PyTOD: Programmable Task-Oriented Dialogue with Execution Feedback

Anonymous ACL submission

Abstract

Programmable task-oriented dialogue (TOD) agents enable language models to follow structured dialogue policies, but their effectiveness hinges on accurate state tracking (DST). We present PyTOD, an agent that generates executable code to track dialogue state and uses policy and execution feedback for efficient error correction. To achieve this, PyTOD employs a simple constrained decoding approach, using a language model instead of grammar rules to follow API schemata. This leads to state-of-the-art DST performance on the challenging SGD benchmark. Our experiments show that PyTOD surpasses strong baselines in both accuracy and stability, demonstrating the effectiveness of execution-aware state tracking.

1 Introduction

TOD agents provide natural language interfaces which enable users to control their digital environment to complete daily tasks. Such agents typically include a *dialogue state tracking* (DST) component, which maps the conversation history to a symbolic representation of the task-relevant information communicated during the exchange. At each turn, a *dialogue manager* (DM) uses this information to take *system actions*[1] necessary to help the user complete the task. The agent behaviour is controlled by a *dialogue policy* defined by application developers.

Adapting to new domains has long been a challenge for state tracking (Jacqmin, 2022), dialogue management (Mosig et al., 2020a) and end-to-end (Zhao et al., 2023) agents, as it often requires developers to collect and annotate new datasets for retraining. To address this challenge, fine-tuning pretrained language models (PLMs) (Radford et al., 2019; Raffel et al., 2020) within the schema-guided paradigm (Rastogi et al., 2019; Mosig et al., 2020a) has emerged as a powerful approach. Schemata define the APIs accessible to the agent, including textual descriptions of their functions and parameters (commonly referred to as *slots*). Mosig et al. (2020a) extend schemata by including descriptions of the actions that TOD agents can perform. Zhao et al. (2023) build on these advances, proposing a neuro-symbolic approach to design AnyTOD, a state-of-the-art (SOTA) agent capable of following

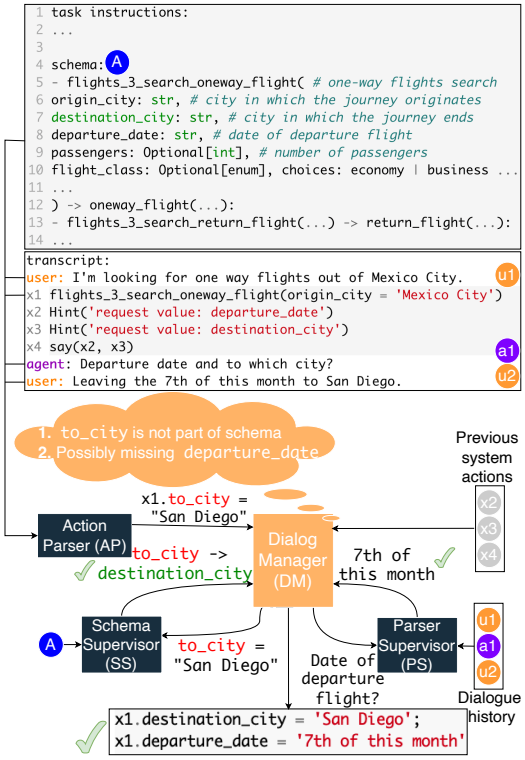

Figure 1: PyTOD overview. The action parser generates `python` instructions (x1.to_city = "San Diego") representing the actions the user took at the current turn (u2) given API schemata (A), dialogue history (u1, a1) and previous user actions (x1). The dialogue manager (DM) executes the user action in a simulated environment. A schema supervisor is invoked by the DM to correct errors if predicted instructions contain slot names that are not part of the API schema (e.g., the slot to_city is mapped to destination_city, a member of the schema). Given knowledge of previous system actions (x2, x3) the DM detects slot omissions and invokes a parser supervisor to correct them (e.g., flight date is recovered).

[1]For example, retrieving information from a knowledge base or prompting the user to provide task constraints.

dialogue policies unseen during fine-tuning[2].

AnyTOD uses the schema and dialogue history to first generate a symbolic *state sequence*. This sequence identifies the API the user wishes to interact with and the slot values the user has mentioned. A second symbolic sequence - encoding the actions the user[3] and system have taken - is generated sequentially after the state sequence. Both sequences are subsequently interpreted by a *deterministic policy program* which recommends the next system action. State sequence is thus critical: prediction errors can prevent the system from taking the correct actions, leading to breakdowns in the interaction.

Despite its strengths, AnyTOD has some limitations. First, it re-estimates the state and action history at every dialogue turn solely based on the dialogue history and schema, which increases generation length and amplifies the risk of state-tracking errors. Second, it fails to exploit previous system actions to verify the correctness of the state sequence. Finally, its reliance on symbolic representations of state and action sequences requires additional system components to translate them into executable code, adding deployment complexity. We present PyTOD, a programmable dialogue system that addresses these challenges by directly communicating with its execution environment and policy programs to perform accurate state tracking for unseen APIs and domains, as discussed next.

PyTOD overview PyTOD incrementally generates dialogue states as code, using policy- and execution feedback for accurate state estimation (Figure 1). It operates as follows: (1) an action parser (AP) (§2.1) processes the user query to produce one or more `python` instructions; (2) the dialogue manager (DM) (§2.2) executes these instructions; (3) any attribute errors raised during execution are resolved by the schema supervisor (SS) (§2.2.2), a language model that constrains code generation according to the decoding schema; and (4) the DM evaluates the constrained output by comparing expected and current environment states, invoking a parser supervisor (PS) (§2.2.3) to recover from possible omissions or semantic errors. The dialogue state is derived by executing the complete program generated by PyTOD at each turn.

Contributions Unlike PyTOD, TOD agents often optimise DST and DM independently in pipeline architectures (Neelakantan et al., 2019), overlooking how policy information can enhance state generation accuracy. We further demonstrate that feedback from the execution environment enables language models to constrain decoding without requiring additional training data, with minimal developer effort and only a slight increase in system latency. PyTOD achieves SOTA performance on the challenging Schema-Guided Dialog (SGD) dataset (Rastogi et al., 2019).

As an additional contribution, we release `pytodlib`, a `python` library that simulates SGD APIs, including database responses and API behavior. This toolkit addresses the scarcity of conversational tool-use corpora, providing a valuable resource for benchmarking large language models' ability to handle complex, multi-task, goal-oriented conversations while following predefined policies.

2 PyTOD

2.1 Action parser

The AP parses user utterances into `python` expressions, which are executed to carry out the user's actions. The prompt consists of: (1) a header containing task-specific instructions, a linearised schema, a list of completed tasks, and entities (e.g., flights) returned by them (§2.1.1); (2) a session transcript, where user and system turns are interleaved with `python` code snippets representing user actions and execution outputs (§2.1.2); and (3) context-dependent instructions, dynamically rendered to provide additional task guidance and entity definitions as the conversation progresses (§2.1.3).

2.1.1 Header

At the start of the conversation, the header (Figure 1, top) provides instructions prompting the model to identify the API that aligns with the user's intent and extract any arguments specified by the user (Figure 6a, App. B.1). Schema API definitions are presented next, linearised as `python` function signatures (Figure 1, lines 4–14). Each API name is followed by an *intent description* summarising its function (line 5), while arguments are annotated with their types and descriptions (lines 6–10). For *categorical slots*, which take *closed values* from a predefined set, the argument descriptions are prefixed with the list of valid options (line 10). Return types specify the entities produced by APIs, excluding their properties (line 12), which are displayed dynamically as entities are returned (§2.1.3).

As tasks are completed, the header is updated

[2] See Appendix A.1 for a detailed description.

[3] For example, providing a slot value or requesting information about a knowledge base item retrieved by the system.

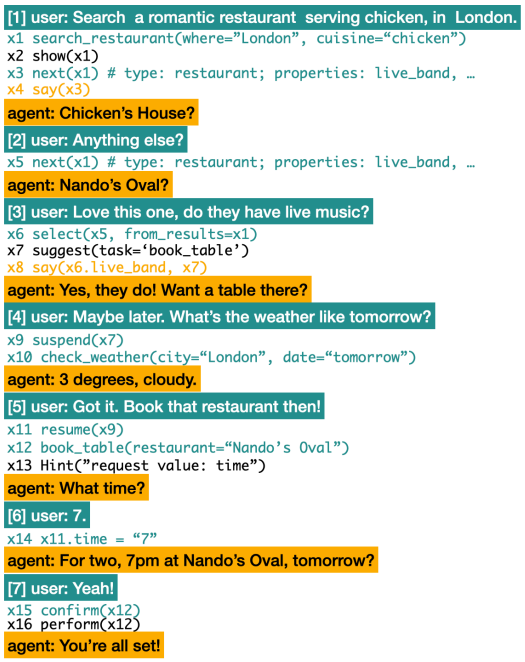

Figure 2: Transcript example. User actions, expressed in natural language, are parsed as program statements which implement a pre-determined dialogue policy. Their execution results - recommended *system actions* (x2, x7, x13, x16) - are executed via `say`, which calls the call the NLG module. The NLG module generates the next agent utterance given information contained in the objects passed as positional arguments to the `say`. NLG calls are shown only for turns 1 and 3, for clarity.

with descriptions of completed tasks and relevant entity definitions (Figure 6b, App. B.1).

2.1.2 Session transcript

The session transcript records the dialogue history, interleaved with program statements that capture user and system actions (Figure 2). To model incremental updates to the dialogue state according to user actions, PyTOD programs employ *intermediate variables* (Stacey et al., 2024).

User actions PyTOD defines user actions as executable commands that update the dialogue state.

API calls (Figure 2, x1, x10, x12) model task initiation, with their output stored in variables that can later be referenced in *assignments* (x14). The latter are semantic representations of utterances where the user provides task constraints or corrects slot values. *Iterations* (x3, x5) handle search results, fetching the top-ranked item (x3) and allowing the user to navigate through them (x5).

