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

Anonymous ACL submission

Abstract

Building Task-Oriented Dialogue (TOD) systems that generalize across different tasks remains a challenging problem. Data-driven approaches often struggle to transfer effectively to unseen tasks. While recent schema-based TOD frameworks improve generalization by decoupling task logic from language understanding, their reliance on neural or generative models often obscures how task schemas influence behaviour and hence impair interpretability. In this work, we introduce a novel framework, **CoDial** (Code for Dialogue), which converts a TOD task schema, represented as a novel structured heterogeneous graph, to programmatic LLM guardrailing code, such as NVIDIA's Colang, enabling interpretable and efficient alignment of dialogue policies during inference. We introduce two paradigms, $\text{CoDial}_{\text{free}}$ and $\text{CoDial}_{\text{structured}}$, for generating LLM guardrails, and propose a feedback mechanism that integrates human feedback to iteratively improve the generated code. Empirically, CoDial achieves state-of-the-art (SOTA) performance on the widely used STAR dataset and is on par with SOTA on the MultiWOZ dataset, while also providing interpretability. We additionally demonstrate CoDial's iterative improvement via manual and LLM-aided feedback, making it a practical tool for expert-guided alignment of LLMs in high-stakes domains.[1]

1 Introduction

Task-Oriented Dialogue (TOD) systems play a crucial role in a wide range of applications, enabling users to accomplish complex tasks such as flight booking or apartment searching through natural language conversation (Qin et al., 2023). Building TOD systems that are capable of operating across different tasks remains a challenging area of exploration (Jacqmin et al., 2022). Data-driven approaches aim to train models on large corpora of conversations spanning multiple tasks, allowing them to capture task-related conversational patterns. Nonetheless, such models often struggle with **generalization**: the ability to transfer effectively to new unseen task(s) (Mehri and Eskenazi, 2021).

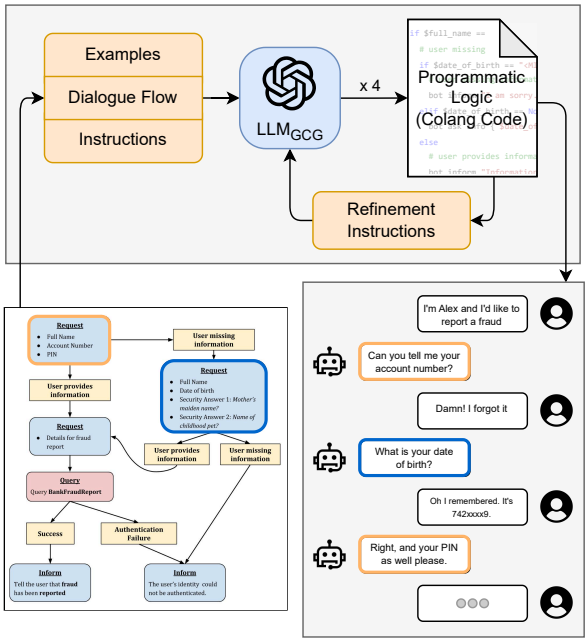


Figure 1: Overview of the proposed CoDial framework. An expert-curated dialogue flow (left) is transformed into executable programmatic logic using an LLM (top). The generated code is iteratively refined before producing the final program, which powers a conversational application (right), enabling the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

Many recent TOD works adopt a schema-based approach, decoupling language understanding from task-specific dialogue policy (Zhang et al., 2023; Zhao et al., 2023; Mehri and Eskenazi, 2021). These systems provide an approach to utilize a parsable *task schema*, often represented as a graph, to encode and enforce complex task logic. Most schema-based methods rely on neural or fully generative-based parsing, which fall short on a key

[1] Our code and data will be made publicly available at [GitHub Placeholder].

property: **interpretability**, or the capacity to examine how the schema is utilized by the model to produce specific outputs. In contrast to opaque neural or generative representations, a programmatic formulation allows one to inspect and reason about the decision process, thereby facilitating *modification* and *improvement* of the system. Interpretability is also especially crucial in high-stakes domains such as law and medicine, where domain experts with minimal technical knowledge need to specify, validate, and refine AI behaviour (Dahan et al., 2023; Tian et al., 2024). AnyTOD (Zhao et al., 2023) addresses this by treating the task schema as a program to be executed by a language model, thereby offering interpretability. However, this approach requires human knowledge to be defined directly as programming functions, which typically demands greater effort and technical expertise than graph-based representations. This added complexity makes the approach less intuitive and more difficult to adopt, particularly for non-technical users.

To enable zero-shot adaptation of interpretable TOD systems to unseen tasks without requiring direct programming, we propose a novel framework, **CoDial** (Code for Dialogue). At the core of CoDial, we leverage programmatic Large Language Model (LLM) guardrailing languages, such as Colang (NVIDIA, 2024). We reframe LLM guardrails as the foundation for defining TOD system behaviour. Therefore, CoDial inherits the advantages of programmatic guardrailing, making it interpretable by design and enabling flexible behaviour definition at inference. In this paper, we use Colang (NVIDIA, 2024), a guardrailing language, to build an interpretable no-code TOD system. Specifically, we convert an input task schema, referred to as a *dialogue flow*, into Colang code. We introduce two paradigms for generating programmatic guardrails: $\text{CoDial}_{\text{free}}$ and $\text{CoDial}_{\text{structured}}$. The contributions of this work are as follows:

- We propose a novel approach for effective alignment of dialogue systems to unseen task schemas that is interpretable by design. To our knowledge, we are the first to treat TOD systems as programmatic LLM guardrailing, such as Colang code, and automate its generation. This allows users to align dialogue systems at inference while ensuring interpretability, without requiring manual coding, thereby lowering the barrier for non-technical users and domain experts.
- The proposed framework, CoDial, consists of three novel components. The heterogeneous dialogue flow representation provides a structure for users to define rich task schemas. The guardrail-grounded code generation pipeline transforms dialogue flows into executable LLM guardrailing programs, allowing for interpretable and flexible control of LLMs in the inference stage. The CoDial human-feedback mechanism incorporates human and LLM feedback to refine and optimize the generated guardrailed conversational models.
- We demonstrate the effectiveness of our interpretable framework on publicly available TOD benchmarks, STAR and MultiWOZ. Our proposed model achieves new state-of-the-art (SOTA) results on STAR and on par results with SOTA on MultiWOZ in a strict zero-shot setting. We also experiment with different code refinement strategies by incorporating user feedback through manual and LLM-aided modification.

