherits potential risks related to bias and factually incorrect outputs. However, our framework mitigates these risks by enforcing structured dialogue flows, guardrailing based on user intent, and template-based responses, reducing the likelihood of hallucinated or biased content. Future work may integrate NeMo Guardrails' input and output rails to filter inappropriate inputs and outputs, enhancing system safety. Since our focus is on structured dialogue flows, we leave this for future exploration.

References

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.

Hongshen Chen, Xiaorui Liu, Dawei Yin, and Jiliang Tang. 2017. A survey on dialogue systems: Recent advances and new frontiers. *Acm Sigkdd Explorations Newsletter*, 19(2):25–35.

Samuel Dahan, Rohan Bhambhoria, David Liang, and Xiaodan Zhu. 2023. Lawyers should not trust ai: A call for an open-source legal language model. *Available at SSRN 4587092*.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Yi Dong, Ronghui Mu, Gaojie Jin, Yi Qi, Jinwei Hu, Xingyu Zhao, Jie Meng, Wenjie Ruan, and Xiaowei Huang. 2024a. Building guardrails for large language models. *arXiv preprint arXiv:2402.01822*.

Yi Dong, Ronghui Mu, Gaojie Jin, Yi Qi, Jinwei Hu, Xingyu Zhao, Jie Meng, Wenjie Ruan, and Xiaowei Huang. 2024b. Building guardrails for large language models. *Preprint*, arXiv:2402.01822.

Yujie Feng, Zexin Lu, Bo Liu, Liming Zhan, and Xiao-Ming Wu. 2023. Towards LLM-driven dialogue state tracking. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 739–755, Singapore. Association for Computational Linguistics.

Vojtěch Hudeček and Ondrej Dusek. 2023. Are large language models all you need for task-oriented dialogue? In *Proceedings of the 24th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 216–228, Prague, Czechia. Association for Computational Linguistics.

Léo Jacqmin, Lina M. Rojas Barahona, and Benoit Favre. 2022. "do you follow me?": A survey of recent approaches in dialogue state tracking. In *Proceedings of the 23rd Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 336–350, Edinburgh, UK. Association for Computational Linguistics.

Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. 2023. Towards mitigating llm hallucination via self reflection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1827–1843.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pages 21314–21328. Curran Associates, Inc.

Zekun Li, Zhiyu Chen, Mike Ross, Patrick Huber, Seungwhan Moon, Zhaojiang Lin, Xin Dong, Adithya Sagar, Xifeng Yan, and Paul Crook. 2024. Large language models as zero-shot dialogue state tracker through function calling. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8688–8704, Bangkok, Thailand. Association for Computational Linguistics.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Ryan Louie, Ananjan Nandi, William Fang, Cheng Chang, Emma Brunskill, and Diyi Yang. 2024. Roleplay-doh: Enabling domain-experts to create llm-simulated patients via eliciting and adhering to principles. *Preprint*, arXiv:2407.00870.

Sarah Masud, Sahajpreet Singh, Viktor Hangya, Alexander Fraser, and Tanmoy Chakraborty. 2024. Hate personified: Investigating the role of LLMs in content moderation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 15847–15863, Miami, Florida, USA. Association for Computational Linguistics.

Shikib Mehri and Maxine Eskenazi. 2021. Schema-guided paradigm for zero-shot dialog. In *Proceedings of the 22nd Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 499–508, Singapore and Online. Association for Computational Linguistics.

Johannes EM Mosig, Shikib Mehri, and Thomas Kober. 2020. Star: A schema-guided dialog dataset for transfer learning. *arXiv preprint arXiv:2010.11853*.

Tomáš Nekvinda and Ondřej Dušek. 2021. Shades of BLEU, flavours of success: The case of MultiWOZ. In *Proceedings of the 1st Workshop on Natural Language Generation, Evaluation, and Metrics (GEM 2021)*, pages 34–46, Online. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA. Association for Computational Linguistics.