Selection (x6) and *confirmation* (x15) statements track user acceptance of system-provided entities[4] and of slot values provided by the agent. Unlike SOTA DST approaches, which extract slot values from both agent and user utterances, these commands track system-mentioned slots by executing the dialogue policy. Specifically, executing a selection updates the referenced command based on the chosen entity[5] whereas executing a confirmation assigns system-provided values to the corresponding parameters of the referenced command[6].

Variables enable PyTOD to handle complex conversations. For example, context switching is represented by `resume`/`suspend` commands (x9, x11). These commands prevent omission errors when copying parameters from the dialogue history[7]. Unlike SOTA TOD agents, PyTOD mitigates transformers' vulnerability to copying errors (Liu et al., 2024; Barbero et al., 2024) by representing negation, conversation pauses as executable and hierarchical slots as executable commands (App. B.1).

System actions are inserted into the transcript by the DM to provide additional cues for state tracking. For example, database call markers (Figure 2, x2) signal the retrieval of results, guiding the language model to predict subsequent iteration (x3, x5) and selection (x6) statements. Additionally, *Hint messages* (x13) are recommended system actions showing missing *required slots*[8], and the DM uses these actions to verify AP output (§2.2.3). Additional system actions are described in App. B.1.

2.1.3 Context-dependent instructions

Context-dependent instructions list properties of retrieved entities along with natural language descriptions of their meaning. This allows PyTOD to answer user queries while reducing system latency[9]. These instructions appear after iteration or confirmation calls and may also include system policy details relevant to state tracking (App. B.1).

2.2 Dialogue manager

The DM coordinates interaction with the user by taking actions recommended by the dialogue policy upon AP output execution. To ensure successful ex-

[4]For example, selecting a restaurant (Figure 2, turn 3).

[5]Executing `select(x5, from_results=x1)` is equivalent to `search_restaurant.restaurant="Nando's Oval"`.

[6]Executing `confirm(x12)` is equivalent to `book_table.people=2, book_table.date="tomorrow"`.

[7]An omission of the `restaurant` parameter in the `book_table` call (Figure 2) can be corrected since the name of the restaurant is known after `select` (x6) execution.

[8]These are all the call parameters that must be specified for the API to perform its intended function.

[9]Since generation latency increases with prompt length, entity definitions are only included when relevant—after the entities have been retrieved—to optimise efficiency.

ecution, it constrains and validates the AP outputs as described in the reminder of this section.

2.2.1 AP output constraints

AP-generated statements must be valid `python` expressions with declared variables to execute. The DM enforces these constraints, inserting `parse_errors` into the transcript if parsing fails. Additionally, it restricts generated API names to those listed in the AP header, correcting errors by minimizing edit distance. Once constrained, statements are executed. If the AP predicts a slot name outside the decoding schema, the DM invokes its SS component to constrain it accordingly.

2.2.2 Schema supervisor

The SS generates a prompt using the schema and the AP output, based on three generic templates (App. B.2.1). It follows a multiple-choice question answering (MQA) format (Figure 3), where the question and options depend on the AP error. If the AP predicts an unknown slot, the prompt lists the all the slot names in the schema with descriptions[10] as answer choices, instructing the model to select the option corresponding to the slot which best matches the AP output (Figure 3a). If the slot name is unknown but its value is listed in the schema, the prompt includes only categorical slot definitions and possible values (Figure 3b). For cases where the AP outputs a slot from a training schema but not the current task schema, the model is presented with slot descriptions from the task schema and instructed to select the closest paraphrase (Figure 3c).

2.2.3 Parser supervisor

The DM invokes the PS when SS-constrained AP assignment expressions do not provide values for slots PyTOD previously requested from the user. In response, the PS generates a prompt using the schema and dialogue history, based on a simple template (App. B.3). The prompt follows an extractive QA format (Figure 4), where the questions correspond to the schema descriptions of the omitted slots (e.g., *Departure date of the flight?* in Figure 1); the context for answering them is limited to the dialogue history of the current task.

The PS corrects slot omissions and semantic errors. First, it extracts values for all requested slots missing from the AP output. Then, for open-value slots, if a predicted answer exactly matches a value already assigned to a slot in the constrained AP output, the system assumes a semantic error and replaces the predicted slot name with the corresponding omitted one. If no such match is found, the PS appends assignment expressions that bind the predicted answers to the omitted slots and updates

[10] Descriptions are replaced with data type for integer-valued slots and value enumeration for categorical slots.

```
Given the definitions, which keyword matches 'to_city'?

    - a) origin_city: flight departure city
    - b) destination_city: city in which the journey ends
    - c) number_of_bags: int
    - d) ticket_fare_class: economy|business
    - e) none: the definitions do not describe 'to_city'

Answer:
```

(a) Unknown slot name. The *SS output* is converted to destination_city, replacing to_city in the AP output.

```
Here are some definitions:

    - subtitle_language: language to use for subtitles (or none)

Given these, 'subtitle_free = true' is a synonym of:

    - a) subtitle_language = none
    - b) subtitle_language = english
    - c) options do not describe 'subtitle_free = true'

Answer:
```

(b) Unknown slot name (closed value). The *SS output* is converted to subtitle_language = none, replacing subtitle_free = true in the AP output.

```
Which sentence paraphrases 'date of car pickup (pickup)'?

    - a) the first date to start using the rental car
    - b) the date to return the car
    - c) no option above paraphrases 'pickup'

Answer:
```

(c) Memorised slot name. The *SS output* is converted to pickup_date, replacing pickup in the AP output.

Figure 3: Illustration of prompts generated by the SS to constrain AP generations to the decoding schema.

```
Q: Answer the following questions. Output "unanswerable"
if the question cannot be answered given the conversation.

In the conversation:

...
user: Great, now search one way flights out of Mexico City.
agent: Departure date and to which city?
user: Leaving the 7th of this month to San Diego.

1) city in which the journey ends?
2) date of departure flight?

Answer:
```

Figure 4: Illustration of the prompt generated by the PS to handle AP omissions and semantic errors. The expected SS output is 1) San Diego 2) 7th of this month. The second answer is used to correct an omission by extending the constrained AP output in Figure 1 (x1.destination_city = "San Diego") with x1.departure_date = "7th of this month". The ellipsis indicates truncated, irrelevant dialogue history.

the current task (viz *departure_date*, Figure 1).

3 Experimental Setup

We evaluate PyTOD on schema-guided DST. While Zhao et al. (2023) also evaluate their system on next-action prediction, PyTOD is designed such that, given the correct dialogue state, it is guaranteed to take a correct action according to the dialogue policy. As a result, next-action prediction performance is correlated with DST performance and does not provide additional insight into system behavior. Turn-based evaluations often fail to reflect a system's ability to satisfy complex user goals in real interactions, as demonstrated by Takanobu et al. (2020) and recently by Elizabeth et al. (2024).

3.1 Datasets and metrics

Datasets SGD consists of $21,106$ dialogues spanning 26 *service schemata*[11] in the training split. The test set comprises $4,201$ dialogues covering 21 service schemata. Designed to assess TOD generalization, the dataset is challenging: among the 90 distinct task sequences in the test set, 85.6% involve a task grounded in a schema *unseen*[12] during training, corresponding to 77% of the dialogues. Additionally, the dialogues exhibit complex conversational phenomena such as context switching, cross-turn corrections, and frequent goal changes, making DST particularly demanding.

Metrics We evaluate performance using *joint goal accuracy (JGA)*, which measures the percentage of dialogue turns where all slot-value pairs are predicted correctly. To assess generalization, we report JGA separately for seen and unseen services. This distinction highlights each model's ability to generalize to unseen slots and values and to correctly interpret API descriptions that were not encountered during training. We compute JGA using the official evaluator[13] but extend the SGD state annotations (App. C.1) for fair comparison.

While JGA evaluates slot extraction accuracy, it does not account for state consistency across turns. If a slot predicted in an earlier turn is later omitted when the state is re-estimated, the agent may unnecessarily prompt the user to repeat previously provided information, degrading interaction quality. We introduce C-JGA, a stricter metric that enforces state consistency: a turn contributes to the JGA only if the state at all previous turns in the same task is jointly correct.

[11] A service expose multiple APIs representing user intents.

[12] Unseen APIs may either introduce new functionality within known domains or belong to entirely new domains.

[13] Available at https://bit.ly/3B7jD1c.

3.2 Baselines

T5DST (Lee et al., 2022) jointly encodes the dialogue history alongside a slot description to generate the corresponding slot value. Each slot is processed independently, so T5DST requires multiple forward passes per turn - one for each slot in the schema. **D3ST** (Zhao et al., 2022) optimises T5DST by predicting all active slot values in a single pass. **SDT-Seq** (Gupta et al., 2022) takes a demonstration-based approach, encoding the dialogue history alongside a sample conversation and its target state sequence to learn DST via in-context finetuning. We replicate SDT-Seq and D3ST (App. C.2).

3.3 State Tracking with PyTOD

API retrieval Consistent with prior work on SGD and our baselines, we assume that the services the user interacts with are known at each turn. Accordingly, PyTOD retrieves APIs from the AP header (§2.1.1) and not the entire assistant schema. In contrast, user actions are tied to the dialogue policy and are learned during finetuning rather than being explicitly defined in the prompt.

Action parser We execute the AP-generated programs using `pytodlib` (App. D.1). The library simulates: (1), the 58 APIs in the SGD development and test sets, complete with simulated databases and API responses; (2) the system policy of all 88 SGD APIs; (3) user actions (§2.1.2, App. B.1). An execution engine runs PyTOD programs in a sandbox environment, tracking the dialogue state. We open-source `pytod-lib`, addressing the limitations of popular resources like MultiWOZ (Budzianowski et al., 2018) (App. D.2) and providing a high-quality resource (Lu et al., 2024; Farn and Shin, 2023) for evaluating LLMs' ability to engage in complex, goal-oriented, conversations.

Schema supervisor The SS constrains the AP output based on the decoding schema (Figure 3). We use MQA prompts and PLM knowledge for *zero-shot schema supervision* with FlanT5 (3B) (Chung et al., 2024), eliminating the need for slot paraphrase collection. This makes PLM-guided constrained decoding simple to implement.