2 Related Work

Task-Oriented Dialogue While LLMs have demonstrated impressive capability in a wide variety of domains, they struggled with TOD and fell behind if not used properly (Hudeček and Dusek, 2023). Some research (Zhang et al., 2023; Mehri and Eskenazi, 2021) used a neural schema-guided approach to generalize TOD systems to unseen tasks without interpretability. AnyTOD (Zhao et al., 2023) provided an interpretable neuro-symbolic approach by viewing task schema as a manually-written policy program that controls the dialogue flow. However, beyond the manual coding requirement, AnyTOD also relied on extensive training with highly similar task schemas. As a result, it suffered substantial performance drops when transferred to even slightly different task structures, revealing limited generalization to unseen tasks.

Guardrails CoDial leverages guardrails to steer LLMs behaviour. Guardrailing aims to enforce human-imposed constraints to control LLMs in the inference time (Dong et al., 2024; Rebedea et al., 2023; Guardrails AI). While originating from AI safety, we argue that they can generally be used to define the desired behaviour of LLMs. NVIDIA NeMo-Guardrails (Rebedea et al., 2023) is a toolkit that adds programmable guardrails to LLM-based conversational applications and employs Colang (NVIDIA, 2024), a programming language, to establish highly flexible conversational flows.

Code Generation and Prompt Optimization Code generation has made remarkable progress

with the introduction of LLMs (Le et al., 2022). Although there are still challenges, such as logical consistency and hallucinations (Liu et al., 2024). LLMs are proficient when in-context examples, documentations, or plans are provided (Jiang et al., 2024). There has been research to improve output by rewriting the input prompt, referred to as prompt optimization (Yuksekgonul et al., 2024). Refer to Appendix A.1 for detailed related work.

3 Methodology

We introduce CoDial, a novel framework for constructing interpretable TOD systems without requiring training data or manual programming, as illustrated in Figure 1. A graph-based dialogue flow, defining the behaviour of the TOD system, is the *only* input of CoDial. The core of our approach is leveraging programmatic LLM guardrailing, which allows interpretable and flexible control over the behaviour of an LLM in the inference stage.

CoDial is composed of three key components: (1) CoDial Heterogeneous Dialogue Flows (**CHIEF**) that allows users to define the task schema (Section 3.1), (2) Guardrail-Grounded Code Generation (**GCG**) that automatically creates a TOD system driven by an executable guardrailing program based on the input dialogue flow (Section 3.2.2), and (3) CoDial Human Feedback (**CHF**) that incorporates human/LLM feedback to optimize the generated guardrailing application (Section 3.3). In this paper, we investigate two code generation paradigms for GCG and use the Colang (NVIDIA, 2024) guardrailing language, but any other programmatic paradigm can be applied.

3.1 CoDial Dialogue Flow Representation

We design a structured framework to define rich task schemas, referred to as "*dialogue flows*", as heterogeneous directed graphs, called CoDial Heterogeneous dIaloguE Flows (**CHIEF**) representation. Unlike prior work (Mehri and Eskenazi, 2021; Zhang et al., 2023) that define task schema as an unstructured homogeneous graph—where the single node type represents user intent, an API return value, or a dialogue state—CHIEF allows for different node or edge types in a heterogeneous manner, supporting structured and richer task definition. To the best of our knowledge, we are the first to frame TOD task schema as a heterogeneous directed graph and structure its definition. Specifically, CHIEF provides different node types that can define rich metadata and natural language logic to cover a wide range of tasks and domains, inspired by Mosig et al. (2020). We will show that CHIEF representation can be automatically converted to code for programmatic guardrailing tools. Below, we briefly discuss the main node types and actions in CHIEF. Refer to Appendix A.2 for more details.

Request The request nodes define variables, hereby referred to as 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 will request information specified by the slots.

External Action This node specifies a call to an external function within a dialogue flow. External actions enable the designer to execute complex logics.

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]*), and an optional follow-up question (e.g. *Do you confirm the booking?*).

Global and Fallback Actions CHIEF supports global and fallback actions that are not tied to particular dialogue steps. Global actions can be triggered at any point in the dialogue flow (e.g. responding to a greeting). We also define fallback actions, general responses used when no other action is selected (e.g. *Sorry, I can't help with that*).

The defined nodes logically connect with **edges**. We add a textual `condition` property to edges to allow conditional branching in dialogue flows. We encode the graphs defined by CHIEF as text in JSON format (Figure 10). The JSON-encoded CHIEF representation is translated into programmatic guardrails with GCG, described below.

3.2 Guardrail-Grounded Code Generation

Guardrailing is a general paradigm to define the flow of conversational systems and enable inference-stage control over LLMs' behaviour (Dong et al., 2024; Rebedea et al., 2023). Unlike neural models, programming codes are inherently interpretable. Therefore, programmatic guardrailing allows interpretable and flexible behaviour definition in conversational systems. Our work is the first to formulate TOD system as programmatic guardrailing and automate its generation, removing the need and technical barrier of programming, while ensuring interpretability.

We propose CoDial Guardrail-Grounded Code Generation (**GCG**) that translates

CHIEF-represented dialogue flows into guardrailing code (e.g. Colang[2]). GCG is performed through prompting a code generation model, LLM_{GCG}, with detailed specifications $\text{prompt}_{\text{GCG}}$[3]. Formally, the GCG process is denoted as $g = \text{LLM}_{\text{GCG}}\left(\text{prompt}_{\text{GCG}}\left(x\right)\right)$, where $\text{prompt}_{\text{GCG}}\left(x\right)$ is a JSON-encoded CHIEF graph x wrapped with the prompt template instructions, and g is the program that guardrails the dialogue LLM agent, LLM_{A}.