Baolin Peng, Chunyuan Li, Jinchao Li, Shahin Shayandeh, Lars Liden, and Jianfeng Gao. 2021. Soloist: Building task bots at scale with transfer learning and machine teaching. *Transactions of the Association for Computational Linguistics*, 9:807–824.

Matt Post. 2018. A call for clarity in reporting BLEU scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 186–191, Belgium, Brussels. Association for Computational Linguistics.

Libo Qin, Wenbo Pan, Qiguang Chen, Lizi Liao, Zhou Yu, Yue Zhang, Wanxiang Che, and Min Li. 2023. End-to-end task-oriented dialogue: A survey of tasks, methods, and future directions. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5925–5941, Singapore. Association for Computational Linguistics.

Traian Rebedea, Razvan Dinu, Makesh Narsimhan Sreedhar, Christopher Parisien, and Jonathan Cohen. 2023. NeMo guardrails: A toolkit for controllable and safe LLM applications with programmable rails. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 431–445, Singapore. Association for Computational Linguistics.

Haipeng Sun, Junwei Bao, Youzheng Wu, and Xiaodong He. 2023. Mars: Modeling context & state representations with contrastive learning for end-to-end task-oriented dialog. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 11139–11160, Toronto, Canada. Association for Computational Linguistics.

Shubo Tian, Qiao Jin, Lana Yeganova, Po-Ting Lai, Qingqing Zhu, Xiuying Chen, Yifan Yang, Qingyu Chen, Won Kim, Donald C Comeau, et al. 2024. Opportunities and challenges for chatgpt and large language models in biomedicine and health. *Briefings in Bioinformatics*, 25(1):bbad493.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2023. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic "differentiation" via text.

Xiaoying Zhang, Baolin Peng, Kun Li, Jingyan Zhou, and Helen Meng. 2023. SGP-TOD: Building task bots effortlessly via schema-guided LLM prompting. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 13348–13369, Singapore. Association for Computational Linguistics.

Jeffrey Zhao, Yuan Cao, Raghav Gupta, Harrison Lee, Abhinav Rastogi, Mingqiu Wang, Hagen Soltau, Izhak Shafran, and Yonghui Wu. 2023. AnyTOD: A programmable task-oriented dialog system. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 16189–16204, Singapore. Association for Computational Linguistics.

A Appendix

A.1 Details on Code Generation

Post-processing Following code generation by the LLM, we apply rule-based post-processing to ensure proper execution. This includes adding

ID	Description	Instruction
RI 1	Revise if statements	Revise the 'if's to exactly reflect the nodes. Comment each 'if' to specify the corresponding node ID. Make sure the generated 'if' statement and its body reflect the instructions for that node type.
RI 2	Fix dst dependent vars	Fix dst's first input parameter. It should reflect which variables should be invalidated when the corresponding slot is updated.
RI 3	Fix request node checks	Fix 'if' checks for request nodes. Comment their rule, if available. The 'if' should reflect the rule for each node.

Table 6: Instructions for Code Refinement

helper flows (Colang's equivalent of functions) to support algorithm execution, enabling the loading of the STAR API function, and injecting additional code for evaluation purposes.

Helper Variables The algorithm designed in Colang (algorithm 1) determines whether a request node should be executed (i.e., prompt the user for information) by checking the values of its associated slots. To track the state of other node types, we instruct LLM_{CG} to define helper variables following a structured naming pattern, where `<id>` represents the corresponding node's ID:

- `action_<id>`: Stores the return value of external actions.
- `inform_<id>`: Indicates whether the node has been executed and the user has been informed.
- `answered_<id>`: For inform and confirm nodes, stores the user's response.

Example Dialogue Flow and Generated Code Figures 6 and 7 present an example of the STAR task schema pictures, our JSON representation of the corresponding dialogue flow, and the generated Colang code.

A.2 STAR Implementation Details

API Calling While not the primary focus of this paper, we use prompting to generate Colang's Python action code for calling STAR's API and processing its outputs automatically, rather than directly feeding ground-truth API responses as input as done in other works. Every piece of code in our pipeline is automatically generated. Since STAR's API returns randomized outputs, we return the ground-truth API response object when it is available for the exact same turn, instead of the random sampling response.