Parser supervisor The PS corrects slot omissions and semantic errors (Figure 4), with finetuning prompts constructed from the same dialogues

as those used for AP finetuning. We perform *multitask learning* for action parsing and parsing supervision. This allows PyTOD to function as a single model which performs corrections on DM request, with schema supervision handled by an off-the-shelf PLM. See training details in App. C.

4 Main Result

Since TOD agents are often deployed in resource-constrained settings, we implement PyTOD's AP and PS components using a small PLM and compare against SOTA models of similar size. Table 1 shows that PyTOD closely matches or outperforms all baselines, while also achieving higher consistency across turns. In particular, PyTOD (B) surpasses D3ST (#2) and T5DST by absolute margins of 5.6% and 4.2%, respectively, with stronger performance on unseen services. PyTOD (L) achieves a similar improvement over D3ST (#7&11).

Size	Model	JGA	C-JGA	JGA (Seen)	JGA (Unseen)	#
220M	D3ST (Zhao et al., 2022)	72.9	–	92.5	66.4	1
	D3ST (Flan-T5, ours)†	71.2	62.2	93.2	63.8	2
	T5DST (Lee et al., 2022)	72.6	–	89.7	66.9	3
	SDT-Seq (Gupta et al., 2022)	76.3	–	–	–	4
	SDT-Seq (Flan-T5, ours)†	77.5	68.7	93.5	72.2	5
780M	D3ST (Zhao et al., 2022)	80.0	–	93.8	75.4	6
	D3ST (Flan-T5, ours)†	76.5	67.7	93.8	70.8	7
	SDT-Seq (Gupta et al., 2022)	83.3	–	–	–	8
	SDT-Seq (Flan-T5, ours)†	82.7	74.2	94.1	78.9	9
220M	PyTOD (Base)	76.8	72.7	91.0	71.8	10
780M	PyTOD (Large)	82.2	78.4	92.1	78.9	11

Table 1: PyTOD DST performance. Rows marked with † report the results of our replication study in App. C.2.

Both D3ST and T5DST re-estimate the entire dialogue state at each turn, meaning their JGA can increase when early errors are later corrected. In real-world interactions, however, these errors would alter the dialogue flow, so the JGA increases due to error recovery overestimate real-world performance. When adjusted for consistency, D3ST JGA drops sharply: 9% (220M, #2) and 8.8% (780M, #7). In contrast, PyTOD incrementally predicts user actions based on its *predicted* past actions, resulting in smaller JGA drops of 4.1% (Base, #10) 3.8% (Large, #11). These smaller decreases stem from two key factors. First, when users change goals mid-task, incorrect states are sometimes corrected. Second, parameters incorrectly copied from previous tasks at the start of a new task are later overridden by their correct values.

PyTOD retrieves the correct API from the prompt whiles D3ST and T5DST JGA is invariant to intent parsing errors, making the figures less sensitive to annotation errors (§5.3) but less reflective of real-world TOD performance. PyTOD's JGA more accurately reflects practical deployment scenarios where intent errors impact conversation quality.

Size	PS	SS	Multitask	JGA	JGA Seen	JGA Unseen	#
220M	✓	✓	✓	76.8	91.0	72.1	1
	✗	✓	✗	75.8	90.1	71.0	2
	✗	✗	✓	64.2	88.2	56.2	3
	✗	✗	✗	64.4	88.8	56.3	4
780M	✓	✓	✓	82.2	92.1	78.9	5
	✗	✓	✗	80.6	91.5	77.0	6
	✗	✗	✓	74.6	90.8	69.2	7
	✗	✗	✗	74.1	90.4	68.6	8

Table 2: Contribution of parser supervisor (PS) and schema supervisor (SS) to PyTOD performance. **Multitask** indicates joint training of the action parsing and parsing supervision tasks. Rows 10&11 from Table 1 are repeated in rows 1&5 to facilitate comparisons.

Descriptions vs demonstrations Beyond surpassing schema-guided baselines, PyTOD performs competitively with approaches requiring additional developer effort, such as SDT-Seq. The JGA gap between PyTOD and SDT-Seq is minimal: 0.7% (Base, #5&10) and 0.5% (Large, #9&11), with SDT-Seq performing better on seen domains.

However, SDT-Seq requires developers to manually craft example demonstrations for each intent in addition to service schemas, increasing deployment overhead. Furthermore, SDT-Seq does not perform API retrieval, requiring an external intent detection model. Finally, PyTOD demonstrates substantially higher consistency, outperforming SDT-Seq by 5.0% (#5&10) and 4.2% (#9&11) in C-JGA, reinforcing its stability in multi-turn interactions.

5 Analysis and Discussion

5.1 Ablation study

Parser supervisor The PS improves PyTOD performance by an average of 1.0% (Base, #1&2) and 1.6% (Large, #5&6). Analysing 210 errors corrected by the PS in the best-performing PyTOD (Large) run (82.6% JGA) reveals that slot omissions occur more frequently than semantic errors (61% vs. 39%). The most common semantic error stems from confusion between similar slots (e.g., start date vs. end date). Jointly training the model for action parsing and parsing supervision has a negligible effect on parsing accuracy (#3&4, #7&8), simplifying PyTOD deployment.

Schema supervisor While the AP occasionally generates slot names absent from the decoding schema, the predicted slots often retain the correct semantics. For instance, the AP produces new

AP Size	SS Size	PS	JGA	JGA Seen	JGA Unseen	#
220M	3B	✓	76.8 (+5.6%)	91.0	72.1	1
		✗	75.8 (+4.6%)	90.1	71.0	2
	780M	✓	75.1 (+3.9%)	89.9	70.2	3
		✗	71.3 (+0.1%)	88.8	65.4	4
	220M	✓	74.1 (+2.9%)	89.7	68.8	5
		✗	70.8 (-0.4%)	89.0	64.7	6
780M	3B	✓	82.2 (+5.7%)	92.1	78.9	7
		✗	80.6 (+4.1%)	91.5	77.0	8
	780M	✓	81.2 (+4.7%)	91.2	77.9	9
		✗	78.6 (+2.1%)	90.3	74.7	10
	220M	✓	80.5 (+4.0%)	91.2	76.9	11
		✗	77.9 (+1.4%)	90.3	73.7	12

Table 3: PyTOD performance as function of SS size. Checkmarks (✓) and crosses (✗) indicate PS presence or absence, respectively. Rows 10 and 11 from Table 1 are repeated in rows 1 and 7, for easy comparison. Bracketed numbers represent absolute improvements with respect to our implementation of D3ST (Table 1, #2&7)

AP Size	SS Size	Samples per sec	Runtime (sec)	Relative Latency	#
220M	-	9.110	5875.12	1.00	1
	220M	6.816	7919.79	1.35	2
	780M	6.766	7946.26	1.35	3
	3B	6.323	8588.64	1.46	4
780M	-	3.331	16269.70	1.00	5
	220M	2.971	18300.93	1.12	6
	780M	2.964	18341.15	1.13	7
	3B	2.903	18771.32	1.15	8

Table 4: Latency of AP size and SS size, reported for the test set. Numbers reported are for models without PS (marked with ✗ in Table 3). Each figure is an average of three runs of making predictions for ≈ 53k test set turns on an NVIDIA A100 GPU.

slot names such as *number_passengers* instead of *num_passengers* or *travel_starts_from* instead of *journey_starts_from*. Other times, the AP outputs slots seen in a training schema implementing the same domain as an (unseen) decoding schema (e.g., outputs *hotel_name* instead of *place_name*), reflecting real-world challenges where TOD agents must support integration of services similar to the ones they have been trained on without further finetuning. The SS effectively mitigates both of these challenges: it improves JGA by 11.4% for PyTOD (Base) (#2&3) and 6.0% for PyTOD (Large) (#6&7). Notably, the SS reduces system latency while improving performance: the 220M AP+SS system (#2) runs 1.89 times faster compared to the 780M AP (#8) while being 1.7% more accurate.

Beyond performance gains, PLM-constrained decoding simplifies deployment compared to grammar-based approaches. Unlike the latter, which require re-engineering to align with new backbone PLM tokenization schemes, PLM-based constrained decoding allows seamless AP updates. Grammar-constrained decoding poses an additional challenge for PyTOD: some of its grammar rules depends on dynamically generated variables[14] and have to be generated dynamically during inference.

5.2 Schema supervisor analysis

Table 3 shows that PLM-constrained decoding remains effective even as the SS sizes decreases: JGA drops by only 2.7% (#1&5) for PyTOD (Base) and 1.7% (#7&11) for PyTOD (Large) when reducing SS from 3B to 220M parameters. The PS recovers some of the errors, improving JGA by an average of 3.55% (PyTOD Base) and 2.6% (PyTOD Large). Regardless of the AP, SS and PS configurations PyTOD consistently outperforms D3ST (Table 3).

Table 4 confirms that SS size has a minimal impact on PyTOD latency. Expanding SS from 220M to 3B increases latency by only 11% for PyTOD (Base) and just 2% for PyTOD (Large). This is expected since the SS prompts are short and the MQA formatting enables SS to constrain decoding with a single token. Most of the latency increase stems from on-demand SS model loading, which can be optimized by keeping SS in memory at the cost of a higher memory footprint. As discussed in App. E.1, PyTOD can be further optimised by decoupling the PS and AP.

5.3 Error analysis

Seen services An analysis of 20 dialogues from each of the 3 services with JGA below SDT-Seq average, reveals that *RideSharing_2 (RS_2)* and *Movies_1 (MOV_1)* contribute most to the discrepancy. For *RS_2*, 70% of the errors involve incorrect slot values: the system consistently misinterprets requests like *cheapest ride* as *regular* instead of *pool*, suggesting that PyTOD could benefit by implementing the AP and PS with a PLM that has stronger world knowledge. For *MOV_1*, value errors are primarily *span errors*, where the model fails to capture the full movie title or crosses span boundaries. Notably, annotation errors where the annotated slot values are absent from the user utterance largely explain the *Travel_1* performance.

Unseen services Annotation errors significantly impact unseen service performance, particularly in *Trains_1*, where intent paraphrase errors (App. E.2) prevent PyTOD from tracking state due to its reliance on retrieved train schedules. When

[14] For example, `confirm`'s sole positional argument can only be a variable bound to a transaction API call.