We investigate two different paradigms for implementing $\text{prompt}_{\text{GCG}}$ in GCG. In the first paradigm, denoted as $\text{CoDial}_{\text{free}}$, $\text{prompt}_{\text{GCG}}$ provides LLM with the syntax and semantic rules of the guardrailing language. Because several code implementations may be valid for a given problem, this paradigm leaves LLM_{GCG} free to design a guardrailing logic that models the given dialogue flow. The second paradigm directly instructs LLM with a certain dialogue flow modelling approach, specifying the structure of g and how to manage the dialogue, treat each node represented in CHIEF, and implement its equivalent guardrailing code. We denote the latter approach as $\text{CoDial}_{\text{structured}}$. Please refer to Appendix A.2 for more details on code generation.

3.2.1 $\text{CoDial}_{\text{free}}$

Since most LLMs are unfamiliar with guardrailing languages, we include the documentation of our chosen language, Colang, in $\text{prompt}_{\text{GCG}}$. As a preliminary design and due to the large context of the documentation, we hand-pick the most essential chunks to provide LLM_{GCG} with a general understanding of Colang's syntax and semantics.

Figure 3a illustrates an overview of the $\text{prompt}_{\text{GCG}}$ for $\text{CoDial}_{\text{free}}$. The prompt begins with Colang syntax and semantic rules, followed by the input dialogue flow x, and concludes with a task description instructing the model to generate Colang code for the flow. The generated code g is an executable guardrailing program that specifies a TOD system aligned to the given CHIEF representation. We also instruct LLM_{GCG} to enable Colang's `continuation on unhandled user intent` flow to allow LLM_{A} to generate output, given fallback actions and all actions defined in the dialogue flow, if the guardrails do not match with the user input in a conversation turn. Figure 8 shows an example of a generated code in $\text{CoDial}_{\text{free}}$.

Algorithm 1 An outline of $\text{CoDial}_{\text{structured}}$.

for each v in $V^{(H)}$ **do**
  $v \leftarrow$ NULL or FALSE ▷ Initialize helpers
end for
while True **do**
  $h_{2i-1} \leftarrow (h_{2i-2}; U_i)$ ▷ User input

  `intent` $\leftarrow$ DETECTINTENT(h_{2i-1}) ▷ Global action
  if `intent` $\neq$ NULL **then**
    $B_i \leftarrow$ INTENTRESPONSE(`intent`)
    continue
  end if

  for each $v_j^{(s)}$ in $V^{(S)}$ **do** ▷ DST
    $v_{old} \leftarrow v_j^{(s)}$
    $v_j^{(s)} = \text{DST}\left(h_{2i-1}, \text{LLM}_{\text{A}}, p_j^{(s)}\right)$
    if $v_j^{(s)} \neq v_{old}$ **then**
      for each hv in $v_j^{(s)}$'s dependents **do**
        $hv \leftarrow$ NULL or FALSE
      end for
    end if
  end for

  `state` $\leftarrow (V^{(S)}, V^{(H)})$ ▷ NAP
  $B_i, V^{(H)} \leftarrow$ NAP(`state`, LLM_{A})
  if B_i = NULL **then** ▷ Fallback action
    $B_i \leftarrow \text{LLM}_{\text{A}}(V^{(H)})$
  end if
end while

3.2.2 $\text{CoDial}_{\text{structured}}$

The simple design of $\text{CoDial}_{\text{free}}$ serves as an interpretable baseline where LLMs generate TOD programs from CHIEF representations and language documentation without guidance. Additionally, we propose $\text{CoDial}_{\text{structured}}$, where we explicitly instruct the model on how to structure the code, model the dialogue states, and interpret each CHIEF node type for GCG. Figure 3b shows an overview of $\text{prompt}_{\text{GCG}}$ for $\text{CoDial}_{\text{structured}}$.

Our $\text{prompt}_{\text{GCG}}$ outlines the output guardrailing code g, as presented in Algorithm 1. We define notations later in this section. The conversation runs within an infinite `while` loop, where the TOD system (1) waits for user input, (2) detects the user's intent for global actions, (3) predicts the slot variables (Dialogue State Tracking; DST); (4) selects an action and generates a response (Next Action Prediction; NAP). In this work, we leverage Colang's built-in intent detection feature for global actions. Note that DST and NAP are combined in a single executable program (i.e., g). Finally, if the NAP component does not generate a response

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

[3] We also experimented with (1) retrieval-augmented generation using the Colang Language Reference documentation and (2) fine-tuning GPT-4o-mini on generation pairs of (programming task, Colang code), but found that prompting with examples works best.

to the given user utterance (e.g., the conversation strays from the defined logic), LLM_A is directly prompted to choose from all available actions, including fallbacks, based on the conversation history. Figure 4 shows the full execution life cycle.

We denote a conversation between user U and chatbot B as a history of messages, Equation 1, where U_i and B_i show user's and chatbot's i-th utterance, respectively. Therefore, the total number of utterances in a conversation history h_{2i} is $2i$.

$$h_{2i} = (U_1, B_1, \ldots, U_i, B_i) \tag{1}$$

We define a set of slot variables $V^{(S)}$ that track values for all of the slots defined in request nodes, and helper variables $V^{(H)}$ that track the state for other (non-request) types of nodes. The union of $V^{(S)}$ and $V^{(H)}$ forms the state of conversation $s = \left(V^{(S)}; V^{(H)}\right)$ at each turn, which is used to determine the next action.

Dialogue State Tracking (DST) As suggested by Feng et al. (2023), LLM prompting shows promising performance in DST, so we take a similar prompting approach in this work. For each slot, LLM_GCG creates explicit instructions to extract the value from the entire conversation history. We leverage Colang's Natural Language Description (NLD) feature to execute these instructions with LLM_A and save the value to a slot variable. Formally, a slot variable $v_j^{(s)} \in V^{(S)}$ is predicted as Equation 2, where $p_j^{(s)} \in P^{(s)}$ is the prompt generated by LLM_GCG to extract the value for $v_j^{(s)}$.