Dialogue Flows We convert the STAR task schemas, originally provided as images, into the JSON format described in Section 3.1. We use one-shot prompting with GPT-4o to convert pictures into JSON. We convert yellow nodes in pictures into conditions for edges. However, we observed that GPT-4o occasionally misassigns edge connections, requiring manual corrections. Additionally, we enrich the JSON representations by adding more context, such as example values for each slot. We also define `hello` action as the only global action for all tasks.

To better align the dialogue flows with the actual collected dialogues, we introduce minor modifications, such as adding the `inform_nothing_found` action for search tasks. We also identified small inconsistencies between the provided API schema and its implementation. To address this, we refine the API definitions and modify the sampling logic to prevent errors when no results match the given constraints. We will release these improvements, aiming to support future research.

Wizard State Approximation For evaluation, since we are working with offline conversations (i.e., the user is not interacting with the actual TOD system), we approximate the wizard's state at the end of each turn and adjust the program state accordingly. This helps prevent the program's state from deviating from the ground-truth conversation. To achieve this, we first find the node in dialogue flow that the ground-truth conversation was in by mapping the ground-truth action label, if available, to a node in the dialogue flow. We manually create this mapping from action labels to the dialogue flow nodes. Next, we use depth-first search to trace the path from the start of the dialogue flow to the current conversation node. Finally, we adjust each state variable based on whether the corresponding node is part of the current conversation pathway, as described in Algorithm 2.

Prompt Context During evaluation, we incorporate the textual guidelines provided to wizards into LLM_{A}'s context. This additional context helps the

Algorithm 2 Wizard state approximation

Require: Variable v, Graph G, Ground-truth action a_{gt}, Mapping ϕ
Ensure: Approximated value or NULL

$n_{tgt} \leftarrow \phi(a_{gt})$
$n_v \leftarrow v.node$
$P \leftarrow \text{DFSPath}(G, G.start, n_{tgt})$
if $n_v \notin P$ **then**
  return NULL
end if
for each $e \in P$ **do**
  if $e.target = n_v$ **then**
    $e_v \leftarrow e$
    break
  end if
end for
return $\text{ApproxValue}(e_v.condition, v)$

LLM infer some details, such as the time or location of the conversation. For example, a guideline might look like: *Some facts you should be aware of: Right now, it is Tuesday, 12 PM.*

A.3 MultiWOZ Implementation Details

Manually Crafted Dialogue Flows Unlike STAR, MultiWOZ does not provide explicit dialogue flows for each domain, nor do its conversations adhere to a specific flow. To address this, we manually construct simple dialogue flows by analyzing a few example dialogues from each domain. We will release these crafted MultiWOZ dialogue flows. Additionally, for evaluation, we modify the prompts and instruct the LLM to generate delexicalized texts.[9]

Naive Multi-domain Rather than adding a separate domain detection step, we use the gold labels for the active domains at each conversation turn and directly apply the corresponding dialogue flows. We preprocess MultiWOZ 2.2 using the code from Li et al. (2024) to annotate each turn with its active domains. Since evaluation is offline, we separate turns in a conversation by domain, simulate the conversation with prior history, and use the corresponding Colang program(s). Finally, we merge all turns and treat slots from all domains as a single set, accumulating DST predictions during evaluation.

[9]Refer to Nekvinda and Dušek (2021) for more details.

Algorithm 3 Our prompt optimization algorithm. We choose a tuning set of $n = 20$ samples for every DST slot, and write new candidate prompts based on $k = 5$ randomly sampled examples.