Error Type	RideSharing_2	Movies_1	Travel_1	RentalCars_3	Trains_1	Music_3	Messaging_1	Total
	78.2 (-13.9%)	80.7 (-11.4%)	92.5 (+0.4%)	55.3 (-23.6%)	60.2 (-18.7%)	64.0 (-14.9%)	65.7 (-13.2%)	–
	✓	✓	✓	✗	✗	✗	✗	–
Missed	15.0(3)	5.9(2)	9.5(2)	3.7(1)	9.4(3)	20.0(5)	–	16(9.2)
Value	70.0(14)	32.4(11)	–	22.2(6)	9.4(3)	36.0(9)	–	43(24.9)
Copy	15.0(3)	29.4(10)	4.8(1)	48.1(13)	21.9(7)	28.0(7)	78.6(11)	52(30.1)
Annot.	–	17.6(6)	71.4(15)	14.8(4)	40.6(13)	16.0(4)	21.4(3)	45(26.0)
Intent	–	–	–	–	9.4(3)	–	–	3(1.7)
Other	–	14.7(5)	14.3(3)	11.1(3)	9.4(3)	–	–	14(8.1)
Total	20	34	21	27	32	25	14	173(100)

Table 5: Error analysis for the best PyTOD (Large) run, including seen (✓) and unseen (✗) services. Second row indicates service JGA and its absolute deviation with respect to the average seen/unseen JGA in Table 1 (row 11). Percentages indicate the fraction of total service errors and (raw counts) are shown. **Missed**=omitted slot, **Value**=incorrect value predicted, **Copy**=copied incorrect value/did not copy value, **Annot.**=annotation error, **Intent**=incorrect intent.

search tasks succeed, *copy errors* - where PyTOD fails to propagate query parameters (e.g., *album* from *LookupMusic* to *PlayMedia*) reduce JGA in *Movie_3* and *RentalCars_3*. For example, while *GetCarsAvailable* succeeds 90% of the time, *ReserveCar* succeeds in just 26.23% of cases. This highlights systematic failures which could be addressed through targeted improvements (App. E.3).

In *Messaging_1* copy errors stem from incorrect co-reference resolution: the *location* slot is incorrectly resolved to cities instead of addresses, due to biases in training data where other services use this slot for city names. TOD agents that support multiple services are prone to similar semantic mismatches when concept names differ across training schemas, and fine-tuning alone can lead to overfitting to schema-specific naming conventions. Future work will explore enhanced supervision mechanisms to mitigate these biases.

6 Related Work

Few- and zero-shot DST Our approach and baselines (§3.2) extend prior transfer learning work on cross-domain DST generalization via schema descriptions (Lin et al., 2021b) and QA tasks (Lin et al., 2021a). Recently, large-scale proprietary LLMs (e.g., ChatGPT (OpenAI, 2022)) have shown strong DST performance with no (Heck et al., 2023) or few (Li et al., 2024; Hu et al., 2022; Feng et al., 2023; Wu et al., 2023, *inter alia*) training dialogues. Like PyTOD, these approaches predict state updates, represented in JSON format (Wu et al., 2023), as SQL queries (Hu et al., 2022), function calls (Stricker and Paroubek, 2024; Li et al., 2024) or code (King and Flanigan, 2023). Unlike PyTOD, they are not policy-guided nor do they operate in a simulated environment, tracking slots from both user and agent utterances - a limitation that degrades performance (Wu et al., 2023).

These methods focus on data-efficient DST, relying on LLMs at inference. Recent works (Kulkarni et al., 2024; Finch and Choi, 2024) use LLMs for data generation, improving generalization without costly per-turn inference and addressing concerns over cost, resource availability, and privacy (Heck et al., 2023; Feng et al., 2023). Distillation (Lee et al., 2024c; Dong et al., 2024) and data augmentation (Feng et al., 2023; Li et al., 2024) provide alternatives, though they still depend on billion-scale models. To improve efficiency, subsequent works (Lee et al., 2024a,b) employ self-correction (Xie et al., 2022; Tian et al., 2021) to maintain accuracy. PyTOD targets stricter resource constraints, achieving accurate and consistent DST with models an order of magnitude smaller. It eschews corrector fine-tuning or feedback generation, using readily available execution feedback and simple QA prompts for zero-shot correction with one token. Its policy- and environment-driven corrections avoid per-turn supervision, improving efficiency.

TOD agents Transfer learning (Zhao et al., 2023) and prompting (Shu et al., 2022; Zhang et al., 2023; Li et al., 2023; Saley et al., 2024, *inter alia*) also apply to TOD agent development. `pytodlib` provides challenges beyond MultiWOZ through policy (App. D.2) and ontology complexity (Hudeček and Dusek, 2023; Lee et al., 2024a; Gao et al., 2024), providing a testbed for advancing TOD agents while addressing the dearth of conversational tool-use corpora (Lu et al., 2024).

7 Conclusion

We introduced PyTOD, a TOD agent that generates code incrementally and tracks dialogue state through execution, leveraging DM and execution feedback to achieve SOTA DST performance on the SGD benchmark. By coupling state tracking with execution, PyTOD shows enhanced cross-turn consistency and thus improved real-world reliability. We release `pytod-lib`, a simulation grounding the SGD dataset, to advance research on zero-shot TOD agents and conversational tool use. Future work will focus on enhancing robustness to copy errors and integrating PyTOD with LLMs for zero- and few-shot end-to-end dialogue modeling.

Limitations

API retrieval Like most state-of-the-art approaches on the SGD and MultiWOZ datasets, PyTOD assumes knowledge of the service schema at each turn (but not the API). In real-world scenarios, however, virtual assistants must first infer the schema before tracking dialogue state. This is particularly challenging in SGD, where multiple services within the same domain exhibit fine-grained differences. For instance, both `Buses_1` and `Buses_2` both implement the `FindBus` APIs, yet disambiguation between the two services is only possible when users mention `fare_type` an *optional* slot. Similarly, `Buses_3` (test set), can only be distinguished if the `category` slot is provided. As discussed in §5.3, PyTOD relies on accurate intent parsing, making it susceptible to service disambiguation errors that will degrade performance.

Schema robustness We have not explicitly evaluated PyTOD's robustness to linguistic schema variations, which are known to affect transfer learning-based DST systems. While AP accuracy will degradoae, PyTOD's SS - the primary contributor to its performance (§5.1) - is expected to mitigate this impact since it performs zero-shot corrections using MQA prompts. Future work could enhance AP and PS robustness using transfer learning from QA task (Lin et al., 2021a; Cho et al., 2023), knowledge-seeking turn grounding (Coca et al., 2023a) or synthetic schemas (Coca et al., 2023b), none of which require additional annotation.

Prompt optimisation While PS and AP latency are optimized through dialogue history truncation and minimal generation length, the AP prompt itself remains unoptimised. The AP header already maintains a stack of completed tasks and retrieved entities, effectively summarising the dialogue history. However, our prompts contain both the header and transcript, introducing redundancy and increasing system latency.

Additionally, the API header presents an opportunity for personalized conversational intelligence. Embedding and storing it in a vector database could enable retrieval across multi-session conversations, improving continuity and personalization. We will explore this in future work.

Grammar-constrained decoding We opted for deep learning approach to constraining AP output to the schema due to the complexity of working with dynamically generated grammar rules needed to constrain decoding to a set of valid tokens. We considered this advanced optimisation to be a research topic that future work could address.

Interactive evaluation Our results demonstrate that state-of-the-art DST models exhibit stability errors when evaluated using C-JGA. As DST systems continue to improve, we believe that evaluating models in real-world user interactions will be essential to assessing their practical viability.

Dataset choice As discussed in §3.3 and §6, we conduct our experiments on the SGD, which includes all the MultiWOZ domains, features more complex conversation flows challenges LLMs. Morover, the MultiWOZ policy depends on unobservable factors such as wizard's task interpretation (Mosig et al., 2020b). Through `pytod-lib` we provide a resource for developing agents capable of following pre-defined policies in a more demanding and controlled settings.

References

Federico Barbero, Andrea Banino, Steven Kapturowski, Dharshan Kumaran, João G. M. Araújo, Alex Vitvitskyi, Razvan Pascanu, and Petar Velickovic. 2024. Transformers need glasses! information over-squashing in language tasks. *CoRR*, abs/2406.04267.

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.

Hyundong Cho, Andrea Madotto, Zhaojiang Lin, Khyathi Chandu, Satwik Kottur, Jing Xu, Jonathan May, and Chinnadhurai Sankar. 2023. Continual dialogue state tracking via example-guided question answering. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3873–3886, Singapore. Association for Computational Linguistics.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Y. Zhao, Yanping Huang, Andrew M. Dai,

Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. 2024. Scaling instruction-finetuned language models. *J. Mach. Learn. Res.*, 25:70:1–70:53.

Alexandru Coca, Bo-Hsiang Tseng, Jinghong Chen, Weizhe Lin, Weixuan Zhang, Tisha Anders, and Bill Byrne. 2023a. Grounding description-driven dialogue state trackers with knowledge-seeking turns. In *Proceedings of the 24th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 444–456, Prague, Czechia. Association for Computational Linguistics.

Alexandru Coca, Bo-Hsiang Tseng, Weizhe Lin, and Bill Byrne. 2023b. More robust schema-guided dialogue state tracking via tree-based paraphrase ranking. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 1443–1454, Dubrovnik, Croatia. Association for Computational Linguistics.

Xiaoyu Dong, Yujie Feng, Zexin Lu, Guangyuan Shi, and Xiao-Ming Wu. 2024. Zero-shot cross-domain dialogue state tracking via context-aware auto-prompting and instruction-following contrastive decoding. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 8527–8540, Miami, Florida, USA. Association for Computational Linguistics.

Michelle Elizabeth, Morgan Veyret, Miguel Couceiro, Ondrej Dusek, and Lina Maria Rojas-Barahona. 2024. Do large language models with reasoning and acting meet the needs of task-oriented dialogue? *CoRR*, abs/2412.01262.

Nicholas Farn and Richard Shin. 2023. Tooltalk: Evaluating tool-usage in a conversational setting. *CoRR*, abs/2311.10775.