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

Next Action Prediction (NAP) We instruct LLM_GCG to convert the CHIEF graph x into a conditional logic consisting of nested if/else statements, to generate a response given s_i, the state of the conversation at turn i. Each generated if statement corresponds to a node n_j in x, and aligns s_i to the conversation logic outlined by the CHIEF representation. If an if statement holds, this indicates s_i is "at" that node and the corresponding action is executed; otherwise, for each outgoing edge at node n_j, the system checks for traversal. If there is a natural language condition associated with the edge and the condition is met, or if there is no explicit condition, LLM_A traverses the graph to the associated target node. Formally, next bot utterance is defined in Equation 3.

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

3.3 CoDial Human Feedback Integration

CoDial's Human Feedback (CHF) mechanism incorporates human feedback to refine or improve the generated guardrailing code g. The code enhancement through feedback comprises two broad approaches: i) manual and ii) LLM-aided modifications.

CHF assists iterative improvement of g in the form of **refinement instructions (RIs)**, shown at the top in Figure 1. RIs allow the user of CoDial to refine the generated logic through hand-crafted instructions. We provide three instructions for refining certain aspects of the output code: correct logic (i.e., if statement) for each node, DST initialization, and request node checks. Since these RIs, presented in Table 6, are a set of prompts, they can be modified and extended dynamically over time. In addition, CHF allows for manual modifications on the dialogue flow (Appendix A.3) and manual DST prompt optimization (Appendix A.4). We also attempt to refine the generated DST prompts ($P^{(s)}$) through automatic prompt optimization. Please refer to Appendix A.4 for details on automatic DST prompt optimization.

4 Experimental Settings

Models We use GPT-4o, GPT-5 (with reasoning levels of minimal and low), Claude 3.5 Sonnet, Gemini 2.0 Flash, and DeepSeek V3 (DSV3) as LLM_GCG and LLM_A. Larger models are used for code generation—given the complexity of the task, we found that smaller models often fail to fully adhere to instructions. For further details, please refer to Appendix A.5.

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. We also use silver state annotations created in STARv2 (Zhao et al., 2023) for ablation studies. Refer to Appendix A.3 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 (Zhang et al., 2023),

Model	Int.	Graph Transfer	STAR			MultiWOZ 2.2				
			F1	Accuracy	BLEU	JGA	Inform	Success	BLEU	Combined
SOLOIST	✗	✗	-	-	-	35.9	81.7	67.1	13.6	88.0
MARS	✗	✗	-	-	-	35.5	88.9	78.0	19.6	103.0
DARD	✗	✗	-	-	-	-	96.6	88.3	12.1	104.6
IG-TOD (few-shot)	✗	✗	-	-	-	27	-	44	6.8	-
			(Strict) Zero-shot Transfer							
IG-TOD (zero-shot)	✗	✗	-	-	-	13	-	31	4.2	-
BERT + Schema	✗	✓	29.7*	32.4*	-	-	-	-	-	-
SAM	✗	✓	51.2*	49.8*	-	-	-	-	-	-
AnyTOD XXL	✓	✗	68.0*	68.0*	44.3*	30.8	76.9	47.6	3.4	65.6
SGP-TOD	✗	✓	53.5	53.2	-	-	**82.0**	**72.5**	**9.2**	**86.5**
CoDial$_{free}$	✓	✓								
CoDial (4o, 4o-mini) − RI			36.6	36.1	23.0	-	-	-	-	-
CoDial$_{structured}$	✓	✓								
CoDial (4o, 4o-mini)			58.5	60.1	45.2	28.4	76.6	54.6	3.5	69.1
CoDial (4o, 5-mini:l)			**59.2**	**60.2**	**46.5**	**37.0**	79.6	70.8	4.3	79.5

Table 1: Comparison of CoDial with baselines on STAR and MultiWOZ benchmarks. In "Strict Zero-Shot" the models have not seen a same task schema architecture in the training data. Results with an asterisk (*) are evaluated in a more relaxed, non-strict setting, and therefore, are not directly comparable. "Int." stands for "Interpretable." SAM results are cited from Zhao et al. (2023).

we report results in a naive oracle domain setting. Please refer to Appendix A.4 for more details.

4.2 Metrics

For the STAR dataset, we compute BLEU-4 score (Papineni et al., 2002) and follow Mosig et al. (2020) to compute F1 and accuracy. 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 of three runs. Refer to Appendix A.5 for more implementation details.

4.3 Baselines

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

- *IG-TOD* (Hudeček and Dusek, 2023), a few-shot prompting-based approach.
- *AnyTOD* (Zhao et al., 2023) pretrains and fine-tunes T5-XXL for DST and response generation. It views task schema as a Python program to enforce the conversation logic.
- *SGP-TOD* (Zhang et al., 2023) is a zero-shot prompting approach that employs graph-based dialogue flows. Refer to Appendix A.6 for details on fair comparison.

For the remainder of this paper, by CoDial we refer to CoDial$_{structured}$ with GPT-4o and GPT-4o-mini as LLM$_{GCG}$ and LLM$_{A}$, respectively, unless otherwise specified.

5 Experimental Results

Superior Performance with Explicit Schemas Table 1 summarizes CoDial results on the benchmark datasets. CoDial achieves strong performance, surpassing all baselines except AnyTOD, and sets the new SOTA in strict zero-shot setting, where no same-architecture task schema is seen by the model. Our framework improves F1 by **+5.7** and accuracy by **+7** points over the previous SOTA. While AnyTOD achieves higher scores, it is evaluated in the easier non-strict setting and requires the user to write code, limiting accessibility to non-programmers. In contrast, CoDial operates in a graph-based transfer manner, eliminating the need for manual programming. We also observe that CoDial$_{free}$ lags behind CoDial$_{structured}$ and most baselines, indicating that LLMs struggle with unsupervised guardrailing code generation, likely due to limited availability of guardrailing languages, and still require human supervision.

Competitive Performance on MultiWOZ Table 1 also shows our results on the MultiWOZ dataset. 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 heuristics-grounded and programmatic approaches like CoDial and AnyTOD. Consequently, CoDial is less effective on MultiWOZ. To address this, we experiment with GPT-5 with built-in reasoning as LLM$_{A}$ to improve the DST performance. We observe that CoDial achieves competitive performance with SOTA on Inform and Success metrics under the strict zero-shot setting, while maintaining interpretability. We further analyze the effect of DST performance in Section 5.2. Similar to AnyTOD, CoDial relies on template-based outputs, which accounts for its lower BLEU score.