Require: DST slots $V^{(S)}$, Validation set $\{h_{2i}^{(u-b)}, y_i, \}_{i=1}^{n}$ $\forall\ v \in V^{(S)}$, Initial prompts $p_v^{(s)}$ $\forall\ v \in V^{(S)}$

for each v in $V_d^{(S)}$ **do**
  $\mathbf{y}_0 \leftarrow \text{DST}(h_{2i-1}^{(u-b)},\ \text{LLM}_\text{A}, p_v^{(s)})$
  $s_0 = \sum \mathbf{y}_0$
  $i = 0$
  while $i < 50$ **do**
    $t \leftarrow$ `Sample`$(X'_d,\ k)$ ▷ Get k examples
    $p_v^{\prime(s)} \leftarrow \text{Rewrite}(t, p_{v,0}^{(s)})$
    $\mathbf{y}_{p'} \leftarrow \text{DST}(h_{2i-1}^{(u-b)}, \text{LLM}_\text{A}, p_v^{\prime(s)})$
    if $\sum \mathbf{y}_{p'} > s_0$ **then**
      `break`
    end if
  end while
end for

Automatic DST Optimization The NAP component's performance is largely dependent on DST, as the next action is determined by the values known to the dialogue system (Equation 3). However, we found in preliminary experiments that the DST performance can be poor with original $P^{(s)}$ prompts, generated by general guidelines outlined in prompt_CG. To this end, we further refine $P^{(s)}$ with automatic prompt optimization.

Our optimization algorithm is summarized in Algorithm 3. From a validation set X, we randomly sample a subset X_d, $\forall\ d \in D$, with $n = 20$ samples per slot, that is set aside to evaluate performance. The remaining examples $X'_d = X \setminus X_d$ are used as few-shot examples for the instruction rewriting. First, we obtain the validation set performance for the initial prompt $p_v^{(s)}$. We rewrite the DST instruction $p_v^{(s)}$ given k examples randomly sampled (without replacement) from X'_d, to obtain a candidate prompt p'_v. Then, we evaluate the performance of p'_v and compare it to the original. If the performance is higher, we replace that prompt in the dialogue's generated Colang code with the new candidate. Otherwise, we repeat this sampling and rewriting process 50 times, or when there are no remaining examples in X'_d.

A.4 Experimental Details

If a generated program contains syntax or runtime errors, we regenerate the code to obtain a functional version. The only exception is Gemini 2.0 Flash, which struggles with calling our defined Colang helper flows. Since this issue is minor, we manually correct the syntax to assess the model's ability to generate programmatic logic for dialogue flows. We access OpenAI models through OpenAI and other models through OpenRouter[10] API.

A.5 NeMo-Guardrails Modifications

We modify NeMo's default `value_from_instruction` prompt structure to begin with a system message, followed by the entire conversation history and instructions combined into a single user message (Figure 4). During our initial experiments, we suspected that NeMo's original prompt structure—where each message in the conversation history was passed as a separate user or assistant message—hindered LLM_A's ability to follow instructions effectively.

Additionally, we refine the post-processing of this action. We found that LLM_A was inconsistent in formatting return values, sometimes enclosing strings in quotation marks while omitting them for non-string types. To address this, we first check if both leading and trailing quotation marks are present and remove them if so. We then attempt to evaluate the return value as a Python literal. If this evaluation fails, we then enclose the value in quotation marks to ensure proper parsing as a string.

Moreover, we fixed an issue related to if-else statements in the Colang parser, which was later merged into the official NeMo repository[11].

A.6 Detailed Baselines

- *AnyTOD* (Zhao et al., 2023) pretrains and fine-tunes T5-XXL for dialogue state tracking and response generation. It uses a Python program to enforce the complicated logic defined by a dialogue flow to guide the LM decisions.

- *IG-TOD* (Hudeček and Dusek, 2023) is a prompting-based approach using ChatGPT to track dialogue states via slot descriptions, retrieve database entries, and generate responses without fine-tuning.

[10] https://openrouter.ai/
[11] GitHub pull request at [PLACEHOLDER].

System Prompt

Below is a conversation between a helpful AI assistant and a user. The bot is designed to generate human-like text based on the input that it receives. The bot is talkative and provides lots of specific details. If the bot does not know the answer to a question, it truthfully says it does not know.

Your task is to generate value for the specified variable. The generated value should be a valid Python literal that is parsable by ast.literal_eval. Always put strings in quotes.

Do not provide any explanations, just output value.