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, EMNLP 2023, Singapore, December 6-10, 2023*, pages 739–755. Association for Computational Linguistics.

James D. Finch and Jinho D. Choi. 2024. Diverse and effective synthetic data generation for adaptable zero-shot dialogue state tracking. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 12527–12544, Miami, Florida, USA. Association for Computational Linguistics.

Haoyu Gao, Ting-En Lin, Hangyu Li, Min Yang, Yuchuan Wu, Wentao Ma, Fei Huang, and Yongbin Li. 2024. Self-explanation prompting improves dialogue understanding in large language models. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy*, pages 14567–14578. ELRA and ICCL.

Raghav Gupta, Harrison Lee, Jeffrey Zhao, Yuan Cao, Abhinav Rastogi, and Yonghui Wu. 2022. Show, don't tell: Demonstrations outperform descriptions for schema-guided task-oriented dialogue. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*, pages 4541–4549. Association for Computational Linguistics.

Michael Heck, Nurul Lubis, Benjamin Ruppik, Renato Vukovic, Shutong Feng, Christian Geishauser, Hsien-chin Lin, Carel van Niekerk, and Milica Gasic. 2023. ChatGPT for zero-shot dialogue state tracking: A solution or an opportunity? In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 936–950, Toronto, Canada. Association for Computational Linguistics.

Yushi Hu, Chia-Hsuan Lee, Tianbao Xie, Tao Yu, Noah A. Smith, and Mari Ostendorf. 2022. In-context learning for few-shot dialogue state tracking. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 2627–2643. 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. 2022. « est-ce que tu me suis ? » : une revue du suivi de l'état du dialogue ("do you follow me ?" : a review of dialogue state tracking). In *Actes de la 29e Conférence sur le Traitement Automatique des Langues Naturelles. Volume 2 : 24e Rencontres Etudiants Chercheurs en Informatique pour le TAL (RECITAL)*, pages 1–19, Avignon, France. ATALA.

Brendan King and Jeffrey Flanigan. 2023. Diverse retrieval-augmented in-context learning for dialogue state tracking. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 5570–5585, Toronto, Canada. Association for Computational Linguistics.

Atharva Kulkarni, Bo-Hsiang Tseng, Joel Ruben Antony Moniz, Dhivya Piraviperumal, Hong Yu, and Shruti Bhargava. 2024. SynthDST: Synthetic data is all you need for few-shot dialog state tracking. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1988–2001, St. Julian's, Malta. Association for Computational Linguistics.

Chia-Hsuan Lee, Hao Cheng, and Mari Ostendorf. 2024a. Correctionlm: Self-corrections with SLM for dialogue state tracking. *CoRR*, abs/2410.18209.

Chia-Hsuan Lee, Hao Cheng, and Mari Ostendorf. 2024b. OrchestraLLM: Efficient orchestration of language models for dialogue state tracking. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 1434–1445, Mexico City, Mexico. Association for Computational Linguistics.

Harrison Lee, Raghav Gupta, Abhinav Rastogi, Yuan Cao, Bin Zhang, and Yonghui Wu. 2022. SGD-X: A benchmark for robust generalization in schema-guided dialogue systems. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 10938–10946. AAAI Press.

Seanie Lee, Jianpeng Cheng, Joris Driesen, Alexandru Coca, and Anders Johannsen. 2024c. Effective and efficient conversation retrieval for dialogue state tracking with implicit text summaries. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 96–111, Mexico City, Mexico. Association for Computational Linguistics.

Zekun Li, Zhiyu Chen, Mike Ross, Patrick Huber, Seungwhan Moon, Zhaojiang Lin, Xin Dong, Adithya Sagar, Xifeng Yan, and Paul A. 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), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 8688–8704. Association for Computational Linguistics.

Zekun Li, Baolin Peng, Pengcheng He, Michel Galley, Jianfeng Gao, and Xifeng Yan. 2023. Guiding large language models via directional stimulus prompting. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Zhaojiang Lin, Bing Liu, Andrea Madotto, Seungwhan Moon, Zhenpeng Zhou, Paul Crook, Zhiguang Wang, Zhou Yu, Eunjoon Cho, Rajen Subba, and Pascale Fung. 2021a. Zero-shot dialogue state tracking via cross-task transfer. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7890–7900, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Zhaojiang Lin, Bing Liu, Seungwhan Moon, Paul Crook, Zhenpeng Zhou, Zhiguang Wang, Zhou Yu, Andrea Madotto, Eunjoon Cho, and Rajen Subba. 2021b. Leveraging slot descriptions for zero-shot cross-domain dialogue StateTracking. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5640–5648, Online. Association for Computational Linguistics.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Trans. Assoc. Comput. Linguistics*, 12:157–173.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. 2024. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. *CoRR*, abs/2408.04682.

Johannes E. M. Mosig, Shikib Mehri, and Thomas Kober. 2020a. STAR: A schema-guided dialog dataset for transfer learning. *CoRR*, abs/2010.11853.

Johannes E. M. Mosig, Vladimir Vlasov, and Alan Nichol. 2020b. Where is the context? - A critique of recent dialogue datasets. *CoRR*, abs/2004.10473.

Arvind Neelakantan, Semih Yavuz, Sharan Narang, Vishaal Prasad, Ben Goodrich, Daniel Duckworth, Chinnadhurai Sankar, and Xifeng Yan. 2019. Neural assistant: Joint action prediction, response generation, and latent knowledge reasoning. *CoRR*, abs/1910.14613.

OpenAI. 2022. Introducing ChatGPT. `https://openai.com/blog/chatgpt`.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2024. Tool learning with large language models: A survey. *CoRR*, abs/2405.17935.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67.

Abhinav Rastogi, Xiaoxue Zang, Srinivas Sunkara, Raghav Gupta, and Pranav Khaitan. 2019. Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. *CoRR*, abs/1909.05855.

Vishal Vivek Saley, Rocktim Jyoti Das, Dinesh Raghu, and Mausam . 2024. Synergizing in-context learning with hints for end-to-end task-oriented dialog systems. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 5596–5612, Miami, Florida, USA. Association for Computational Linguistics.

Raphael Shu, Elman Mansimov, Tamer Alkhouli, Nikolaos Pappas, Salvatore Romeo, Arshit Gupta, Saab Mansour, Yi Zhang, and Dan Roth. 2022. Dialog2api: Task-oriented dialogue with API description and example programs. *CoRR*, abs/2212.09946.

Joe Stacey, Jianpeng Cheng, John Torr, Tristan Guigue, Joris Driesen, Alexandru Coca, Mark Gaynor, and Anders Johannsen. 2024. LUCID: LLM-generated utterances for complex and interesting dialogues. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 4: Student Research Workshop)*, pages 56–74, Mexico City, Mexico. Association for Computational Linguistics.

Armand Stricker and Patrick Paroubek. 2024. A few-shot approach to task-oriented dialogue enhanced with chitchat. In *Proceedings of the 25th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 590–602, Kyoto, Japan. Association for Computational Linguistics.

Ryuichi Takanobu, Qi Zhu, Jinchao Li, Baolin Peng, Jianfeng Gao, and Minlie Huang. 2020. Is your goal-oriented dialog model performing really well? empirical analysis of system-wise evaluation. In *Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and Dialogue, SIGdial 2020, 1st virtual meeting, July 1-3, 2020*, pages 297–310. Association for Computational Linguistics.

Xin Tian, Liankai Huang, Yingzhan Lin, Siqi Bao, Huang He, Yunyi Yang, Hua Wu, Fan Wang, and Shuqi Sun. 2021. Amendable generation for dialogue state tracking. In *Proceedings of the 3rd Workshop on Natural Language Processing for Conversational AI*, pages 80–92, Online. Association for Computational Linguistics.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. 2019. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771.

Yuxiang Wu, Guanting Dong, and Weiran Xu. 2023. Semantic parsing by large language models for intricate updating strategies of zero-shot dialogue state tracking. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 11093–11099. Association for Computational Linguistics.

Hongyan Xie, Haoxiang Su, Shuangyong Song, Hao Huang, Bo Zou, Kun Deng, Jianghua Lin, Zhihui Zhang, and Xiaodong He. 2022. Correctable-DST: Mitigating historical context mismatch between training and inference for improved dialogue state tracking. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 876–889, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Xiaoying Zhang, Baolin Peng, Kun Li, Jingyan Zhou, and Helen Meng. 2023. SGP-TOD: building task bots effortlessly via schema-guided LLM prompting. *CoRR*, abs/2305.09067.

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.

Jeffrey Zhao, Raghav Gupta, Yuan Cao, Dian Yu, Mingqiu Wang, Harrison Lee, Abhinav Rastogi, Izhak Shafran, and Yonghui Wu. 2022. Description-driven task-oriented dialog modeling. *CoRR*, abs/2201.08904.

A Background

A.1 AnyTOD

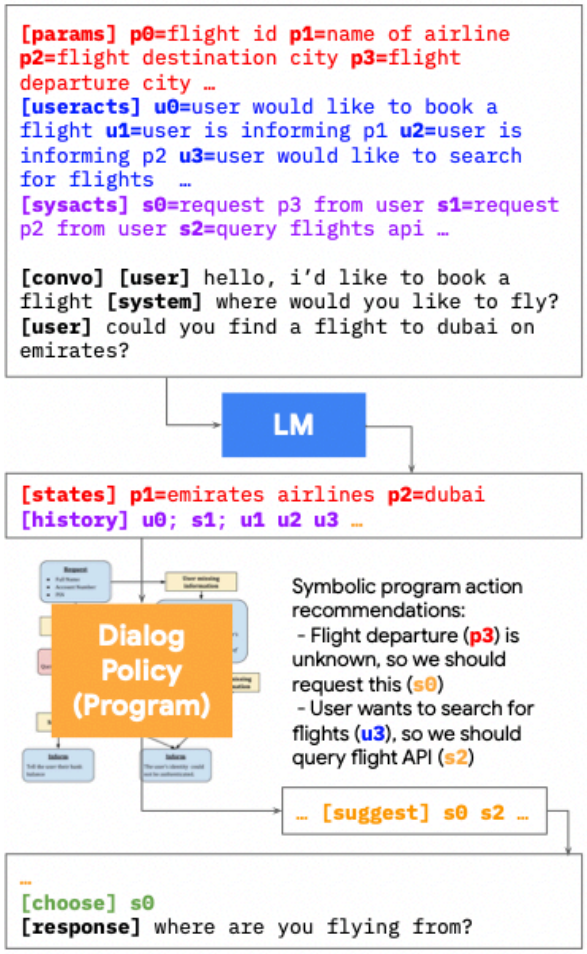

Figure 5: AnyTOD overview. Image reproduced from Zhao et al. (2023) with authors' permission.