Model	LLM_{GCG}	LLM_A	RI	F1	Acc.	BLEU
$CoDial_{free}$						
CoDial − RI	4o	4o-mini	✗	36.6	36.1	23.0
CoDial − RI	Sonnet	4o-mini	✗	32.1	31.8	18.0
$CoDial_{structured}$						
CoDial ORIGINAL DFS	4o	4o-mini	✓	51.9	51.1	38.9
CoDial − RI	4o	4o-mini	✗	56.1	57.3	38.4
CoDial − RI	Sonnet	4o-mini	✗	57.0	58.4	39.2
CoDial	DSV3	4o-mini	✓	46.1	48.0	28.0
CoDial*	Gem. 2 Fl.	4o-mini	✓	50.5	52.1	32.9
CoDial	Sonnet	4o-mini	✓	57.7	58.5	39.3
CoDial	4o	DSV3	✓	55.6	56.8	44.2
CoDial	4o	4o-mini	✓	58.5	60.1	45.2
CoDial	4o	5-mini:m	✓	59.0	**60.4**	45.2
CoDial	4o	5-mini:l	✓	**59.2**	60.2	**46.5**

Table 2: Comparison of CoDial performance across different settings and (LLM_{GCG}, LLM_A) pairs on STAR dataset. The generated code for the model with an asterisk (*) has been manually fixed and is not directly comparable. DF stands for "dialogue flow."

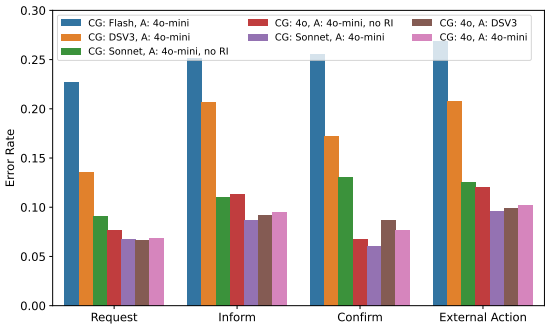


Figure 2: Error rate comparison of agents' predicted state on the STAR dataset across different node types, coloured by (LLM_{GCG}, LLM_A) pairs.

5.1 Detailed Analysis

Impact of Model Selection and CHF We experiment with different model choices for the (LLM_{GCG}, LLM_A) pairing (Table 2). Better instruction following and more robust code generation often translate to higher overall performance. Because most LLMs are unfamiliar with guardrailing languages such as Colang, they must accurately interpret the $prompt_{GCG}$ to 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, 5-MINI) with built-in reasoning achieve the highest performance in all metrics. CoDial (4o, 4o-MINI) performs comparably with lower cost and latency. Therefore, we use GPT-4o-mini for our ablations in Section 5.2. 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 F1 and +1.5 accuracy.

Additionally, without modifications (Appendix A.3), the original STAR dialogue flows result in lower performance (F1: 51.9). Manually modifying the dialogue flow and applying RIs to the generated code significantly enhances performance. We further explore the impact of LLM-aided corrections in Section 5.2.

Action Prediction and API Calls We find that NeMo Guardrails' intent detection performs strongly, achieving an F1 score of 96.3 on global actions (Table 3). 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 3 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 fallbacks, LLM-generated actions only account for 9.2% of predictions, indicating that our NAP logic is generally effective at generating outputs based on the predicted state. Since fallback actions are 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 wizard annotations. Additionally, we observe a significant performance drop from fallback to non-fallback actions in both F1 (51.4 → 38.7) and accuracy. This suggests that despite having an explicit schema, LLMs struggle to capture the more complex logic needed to predict non-fallback actions. Our findings align with Dong et al. (2024), reinforcing the need for a neuro-symbolic approach.

State Prediction Figure 2 shows the error rate of the predicted conversation state across different node types for each model. To approximate the error, we compare the model's predicted state with the estimated ground-truth state (i.e., the wizard's state), as described in Appendix A.3. We find that the error rate generally inversely correlates with the overall performance in Table 2; higher-performing models tend to exhibit lower state prediction error.

Single- vs. Multi-Domain Performance 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

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 3: 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	F1	Acc.	BLEU
CoDial	58.5	60.1	**45.2**
Refinement Instructions (RI)			
– 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
Generative Approach			
– NAP	47.4	47.0	25.8
– NAP & DST	42.7	43.0	23.8
Oracle Vote			
DST: 4o-mini + DSV3	60.2	61.6	46.8
Silver Label DST			
+ 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 (few-shot)	27	-	44	6.8	-
CoDial SINGLE*	46.2	91.5	77.6	3.2	87.7
CoDial	28.4	76.6	54.6	3.5	69.1
Oracle Belief State					
IG-TOD (few-shot)	-	-	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 AUTO	31.1	72.3	57.2	3.6	68.3
CoDial MANUAL	28.5	80.4	57.9	3.6	72.7

Table 5: Ablations on MultiWOZ. Settings with an asterisk (*) are not directly comparable due to a simpler task setup.

dialogues in Table 5, where it performs well. However, with our naive oracle domain setting, CoDial performance drops significantly. This is likely due to compounded errors from DST to NAP, which we analyze further in Section 5.2.

5.2 Ablation Studies

Oracle DST Performance To assess the impact of DST error, we evaluate CoDial under an Oracle setting. Since STAR does not provide gold DST labels, we simulate an oracle setting by using 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 use the gold belief states, which leads to a substantial performance improvement (Table 5). These findings suggest that investigating more advanced DST approaches, such as inference-time scaling explored in Section 5, could be a promising direction to improve performance. We also experiment briefly with prompt optimization for lower cost DST improvements, described below.

Code Optimization We use LLMs to perform iterative code refinement and 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. After prompting the LLM to iteratively refine its outputs, CoDial achieves better accuracy and fluency (compared to CoDial − RI in Table 2). We also conduct an ablation study to examine the effect of the individual RIs, summarized in Table 4. Although all RIs are beneficial, most of the performance improvements can be attributed to the third RI, which refines the conditional logic of request nodes.