User Prompt

This is some information that is given to the bot to answer to user:
Authenticate the user and tell them their bank balance

This is the current conversation between the user and the bot:
== User:
user action: Hi I would like to check my balance.
== Bot:
bot action: Could I get your full name, please?
== User:
user action: Katarina Miller
== Bot:
bot action: Can you tell me your account number, please?
== User:
user action: I can't remember it right now.

Follow the following instruction to generate a value that is assigned to: $val
Instruction: `this variable stores user's account number. examples of the variable value are "12345678", "87654321". the current variable value is None. given the last user and bot interaction in the current conversation, if the last user message has provided a new value for this variable, output it. if the last interaction is not about this variable, output the current value.`

Figure 4: Example of the modified NeMo `value_from_instruction` action prompt, which is used for DST. $h_{2i-1}^{(u-b)}$ and $p_j^{(s)}$ are provided in each prompt to generate a value for that slot.

- *SGP-TOD* (Zhang et al., 2023) is a purely generative approach that uses two-stage prompting to track dialogue state and generate response. It employs graph-based dialogue flows to steer LLM actions, ensuring adherence to predefined task policies without requiring fine-tuning or training data.

- *BERT + Schema* and *Schema Attention Model (SAM)* (Mosig et al., 2020; Mehri and Eskenazi, 2021) incorporate task schemas by conditioning on the predefined schema graphs, enabling structured decision-making in TODs. SAM extends BERT + Schema approach with an improved schema representation and stronger attention mechanism, aligning dialogue history to the schema for more effective next-action prediction. Both models rely on fine-tuning to learn

Dialogue History	USER: Help there have been suspicious transfers over the past week. my account number is 351531510 and my PIN is 1596.
Wizard Action	ask_name
SGP-TOD Action	bank_ask_fraud_report
CoDial Action	ask_name

(a) *Bank Fraud Report* example dialogue. SGP-TOD fails to collect all necessary authentication details before requesting fraud report information, as its schema defines the next action after `user_bank_inform_pin` as `bank_ask_fraud_details`. In contrast, CoDial verifies that all required information is provided at each request node before proceeding, correctly identifying that the user's name is missing.

Dialogue History	USER: Hi, I am Ben. I would like to plan a party. WIZARD: On what day would you like your party organised? USER: Saturday at 10pm. WIZARD: At what venue would you like to have your party organised? USER: The North Heights Venue if it's available.
Wizard Action	party_ask_number_of_guests
SGP-TOD Action	party_venue_not_available
CoDial Action	party_ask_number_of_guests

(b) *Party Plan* example dialogue. SGP-TOD produces an incorrect and uninterpretable prediction. In contrast, CoDial follows a programmatic logic aligned with the dialogue flow, ensuring interpretability.

Figure 5: Cherry-picked comparison of CoDial and SGP-TOD performance. We use GPT-4o-mini to reproduce SGP-TOD results.

schema-based task policies and improve generalization across tasks.

- *SOLOIST* (Peng et al., 2021) is a Transformer-based model that unifies different dialogue modules into a single neural framework, leveraging transfer learning and machine teaching for TOD systems. It grounds response generation in user goals and database/knowledge, enabling effective adaptation to new tasks through fine-tuning with minimal task-specific data.