Figure 5 shows an overview of AnyTOD (Zhao et al., 2023), a neuro-symbolic end-to-end TOD agent capable of zero-shot transfer to unseen tasks and domains. The prompt contains dialogue history (prefixed by `[convo]` in Figure 5) and a linearised schema (above it). The schema prompt contains three parts, deliniated by the `[params]`, `[useracts]` and `[sysacts]` tokens. The first lists the slot descriptions alongside *slot identifiers* and the following two parts provide natural language descriptions of the user and system actions, alongside *action identifiers*. This prompt is input to a language model (LM), which predicts a sequence of slot identifiers (*state sequence*, in red), followed by a sequence of ;-separated action identifiers (*action sequence*, in purple). The separator indicates turn-boundaries. These sequences are interpreted by the dialogue policy, a program which outputs a sequence of recommended actions given the current context (in orange). This sequence is appended to the prompt along with the state and action sequences, providing context for the language model to select the next action (in green) and generate the agent response in natural language.

B Prompts

B.1 Action parser

Header Figure 6 presents a sample AP prompt header. If no task has been completed, the prompt begins with task instructions (Figure 6a). Once a task is completed, a summary of the task and the returned entity is prepended before the task instructions (Figure 6b). The entity definitions include a docstring instructing the model to retain and copy relevant argument values into subsequent API call paramater values.

```
your task is to identify which of the following tasks
the user wants to complete, and update it accordingly
as the user provides more information during the conversation.
you should also carefully follow any developer instructions
to answer user questions and complete follow-up tasks.
```

(a) Task instruction

```
1. considered one-way air travel options for their itinerary.
the 'flights_3_search_oneway_flight' api returned a
'oneway_flight' object (x11).

class oneway_flight:
    """object returned by the 'flights_3_search_oneway_flight' api.
    in subsequent api calls, you should pass, to compatible
    arguments, the last value mentioned for the following entity
    properties:

    properties
    ----------
    origin_city: str
        city in which the journey originates
    destination_city: str
        city in which the journey ends
    departure_date: str
        date of departure flight
    airlines: enum
        name of airline operating the flight
    passengers: int
        number of passengers to find flight seats for
    flight_class: enum
        fare class of flight booking
    number_checked_bags: int
        number of bags to check in
    """

your task is to identify ...
```

(b) Task stack

Figure 6: Action parser header prompt components.

Session transcript Beyond the user and system actions described in Section 2.1, we introduce additional actions that enable PyTOD to handle more complex user interactions.

User actions Conversation interruptions (Figure 7, turn 7) and negations (turn 9) require TOD agents to copy the dialogue state from the history,

Figure 7: An alternative continuation of the conversation in Figure 2 after turn 6. Additional user/system actions and an example of how PyTOD can handle complex compositional utterances (turn 9) are depicted. Note that unlike in Figure 2 where some of the `say` calls were omitted for clarity, all calls to the NLG module are shown.

making them vulnerable to copying errors. PyTOD mitigates this by generating special program statements (x15, x27), ensuring the state is carried over to the next turn without errors through execution. In contrast, AnyTOD (Figure 5) predicts the entire state and action sequences from scratch at every turn, which is inefficient and error prone.

System actions In §2.1 we introduced hints as prompts generated by a deterministic dialogue policy through the execution of API calls or assignments. More broadly, the system policy can be designed to generate such prompts in a wider variety of contexts. As shown in Figure 7, PyTOD can be programmed to prompt the user to take action after a conversation pause (Figure 7, x16) or a negation (x28). If the user does not respond (as in turn 7), these actions are executed to continue the dialogue.

Successful API calls are marked by `perform` statements. These are always followed by a `say` command call to generate an agent utterance which informs the user the task has been successfully completed.

API call failures are also included in the prompt to enable PyTOD to reason about the dialogue state in such cases (Figure 7, x19). Depending on the API response, PyTOD may assist the user in recovering from failures by offering alternatives, represented as hint messages (x20).

NLG calls Unlike Figure 2, Figure 7 shows all NLG calls, illustrating how PyTOD can function as an end-to-end TOD agent. Since `say` commands ground agent utterances but do not provide relevant information for state tracking, they are omitted from the action parser prompt (the dialogue manager appropriately re-indexes the variables).

Handling complex utterances Figure 7 demonstrates how PyTOD handles compositional user utterances requiring nested function calls. Such utterances cannot represented by AnyTOD or other state-of-the-art TOD agents and DST models, which are limited to parsing slot-value pairs from user and agent utterances.

Context-dependent instructions Context-dependent instructions, formatted as developer: turns, appear in the prompt after an iteration (ie `next`) or confirmation instructions. In the former case (Figure 8a), they provide an itemized list of entity properties that the user may inquire about along with their natural language descriptions from the schema. A brief instruction precedes this list, prompting the language model to invoke the `say` routine to communicate the requested information to the user. In the latter case (Figure 8b), additional system policy instructions relevant to state tracking may be included. For instance, line 6-9 in Figure 8b illustrate how PyTOD can be guided to correctly parse API parameters following an API calling error.

B.2 Dialogue Manager

B.2.1 Schema supervisor

Figure 9 shows the SS prompt generation templates. A common input to these is `slot_schemas`, a list of dictionaries containing the names, descriptions, data type and possible values (for categorical slots) for the active service. These are formatted by developer-defined filters (e.g., `slot_definition_formatter`). The template for constraining the value of a categorical slot to one of the values listed in the schema (Figure 9c) is a special case of the template for constraining unknown slot names with categorical values (Figure 9b): only one slot definition corresponding

```
prompt:
user: weather on 11th of this month in london, england.
0 weather_1_get_weather(date = '11th of this month', city = 'london, england')
1 show(x0)
2 next(x0) # type: forecast
developer: the user may request specific information about the  'forecast' object (x2)
properties listed below. pass the relevant property or properties to 'say' to answer
(eg, say(x2.precipitation, x2.humidity)).
- precipitation: the possibility of rain or snow in percentage
- humidity: percentage humidity
- wind: wind speed in miles per hour
- temperature: temperature in fahrenheit
- date: date for the weather
agent: 71 degrees 15 percent chance
user: how humid though? windy?
3

target:
say(x2.humidity, x2.wind)
```

(a) Post-iteration context-dependent instructions.

```
...
agent: alright, so you'd like to send $1,470 from your checking account
    to the checking account belonging to yumi?
user: yes. how many days will that take?
9 confirm(x7)
developer: in the event of a 'banks_2_transfer_money' failure,
copy the values of the keywords below the agent last mentioned
to subsequent 'banks_2_transfer_money' calls:
- recipient_account_type
developer: unless a signal indicates a 'banks_2_transfer_money' calling
error, the properties
- transfer_time: number of days for the transfer to go through
may be communicated to the user upon their request by referencing
'x13' while calling 'say' (eg, say(x13.transfer_time)).
```

(b) Post-confirmation context-dependent instructions.

Figure 8: Sample context-dependent instructions.

to the one predicted by the AP is displayed and the none option is removed to ensure the output is one of the values enumerated in the schema.

B.3 Parser supervisor

Figure 10 shows the PS prompt generation template. The DM filters the transcript to extract user and agent relevant to the current task, ignoring irrelevant previous tasks which are not relevant for predicting the current slot value. These turns are processed by the `conversation_formatter` filter, which preprends the conversation role to the utterance. The schemata of the slots requested at the previous turned are passed to the template as `slot_list`. The `question_formatter` filter the formats the slot descriptions by lowercasing them and appending a question mark.

C Experimental details

C.1 Slot values normalisation

In SGD, the dialogue state updates when the user either explicitly provides a slot value or accepts a system-proposed value. Traditional TOD agents track the latter by extracting slot-value pairs from agent utterances. In contrast, as described in Section 2.1.2, PyTOD updates the state by executing `select` and `confirm` commands. These commands read relevant slot values from entities retrieved via database queries (for search-based interactions) or API responses (for transactional interactions). Since SGD system call annotations[15] are canonicalised, PyTOD must normalise open-valued parameters extracted from the dialogue history before making API calls and ensure that system-proposed slot values are de-normalised for evaluation.

In practice, normalisation is performed by looking up the surface form of a predicted value in a mapping that links surface forms to their canonical counterparts. This table is easily constructed from SGD semantic annotations, as illustrated in Figure 11. Instead of de-normalising slot values copied from entities or API responses, we equivalently extend the corpus annotations to include their canonical forms. This ensures that slots tracked via execution are directly comparable to the reference values used by the official evaluator.

```
Given the definitions, which keyword below best matches
 {{ predicted_argument_with_definition }}?

        {% for slot in slot_schemas -%}
        - {{ loop.index | int2alpha }}) {{ slot | slot_definition_formatter }}
        {%if loop.last -%}
        - {{ (loop.index + 1) | int2alpha }}) none:
            the definitions do not describe {{ predicted_argument }}
        {% endif -%}
        {%- endfor %}
Answer:
```

(a) Unknown slot name. Sample prompt in Figure 3a.

```
Here are some definitions:
{% for slot in slot_schemas %}
    - {{ slot | slot_definition_formatter }}
{%- endfor %}
{% set cnt = [0] %}
Given these, {{ predicted_keyword }} is a synonym of:
    {% for slot in slot_schemas -%}
    {% for value in slot.possible_values -%}
    {% if cnt.append(cnt.pop()  + 1) %}{% endif %}
    - {{ cnt[0] | int2alpha }}) {{ slot.name }} {{ "=" }} {{ value | lower }}
    {%- endfor -%}
    {%if loop.last %}
    - {{ (cnt[0] + 1) | int | int2alpha }}) \
    options do not describe {{ predicted_keyword }}
    {% endif -%}
    {%- endfor %}
Answer:
```

(b) Unknown slot name (closed value). Sample prompt in Figure 3b.

```
Here are some definitions:

    - {{ arg_def.name }}: {{ arg_def.description | lower }}

Given these, {{ predicted_keyword }} is a synonym of:
    {% for value in arg_def.possible_values %}
    - {{ loop.index | int2alpha }}) \
    {{ arg_def.name }} {{ "=" }} {{ value | lower }}
    {%- endfor %}

Answer:
```

(c) Unknown categorical slot value

```
Which sentence below paraphrases {{ memorised_argument_description_with_name }}?

    {% for slot in slot_schemas -%}
    - {{ loop.index | int2alpha }}) {{ slot | slot_definition_formatter }}
    {%if loop.last -%}
    - {{ (loop.index + 1) | int2alpha }}) \
        none of the sentences above paraphrases {{ memorised_argument_name }}
    {% endif -%}
    {%- endfor %}
Answer:
```

(d) Memorised slot name. Sample prompt in Figure 3c.