After observing the results of the oracle DST setting, we also apply prompt optimization to improve DST accuracy. As shown in Table 5, automatic prompt optimization yields only marginal gains across metrics, with the exception of Inform, suggesting that automatic DST improvement remains a non-trivial challenge. To explore the impact of human feedback, we also experiment with manual prompt optimization (Appendix A.4), making minor edits to the prompts for the "attraction" domain. This results in consistent improvements across all metrics, reinforcing that human-crafted prompts can still outperform automatic optimization.

Generative Approach To better understand the effectiveness of the proposed CoDial architecture, we experiment with a setting in which the NAP component is removed and all actions are predicted in a fully generative manner by the LLM_A, similar to Zhang et al. (2023). We prompt the LLM_A with simplified dialogue flows following their work and include predicted DST slots in the prompt. This results in a substantial drop in performance (Table 4), highlighting the importance of our NAP approach. We further ablate the model by removing the DST component, causing a larger drop as expected.

6 Conclusion

In this work, we introduced CoDial, a novel framework for building interpretable TOD systems by grounding structured dialogue flows to programmatic guardrails. CoDial introduces CHIEF, a heterogeneous graph representation of dialogue flows, and employs LLM-based code generation to automatically convert dialogue flows into executable guardrail specifications (e.g., in NVIDIA's Colang), enabling zero-shot creation of interpretable TOD systems with minimal human effort. Moreover, through manual and LLM-aided refinements, CoDial supports rapid incorporation of user feedback, further enhancing the generated code. Our empirical findings support CoDial's effectiveness, achieving SOTA performance on STAR and competitive results on MultiWOZ in a strict zero-shot setting.

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, improving DST performance and efficiency remains a potential direction for 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

Moreover, developing measurable metrics for interpretability and user accessibility, two central motivations of this work, remains an open direction. We demonstrate interpretability indirectly through modifiability, as shown in the incorporation of human feedback in CHF. In terms of accessibility, an ideal study would evaluate the effort required for users to represent their knowledge as a task schema (e.g., CHIEF representation in CoDial) and compare it across approaches. While CoDial abstracts away manual programming, certain applications may still require some familiarity with LLM guardrails and Colang for effective modification. CoDial mitigates this through textual refinement interfaces (RIs), though their adequacy ultimately depends on the specific use case.

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. 2024. 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.

Guardrails AI. Guardrails: Adding guardrails to large language models. `https://github.com/guardrails-ai/guardrails`. Accessed: 2025-05-16.

Aman Gupta, Anirudh Ravichandran, Narayanan Sadagopan, and Anurag Beniwal. 2024. DARD: A multi-agent approach for task-oriented dialog systems. In *NeurIPS 2024 Workshop on Open-World Agents*.

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.

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.

NVIDIA. 2024. Nvidia nemo guardrails, docs.nvidia.com/nemo/guardrails/colang_2/overview.html. Accessed: 2025-05-19.

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 Detailed Related Work

Task-Oriented Dialogue Building generalizable conversational systems is challenging due to the complexity of human conversations, particularly when domain expertise is involved (Chen et al., 2017), leading to a focus on task-oriented systems for specific domains (Jacqmin et al., 2022). While LLMs have demonstrated impressive capability in a wide variety of domains, they struggled with TOD and fell behind if not used properly (Hudeček and Dusek, 2023). Some research (Zhang et al., 2023; Mehri and Eskenazi, 2021) has used a neural schema-guided approach to generalize TOD systems to unseen tasks without interpretability. AnyTOD (Zhao et al., 2023) provided an interpretable neuro-symbolic approach by viewing task schema as a manually-written policy program that controls the dialogue flow. However, beyond the manual coding requirement, AnyTOD also relied on extensive training with highly similar task schemas. As a result, it suffered substantial performance drops when transferred to even slightly different task structures, revealing limited generalizability to unseen tasks.

Guardrails CoDial employs guardrails to steer LLMs behaviour. Guardrailing aims to enforce human-imposed constraints to control LLMs in the inference time (Dong et al., 2024; Rebedea et al., 2023; Guardrails AI). While originating from AI safety, we argue that they can generally be used to define desired behaviour to constrain. Although traditional dialogue management systems, like Google Dialogflow[4], allow rigid modelling of dialogue states, they often lack the flexibility to define complex task logic, and it is difficult for a user to further enhance the system. NVIDIA NeMo-Guardrails (Rebedea et al., 2023) is a toolkit that adds programmable guardrails to LLM-based conversational applications without fine-tuning. NeMo-Guardrails employs Colang (NVIDIA, 2024), a programming language, to establish highly flexible conversational flows and guide LLMs within them. Dong et al. (2024) suggested using neuro-symbolic approaches to guardrail LLMs, where a neural agent (e.g., an LLM) can deal with frequently seen cases, and a symbolic agent can embed human-like cognition through structured knowledge for the rare cases.

[4] https://dialogflow.cloud.google.com

Code Generation and Prompt Optimization We use code generation strategies to convert structured graphs into programmatic guardrails. Code generation has made remarkable progress with the introduction of LLMs (Le et al., 2022). Although there are still challenges such as logical consistency and hallucinations (Liu et al., 2024). LLMs are proficient when in-context examples, documentations, or plans are provided (Jiang et al., 2024). There are many emerging methods to further optimize LLM generations (e.g., self-reflection, where LLMs are requested to update their own response), which have been shown to reduce hallucinations and improve problem solving (Ji et al., 2023). There has been research to improve output by rewriting the input prompt, referred to as prompt optimization (Yang et al., 2023; Yuksekgonul et al., 2024).

A.2 Details on CHIEF and GCG

A.2.1 CHIEF

Below, we discuss the main node types and actions in CHIEF.

Request The request nodes define 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 will request information specified by the slots. Each slot is assigned a data type (e.g. categorical) and accompanied by a few example values. Additionally, CHIEF includes 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 an external function 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 agent 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.