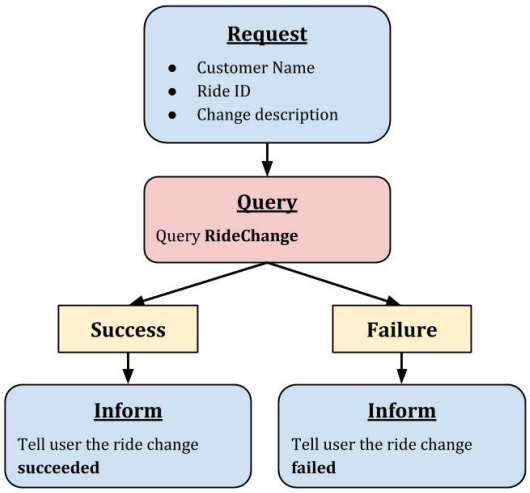
- *MARS* (Sun et al., 2023) is an end-to-end TOD system that models the relationship between dialogue context and belief/action state representations using contrastive learning. By employing pair-aware and group-aware contrastive strategies, Mars strengthens the modelling of relationships between dialogue context and semantic state representations during end-to-end dialogue training, improving dialogue state tracking and response generation.

```
import core
import llm

flow main
  activate automating intent detection
  activate generating user intent for unhandled user utterance
  $action_2 = None
  $inform_3 = False
  $inform_4 = False

  global $generated_output
  while True
    $generated_output = None
    when user said hello
      bot say "Hello, how can I help?"
      continue
    or when unhandled user intent as $state
      $transcript = $state.event.final_transcript

    $customer_name = dst ["action_2", "inform_3", "inform_4"] $customer_name "this variable
stores the name of the customer requesting the ride change. examples of the variable value are
\"John\", \"Jane\". the current variable value is {$customer_name}. given the last user and bot
interaction in the current conversation, if the last user message has provided a new value for
this variable, output it. if the last interaction is not about this variable, output the current
value."
    $ride_id = dst ["action_2", "inform_3", "inform_4"] $ride_id "this variable stores the unique
identifier for the ride (ride id). examples of the variable value are \"102\", \"500\". the
current variable value is {$ride_id}. given the last user and bot interaction in the current
conversation, if the last user message has provided a new value for this variable, output it. if
the last interaction is not about this variable, output the current value."
    $change_description = dst ["action_2", "inform_3", "inform_4"] $change_description "this
variable stores the description of the requested change to the ride. examples of the variable
value are \"Change pickup time\", \"Change destination\", \"Update contact details\". the current
variable value is {$change_description}. given the last user and bot interaction in the current
conversation, if the last user message has provided a new value for this variable, output it. if
the last interaction is not about this variable, output the current value."

    if $customer_name == None or $ride_id == None or $change_description == None
      bot ask info {"$customer_name": $customer_name, "$ride_id": $ride_id,
"$change_description": $change_description}
    else
      if $action_2 == None
        $action_2 = await RideChangeAction(customer_name=$customer_name, ride_id=$ride_id,
change_description=$change_description)

      if $action_2["status"] == "Success"
        if not $inform_3
          bot inform "Alright, thats all changes done for you!"
          $inform_3 = True
      elif $action_2["status"] == "Failure"
        if not $inform_4
          bot inform "Unfortunately I wasn't able to update your booking, sorry."
          $inform_4 = True

    if $generated_output == None
      $generated_output = await LLMGenerateOutputAction()
      $generated_output = str($generated_output)
      await UtteranceBotAction(script=$generated_output)
```

Figure 6: Example of a generated code for STAR *Ride Change* task

```
{
  "nodes": [
    {
      "id": 1,
      "type": "request",
      "slots": {
        "customer_name": {
          "description": "Name of the customer
requesting the ride change",
          "type": "categorical",
          "examples": [ "John", "Jane" ]
        },
        "ride_id": {
          "description": "Unique identifier
(ride ID) for the ride",
          "type": "integer",
          "examples": [ "102", "500" ]
        },
        "change_description": {
          "description": "Description of the
requested change to the ride",
          "type": "string",
          "examples": [
            "Change pickup time",
            "Change destination",
            "Update contact details"
          ]
        }
      }
    },
    {
      "id": 2,
      "type": "external_action",
      "action": "query",
      "query": "RideChange"
    },
    {
      "id": 3,
      "type": "inform",
      "message": "Alright, thats all changes
done for you!"
    },
    {
      "id": 4,
      "type": "inform",
      "message": "Unfortunately I wasn't able
to update your booking, sorry."
    }
  ],
  "edges": [
    { "source": 1, "target": 2 },
    { "source": 2, "target": 3, "condition":
"Success" },
    { "source": 2, "target": 4, "condition":
"Failure" }
  ]
}
```


(a) Converted JSON representation for STAR *Ride Change* task



(b) STAR *Ride Change* task schema

Figure 7: Example of STAR task schema and converted JSON object