Figure 9: Schema supervisor prompt templates

[15] These include call parameters, entity properties, and API responses.

```
Q: Answer the following questions. Output "unanswerable" \
if the question cannot be answered given the conversation.

In the conversation:

{{source_turns | conversation_formatter }}

{%- for item in slot_list %}
{{ loop.index }}) {{ item | question_formatter }}
{%- endfor %}

Answer:
```

Figure 10: Parser supervisor template

C.2 Implementation details

Library versions We finetune D3ST and SDT-Seq using the `transformers` (Wolf et al., 2019) library and the key software dependencies in Table 6. With the exception of FlanT5 (Chung et al., 2024) D3ST, which was trained on two NVIDIA A100 GPUs (80GB), all models were trained on a single NVIDIA RTX 3090 GPU (24GB).

Library	Version
transformers	4.35.2
accelerate	0.24.1
torch	1.17
numpy	1.26.2

Table 6: Software dependencies used to train PyTOD.

D3ST replication We set all training parameters to match those reported by Zhao et al. (2022) and pre-process the data using their official script[16]. As the original work does not specify a model selection metric or evaluation frequency, we evaluate every $5,000$ steps and select the best checkpoint based on overall JGA on the development set. Training is terminated early if accuracy does not improve within $15,000$ steps (approximately 3 epochs).

Our results show an absolute 1.7% difference from the published JGA for the base model (Table 7, rows 1 & 2). We observe a $+1.0\%$ improvement on seen services, but a 2.6% drop on unseen services.

To rule out overfitting, we increase the evaluation frequency to 900 steps and select the model maximising the *unseen serivces* JGA, stopping the training after 1 epoch if there are no improvements. However, this leads to a slight performance drop (Table 7, rows 2 & 3). We finetune Flan-T5 (780M) with the best settings, achieving seen services performance on par with the published results but a 4.6% discrepancy on unseen services (rows 4 & 5).

[16]Available at https://bit.ly/4aKe9KL

```
1  "frames": [
2    {
3      "actions": [
4        {
5          "act": "INFORM",
6          "canonical_values": [
7            "11:30"
8          ],
9          "slot": "time",
10         "values": [
11           "half past 11 in the morning"
12         ]
13       }
14     ],
15     "service": "Restaurants_1",
16     "slots": [
17       {
18         "exclusive_end": 49,
19         "slot": "time",
20         "start": 22
21       }
22     ],
23     "state": {
24       "active_intent": "ReserveRestaurant"
25       "requested_slots": [],
26       "slot_values": {
27         "time": [
28           "half past 11 in the morning",
29           "11:30",
30         ],
31         ...
32       }
33     }
34   }
35 ]
```

Figure 11: Semantic frame for the utterance *I would like it to be half past 11 in the morning.* The action annotations (line 3 - 14) are processed to extend slot-value annotation with the corresponding canonical value (e.g., line 29). The ellipsis in line 31 marks slot-value pairs which were omitted for clarity.

SDT replication We set training hyperparameters to the values reported by Gupta et al. (2022) and use the data processing scripts from the official SDT code release[17]. Since the evaluation frequency and model selection metric are unspecified, we evaluate every 1600 steps, selecting the model with the highest development set overall JGA. We closely replicate the reported results (Table 8).

PyTOD implementation details We finetune PyTOD with the same software versions as our baselines and the parameters in Table 9 until the development set JGA for *unseen* services is max-

[17]Available at https://bit.ly/4aKe9KL

Size	Model	JGA	JGA Seen	JGA Unseen	Setting	#
220M	D3ST (Zhao et al., 2022)	72.9	92.5	66.4	-	1
	D3ST (Flan-T5, ours)	71.2	93.2	63.8 (65.0/64.8/61.7)	A	2
	D3ST (Flan-T5, ours)	70.7	92.9	63.3 (61.9/65.3/62.7)	B	3
780M	D3ST (Zhao et al., 2022)	80.0	93.8	75.4	-	4
	D3ST (Flan-T5, ours)	76.5	93.8	70.8 (69.9/71.6/70.9)	A	5

Table 7: D3ST replication results. Numbers in brackets show the metric values for each experiment run, three-runs averages are shown otherwise.

Size	Model	JGA	JGA Seen	JGA Unseen	#
220M	SDT-Seq (Gupta et al., 2022)	76.3	-	-	1
	SDT-Seq (Flan-T5, ours)	77.5	93.5	72.2	2
780M	SDT-Seq (Gupta et al., 2022)	83.3	-	-	3
	SDT-Seq (Flan-T5, ours)	82.7	94.1	78.9	4

Table 8: SDT-Seq replication results.The reported numbers are averaged over five runs, each using a distinct set of demonstrations to construct the fine-tuning prompts.

imised. The learning rate is constant, with no scaling. We follow the same protocol when finetuning `google/flan-t5-large`, except that we allocate a training budget of just one epoch.

Hyperparameter	Value
Pretrained model	`google/flan-t5-base`
Optimizer	Adafactor
Batch size	32
Learning rate	0.0001
Warmup steps	1500
Number of epochs	2
Evaluation frequency	1500 steps

Table 9: PyTOD training hyperparameters.

D `pytod-lib`

D.1 Simulation framework

D.1.1 Service APIs

Figure 12 shows a sample implementation of the `Buses_3` service from SGD. The service provides two APIs: (1) `FindBus`, a *search* or *query* API (Figure 12a) that enables users to query a bus schedule database using natural language, and (2) `BuyBusTicket`, a *transactional* API (Figure 12) that allows users to purchase a ticket for an itinerary proposed by the TOD agent based on `FindBus` search results or by specifying ticket details directly.

D.1.2 Dialogue policy

The SGD conversations are generated by sampling from a *policy graph* (Rastogi et al., 2019; Mosig et al., 2020a), which outlines the intended flow of a dialogue. Both search and transactional APIs require zero or more specific slots to function. To provide them, the system processes the initial user turn and takes actions to elicit missing slot values. Once all required slots are filled, search APIs can construct a valid database query, while transactional APIs execute an external service call (e.g., to a ticket booking service).

```python
@register_command(service="Buses_3")
class FindBus(SearchCommand):
    from_city: SearchCommandArgument[str] = SearchCommandArgument()
    to_city: SearchCommandArgument[str] = SearchCommandArgument()
    departure_date: SearchCommandArgument[str] = SearchCommandArgument()
    num_passengers: SearchCommandArgument[str] = SearchCommandArgument()
    category: SearchCommandArgument[str] = SearchCommandArgument()

    def __init__(self, dialogue_id: DialogueID):
        super().__init__(dialogue_id)
```

(a) Search API.

```python
@register_command(service="Buses_3")
class BuyBusTicket(ConfirmedCommand):
    from_city: ConfirmedCommandArgument[str] = ConfirmedCommandArgument()
    to_city: ConfirmedCommandArgument[str] = ConfirmedCommandArgument()
    departure_time: ConfirmedCommandArgument[str] = ConfirmedCommandArgument()
    departure_date: ConfirmedCommandArgument[str] = ConfirmedCommandArgument()
    num_passengers: ConfirmedCommandArgument[str] = ConfirmedCommandArgument()
    additional_luggage: ConfirmedCommandArgument[str] = ConfirmedCommandArgument()

    def __init__(self, dialogue_id: DialogueID):
        super().__init__(dialogue_id)
```

(b) Transactional API.

Figure 12: Implementation of the SGD Buses_3 service

Slot-filling is abstracted in the `SearchCommand` and `ConfirmedCommand` interfaces, which all concrete service implementations (e.g., `Buses_3`) inherit. Upon execution, the interfaces return system actions, such as `show`, `perform`, `Hint` and `Notification` (see Figure 1 in §1, Figure 2 in §2.1.2 and Figure 7 in App. B.1). For example, if a user says, *I need a bus from London to Manchester.*, executing `FindBus` returns a system action `Hint(request value: departure_date)`[18], which an agent can verbalise to ask for the missing constraints. Unlike a majority of examples in datasets such as STAR (Mosig et al., 2020a) and STARv2 (Zhao et al., 2023), where users provide constraints only when prompted, SGD conversations frequently include user-initiated constraint specification.

Complex policy The SGD policy models realistic, multi-turn interactions beyond simple slot filling.

For search APIs, it supports user goal changes, iteration through multiple results satisfying the same constraints, and cases where no results match the user's constraints.

Transactional API policy incorporates complex error recovery, where the system may suggest alter-

[18]The other arguments in Figure 12 are optional, so are not requested by the system.

native actions (e.g., an alternative bus time if the selected bus is fully booked). The user can accept or decline system-initiated changes or update their constraints to resolve the issue. Additionally, the system can initiate tasks (e.g., proposing a ticket purchase after retrieving a bus schedule), at which point the user may switch context to another task and later return to complete the system-initiated one. This policy extends beyond the standard slot-filling approach seen in MultiWOZ, where users typically accept system proposals without modifying their constraints or declining offers in response to API failures.

We encourage interested readers to explore the documentation of the `simulation` package in our pre-release code.