=== Colang Syntax:

== Flow Definition:

…

== Action Definition:

…

== Conditional Branching Definition:

…

=== Basic Colang Description:

Colang is a language to define LLM dialogical behaviour. …

=== Dialogue Flow

[x: Input CHIEF-Formatted Dialogue Flow as JSON]

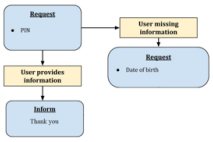
=== Task Description

Convert the given task graph to a Colang code…

(a) $\text{prompt}_{\text{GCG}}$ for $\text{CoDial}_{\text{free}}$

=== DST in Colang Example

[Simple Dialogue Flow]



Equivalent DST Code

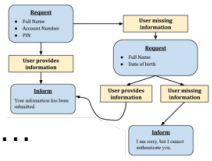
```
$pin = dst [...] $pin "extract
user pin from conversation history ..."
$dob = dst [...] $dob "extract user date of birth
from conversation history ..."
```

=== 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

[x: Input CHIEF-Formatted Dialogue Flow as JSON]

=== 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, ...

(b) $\text{prompt}_{\text{GCG}}$ for $\text{CoDial}_{\text{structured}}$

Figure 3: An overview of $\text{prompt}_{\text{GCG}}(x)$, where a dialogue flow x is wrapped with system prompt template.

Global and Fallback Actions In addition to nodes, CHIEF supports representing global and fallback actions that are not tied to particular dialogue steps. Global actions can be triggered at any point in the dialogue flow (e.g. responding to a greeting). We also define fallback actions, general responses used when no other action is selected (e.g. *Sorry, I can't help with that*).

We represent the graphs defined by CHIEF (Section 3.1) 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 10). The JSON nodes, global and fallback actions, and functional specifications for function calls are translated into Colang code with our automatic code generation pipeline. The external action node functions, referred to as Actions in Colang, are implemented in Python.

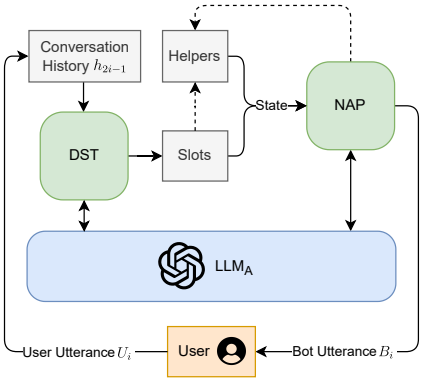


Figure 4: Execution life cycle of the generated agent in $\text{CoDial}_{\text{structured}}$.

A.2.2 GCG

Dialogue State Tracking (DST) Since updating a slot may affect the state (e.g. in a search task,

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

modifying the search criteria requires re-executing the search), CoDial needs to identify the helper variables that need to be invalidated when each slot is updated. We instruct LLM_{GCG} 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, when the slot is updated during execution.

Post-processing Following code generation by the LLM, we apply rule-based post-processing to ensure proper execution. This includes adding 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 $\text{CoDial}_{\text{structured}}$ 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_{GCG} 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 8 to 10 present an example of the STAR task schema pictures, our JSON representation of the corresponding dialogue flow, and the generated Colang code.

A.3 STAR Implementation Details

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 ablation studies.

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 CHIEF representation 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 and `goodbye`, `out_of_scope`, and `anything_else` as fallback actions 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

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 $\text{APPROXVALUE}(e_v.condition, v)$

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 $\text{CoDial}_{\text{structured}}$ 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's 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 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.4 MultiWOZ Implementation Details

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.

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.[5]

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.

DST Prompt 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_GCG. To this end, we further refine $P^{(s)}$ with automatic prompt optimization.

Our optimization algorithm is summarized in

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

Algorithm 3 Our prompt optimization algorithm. We randomly sample a training and validation set of size 20 and 50 for every DST slot, respectively.

Require: Training set $\mathcal{D}_{\text{train}}$, Validation set $\mathcal{D}_{\text{val}}$, Instruction I, Agent LLM_{A}, Optimizer LLM M, Batch size B

$\hat{Y}_{\text{val}} \leftarrow \text{DST}(\mathcal{D}_{\text{val}}.H, \text{LLM}_{\text{A}}, I)$
Initialize $S \leftarrow \textsc{ComputeScore}(\hat{Y}_{\text{val}}, \mathcal{D}_{\text{val}}.Y)$
$I_{\text{best}} \leftarrow I$
Divide $\mathcal{D}_{\text{train}}$ into batches $\mathcal{B}_1, \ldots, \mathcal{B}_n$ of size B
for each batch $\mathcal{B}$ in $\mathcal{D}_{\text{train}}$ **do**
  $(H, Y) \leftarrow \mathcal{B}$
  $\hat{Y} \leftarrow \text{DST}(\text{H}, \text{LLM}_{\text{A}}, I_{\text{best}})$
  $I \leftarrow M.\textsc{Rewrite}(H, \hat{Y}, Y, I)$
  $\hat{Y}_{\text{val}} \leftarrow \text{DST}(\mathcal{D}_{\text{val}}.H, \text{LLM}_{\text{A}}, I)$
  $S \leftarrow \textsc{ComputeScore}(\hat{Y}_{\text{val}}, \mathcal{D}_{\text{val}}.Y)$
  if $S > S_{\text{best}}$ **then**
    $S_{\text{best}} \leftarrow S$
    $I_{\text{best}} \leftarrow I$
  end if
end for
return $I_{\text{best}}, S_{\text{best}}$

Algorithm 3. For each DST variable $v_j^{(s)}$, we randomly sample two mutually exclusive sets of conversation turns to serve as training and validation sets. The training examples are divided into batches of 5, and each batch is used to guide the optimizer GPT-4o model to rewrite the instruction $p_j^{(s)}$, resulting in a candidate prompt. If the revised instruction improves performance on the validation set, it is retained; otherwise, the original is kept, ensuring that modifications are only accepted when they lead to measurable improvements.

In addition, we manually refine the prompts for the worst-performing domain, "attraction." The edits include defining what an "attraction" is by listing all possible types, and propagating the predicted `type` value to other slot instructions to maintain consistency. We leave further investigation of this technique—passing key slot predictions across instructions within a domain—as future work.

A.5 Experimental Details

We use GPT-4o[6], Claude 3.5 Sonnet[7], Gemini 2.0 Flash[8], and DeepSeek V3 (DSV3) (DeepSeek-AI et al., 2024) as LLM_{GCG}, and GPT-4o-mini, GPT-5, and DSV3 as LLM_{A}. We access OpenAI models through OpenAI and other models through OpenRouter[9] API [10].

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.

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). Since $\text{CoDial}_{\text{free}}$ does not include an explicit DST component and most MultiWOZ metrics rely on DST predictions, we do not report $\text{CoDial}_{\text{free}}$ results on this dataset.

NeMo-Guardrails To implement the Colang guardrails, we use a fork from NeMo-Guardrails version 0.11, modified to inject our evaluator class[11]. We use this class to evaluate on the ground truth user-wizard history, instead of the history of user-bot conversation, similar to Nekvinda and Dušek (2021).

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 5). 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

[6] https://openai.com/index/hello-gpt-4o/
[7] https://www.anthropic.com/news/claude-3-5-sonnet
[8] https://developers.googleblog.com/en/gemini-2-family-expands/
[9] https://openrouter.ai/
[10] All models were accessed from January to September 2025.
[11] The modified NeMo-Guardrails version that we used for the experiments is available at [GitHub Placeholder].

Model	F1	Acc.
SGP-TOD GPT3.5-E2E	53.5	53.2
SGP-TOD GPT4O-MINI-E2E	41.3	44.3
SGP-TOD GPT4O-MINI-E2E Adapted	40.3	43.8

Table 7: Comparison of SGP-TOD baselines.

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.

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

Moreover, we fixed an issue related to if-else statements in the Colang parser, which was later merged into the official NeMo repository[12].

[12]GitHub pull request at [PLACEHOLDER].

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

A.6 Detailed Baselines

- *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.

- *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.

- *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

```
if $venue_name == None or $host_name == None or $day == None or $start_time == None or $number_of_guests == None:
      bot ask info {"$venue_name": $venue_name, "$host_name": $host_name, "$day": $day, "$start_time": $start_time,
"$number_of_guests": $number_of_guests}
```

Figure 7: Example of code logic in CoDial that enables interpretability. A user can inspect runtime variables to trace the reasoning behind the outputs generated by the TOD system.

actions, ensuring adherence to predefined task policies without requiring fine-tuning or training data. To ensure a fair comparison, we replicated their setup using the same newer LLM_A model as ours (Table 7). We ran their released code without modification, except for switching the API model to GPT-4o-mini. Surprisingly, performance dropped significantly. After contacting the authors, they advised adapting the prompt structure to the aligned LLMs—placing instructions in the system message and including examples and dialogue history in the user message. However, even with this adaptation, the performance did not match the results originally reported with GPT-3.5, suggesting that a generative approach could not be a trivial solution and requires careful prompt engineering. Figure 6 further illustrates differences between CoDial and SGP-TOD through two cherry-picked examples. Specifically, in Figure 6b, by analyzing the variable values in the runtime, a user can easily spot that the generated output stems from the code snippet in Figure 7, where it asks for the next missing value, if any.

- *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 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.

- *DARD* (Gupta et al., 2024) is a multi-agent TOD system that delegates responses across domain-specific agents coordinated by a central dialogue manager. It combines fine-tuned models (Flan-T5-large, Mistral-7B) with large LLMs (Claude Sonnet 3.0), yielding SOTA results on MultiWOZ with significant gains in inform and success rates. However, its performance depends heavily on extensive carefully designed prompt tuning and few-shot examples, limiting efficiency and increasing human effort.

```
import core
import llm

flow main
  activate automating intent detection
  activate generating user intent for unhandled user utterance
  activate continuation on unhandled user intent

# User intent definitions
flow user greeted
  user said "hello" or user said "hi"

flow user requested ride change
  user said "I want to change my ride" or user said "Can you help me with my ride?"

flow user provided ride id
  user said "My booking ID is {$ride_id}"

flow user provided change description
  user said "Change pickup time" or user said "Change destination" or user said "Update contact details"

[...]

# Bot action flows
flow bot greet
  bot say "Hello, how can I help?"

flow bot ask for ride change
  bot say "Sure, what can I change for you?"

flow bot ask for booking number
  bot say "Can I get your booking ID, please?"

[...]

# Policy flows
@active
flow handling greeting
  await user greeted
  bot greet

@active
flow handling ride change request
  await user requested ride change
  bot ask for name

@active
flow handling name provision
  await user provided name
  bot ask for booking number

@active
flow handling booking number provision
  await user provided ride id
  bot ask for ride change

@active
flow handling change description
  await user provided change description
  $query_result = await QueryAction(name="{$customer_name}", ride_id="{$ride_id}",
change_description="{$change_description}")
  if $query_result == "Success"
    bot inform changes successful
  else
    bot inform changes failed

[...]
```

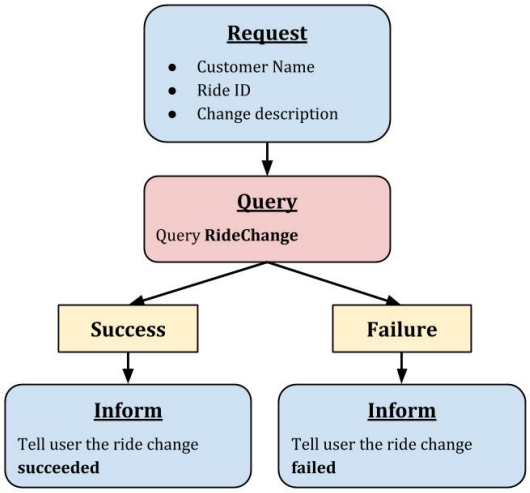
Figure 8: Example of a generated code for STAR *Ride Change* task with $\text{CoDial}_{\text{free}}$

```
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 9: Example of a generated code for STAR *Ride Change* task with $\text{CoDial}_{\text{structured}}$

```
{
  "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 10: Example of STAR task schema and converted JSON object