D.1.3 API behaviour

Feedback The `pytod` APIs define slots as class variables implementing the *descriptor protocol*[19], enabling advanced functionality. One such function is *execution error feedback*: setting an undefined attribute on an API returns a string describing the error instead of raising an exception. Another example is *type coercion*: descriptors cast slot values to the data types specified in the schema, and may be configured to provide natural language feedback if conversion fails.

Execution engine The execution engine converts the dialogue manager's output into a `python` object. Figure 13 illustrates how a program statement is interpreted as an API object implemented by a service (e.g., `FindBus` in Figure 12a). The execution process begins by parsing the statement into an abstract syntax tree (AST) (line 24). If the tree matches a function call signature (line 25), a conversion function (`string_to_py_cmd`) first extracts the function name and arguments (lines 6–10). If the function corresponds to an API implemented by the schema (line 14), the engine retrieves the command from a registry, instantiates the appropriate object (e.g., `FindBus`), and assigns slot values (lines 15–18). For user or system actions, the function name is returned (lines 20–22), and object instantiation is handled downstream. We invite interested readers to consult the documentation of the `execution` package in our pre-release code for further details.

[19] See https://bit.ly/4hLB4aU.

```python
def maybe_parse_api_call_or_action(
    program_statement: str
) -> Command | str | None:
    """Parse an action parser output to a python API or action name."""
    def string_to_py_cmd(ast_expr: ast.Expr) -> Command | str:
        match ast_expr:
            # matches a fcn call e.g.,
            # buses_3_find_buses(location="Cambridge")
            case ast.Call(ast.Name() as func, args, keywords):
                command_name = func.id  # "buses_3_find_buses"
                match command_name:
                    # if the command is not a user/system action
                    # it must be an API call
                    case tool if tool not in USR_SYS_ACTIONS:
                        # build FindBus command from the schema
                        command = _build_command(command_name)
                        # set slot values
                        _set_command_properties(command, args, keywords)
                        return command
                    case _:
                        # return the action name, parsed downstream
                        return command_name

    match ast.parse(program_statement).body:
        # matches function calls e.g. find_bus(), next(x1), etc
        case [ast.Expr(value=ast.Call() as expr)]:
            api_py_obj = string_to_py_cmd(expr)
            return api_py_obj
        case _:
            return
```

Figure 13: Sample execution engine code, showing how a program statement is interpreted as a `python` object.

D.2 Task-oriented agents beyond MultiWOZ

Endowing LLMs with tool-use capabilities has attracted widespread focus in the research community. While LLMs have facilitated creation of synthetic data to evaluate LLMs' ability to parse individual user commands into tool calls, extending this to generate high quality conversations following predefined policies remains an open challenge (Zhao et al., 2023; Stacey et al., 2024). As a result, while a plethora of corpora evaluating single-turn interactions exist[20], few provide an interactive evaluation setting for conversational use. ToolSandbox (Lu et al., 2024) is among the few notable exceptions, yet, as Lu et al. (2024) highlight, hallucinations remain a key limitation in their approach to policy-grounded conversation simulation.

In contrast, task-oriented dialogue corpora are collections of natural conversations following predefined policies. These collections are expert-curated and their dialogues are grounded in rich ontologies of user/system actions and diverse API calls. As we showed in this paper, actions and API calls can be naturally represented as tool calls.

By introducing an interactive environment for SGD, the largest and most complex TOD dataset, we provide a valuable resource to both the tool-use and TOD research communities. Compared to MultiWOZ (Budzianowski et al., 2018), a widely used benchmark in both dialogue and LLM research, SGD presents significantly greater challenges due to its complex policy (§D.1.2), richer ontology and task diversity (Table 10). These fac-

[20] See Qu et al. (2024) for a recent review.

Dataset	# Domains	# Intents	# Slots	# Dialogues	Avg. Turns
SGD	20	88	365	6684	20.44
MultiWOZ	7	11	35	2000	13.46

Table 10: Comparison of dataset statistics for SGD and MultiWOZ. Dialogue counts reported are for combined development and test sets whereas average turns is reported for the training split.

tors make it a more rigorous benchmark for studying conversational tool use and TOD agent generalization, aligning with recent efforts to develop scalable, policy-driven dialogue systems (Hudeček and Dusek, 2023).

E Extended Analysis and Discussion

E.1 Parser supervisor

As the PLM size of the AP increases, the cost of correcting semantic errors and omissions rises, prompting us to explore separate models for PS and AP.

Table 11 shows that reducing PS size from 780M to 220M results in a negligible performance drop for PyTOD (Large), regardless of the SS size (cf. rows 1 - 3 vs. 4 - 6). This holds even though the 780M PS outperforms the 220M PS by 1.9 exact match points as it adapts better to unknown questions[21]. However, the overall system performance is marginally worse compared to jointly training a single model for action parsing and parsing supervision, indicating a small benefit from multi-task learning.

PS Size	EM	EM Seen	EM Unseen	SS Size	JGA	JGA Seen	JGA Unseen
220M	91.5	96.7	90.3	220M	80.0 (-0.5%)	90.6	76.4
				770M	80.7 (-0.5%)	90.5	77.4
				3B	81.8 (-0.4%)	91.7	78.5
770M	93.4	96.5	92.8	220M	80.1 (-0.4%)	90.6	76.5
				780M	80.8 (-0.4%)	90.5	77.6
				3B	81.9 (-0.3%)	91.7	78.7

Table 11: Performance of PyTOD (Large) when the parser supervisor (PS) is implemented with a specialised Flan-T5 model. EM denotes exact match answer accuracy, evaluated on the SGD test set. Bracketed numbers represent absolute JGA deviation with respect to the corresponding PyTOD models where the AP and PS models are jointly trained (Table 3, ✓-marked rows).

E.2 Annotation errors

Table 12 presents sample errors identified in our analysis in §5.3, showing intent paraphase errors

[21]For comparison, the EMs for PyTOD (Large) and PyTOD (Base) in Table 1 are 92.5% and 91.0% ,respectively.

#	Utterance	Annotation	Explanation	Service
1	Today at 2 in the afternoon.	*pickup_time='2 in the afternoon'*	*start_date='today'* in utterance	RC_3
2	I'm in the mood for some music and would like to play some songs.	*intent=LookUpMusic*	The utterance semantics is better represented as *intent=PlayMedia*.	MUS_3
3	I need a train ticket with a fully refundable feature.	*intent=FindTrains class='Flexible'*	The utterance semantics is better represented as *intent=GetTrainTickets*.	TR_1
4	Okay, what about attractions there. I need Place of Worship, and something with no entry fee.	*intent=FindAtractions category="Place of Worship" free_entry=True good_for_kids=True*	No *good_for_kids=True* mention in utterance.	TRA_1

Table 12: Sample annotation errors identified during the error analysis. *RC_3=RentalCars_3*, *MUS_3=Music_3*, *TR_1=Trains_1*, *TRA_1=Travel_1*.

in #2&3. While our DM performs argument-based disambiguation to identify intent, *GetTrainTickets* and *FindTrains*, the two *Trains_1* intents, share all their arguments. Consequently, misparaphrased intent annotations prevent PyTOD from retrieving train schedules, leading to degraded DST performance. Analysis of 30 additional dialogues from this domain, we found that intent confusion cased state errors in 20 out of 50 cases.

Table 12 highlights some utterances contain slots mentions without corresponding user action annotations (#1) while others sometimes fail to paraphrase actions parametrised by boolean or categorical slots(#4). These issues were difficult to identify with the methods available to Rastogi et al. (2019). During PyTOD development we, however, identified 451 dialogues across the train, development and test releases[22] while experimenting with fault-tolerant execution. This suggests that programmable dialogue systems, while challenging to develop, can serve as valuable tools for dataset quality improvements.

E.3 Overcoming copy errors with PyTOD

§5.3 identified slot propagation from search queries to transactions (e.g., Figure 14) as a key PyTOD failure mode. These errors can be mitigated by training PyTOD to pass object references to follow-up tasks (e.g., predicting *music_3_play_media(entity=x5, device=patio)* instead of enumerating all slots to be copied). The state could then be robustly tracked by execution. Note PyTOD is fine-tuned to predict entity selection in the symbolic form shown in (x5, Figure 14)

[22]We include dialogue IDs in our code release.

```
user: i want to find a song now, and i know that there are some really good
songs in the album called we are not your kind, also i enjoy rock-and-roll songs
which are from the '19.
x1 music_3_lookup_music(album = 'we are not your kind', genre = 'rock-and-roll',
   year = 2019)
x2 show(x1)
x3 next(x1)
x4 say(x2)
agent: there is 1 song which you might like. how would you like listening to the
birth of the cruel by slipknot from the album called we are not your kind?
user: yes, that is the song i want.
x5 select(x3, from_results = x1)
x6 suggest(task = 'music_3_play_media')
agent: do you want this song to start playing now?
user: yes, i do. play the song on the patio speaker.
x7 music_3_play_media(track = 'the birth of the cruel', device = 'patio')
```

Figure 14: A *Music_3* copy error. The slot *album* is not copied to to the *music_3_play_media* call (x7).

and these symbolic representations are rendered in the dialogue history to provide relevant entity information (i.e., *song(track, optional=['album'])*). This facilitates robust slot propagation by reasoning over entity types, a topic future research will explore.

E.4 C-JGA breakdown

To support future comparisons, we provide a breakdown of C-JGA metrics for our models and replicated baselines across seen and unseen services. Upon public release, we plan to contribute the C-JGA implementation to the official SGD evaluation code.

Size	Model	C-JGA	JGA (Seen)	JGA (Unseen)	#
220M	D3ST (Flan-T5, ours)	62.2	86.0	54.3	1
	SDT-Seq (Flan-T5, ours)	68.7	86.6	62.8	2
780M	D3ST (Flan-T5, ours)	66.5	87.9	61.0	3
	SDT-Seq (Flan-T5, ours)	74.2	88.0	69.6	4
220M	PyTOD (Base)	72.7	87.3	67.8	5
780M	PyTOD (Large)	78.4	89.1	74.9	6

Table 13: Breakdown of C-JGA reported in Table 1 by seen/unseen service.