ASPERA: A Simulated Environment to Evaluate Planning for Complex Action Execution

Anonymous ACL submission

Abstract

This work evaluates the potential of large language models (LLMs) to power digital assistants capable of complex action execution. Such assistants rely on pre-trained programming knowledge to execute multi-step goals by composing objects and functions defined in assistant libraries into action execution programs. To achieve this, we develop ASPERA, a framework comprising an assistant library simulation and a human-assisted LLM data generation engine. Our engine allows developers to guide LLM generation of high-quality tasks consisting of complex user queries, simulation state and corresponding validation programs, tackling data availability and evaluation robustness challenges. Alongside the framework we release *Asper-Bench*, an evaluation dataset of 250 challenging tasks generated using ASPERA, which we use to show that program generation grounded in custom assistant libraries is a significant challenge to LLMs.

1 Introduction

Digital assistants such as Siri or Alexa provide a conversational interface for users to execute *simple actions* (e.g., *Set a timer for 5 minutes*). To achieve this, developers typically define APIs (intents) and collect data to train specialised parsing models responsible for translating user requests into machine-interpretable, domain-specific languages that can execute these APIs (Andreas et al., 2020; Cheng et al., 2020). Equivalently, action execution in this setting can be modelled as a function call to an intent API implemented by a target application (e.g., `alarm_set_timer(duration=5, unit='min')`). Function calling supports simple actions, but extension to execute *any* action on the device requires implementation of fine-grained intents and/or specialised parsing functions for an intractably large number of requests. To enable future digital assistants to execute *complex actions* (Figure 1), Jhamtani et al. (2024) propose generation of a program implemented with low-level *primitives* from assistant libraries[1]. We aim to evaluate the ability of LLMs to generate such programs when (1) the LLM has access to all the relevant information for generation, encoded in the assistant library documentation, or (2) the LLM selects the relevant primitives by exploring the entire assistant library as a first step prior to program generation. To this end, we address two challenges.

```python
1  def user_checks_team_member_birthday() -> list[str]:
2      """
3      Query: Hey [Voice Assistant], is it anyone's
4      birthday in my team today?
5      """
6      # Find user's team
7      user = get_current_user()
8      team = find_team_of(user)
9      today = now_().today()
10
11     names = []
12     for member in team:
13         # determine when colleague's birthday falls
14         profile = get_employee_profile(member)
15         this_year_birth_day = replace(
16             profile.birth_date, year=today.year
17         )
18         if this_year_birth_day == today:
19             names.append(member.name)
20     return names
```

Figure 1: Program executing the complex action *Is it anyone's birthday in my team today?* A possible query decomposition is marked by planning steps (lines 6, 12). The assistant must call 5 APIs (lines 7 - 9, 13 - 14), perform operations such as attribute access and passing values by attribute reference (line 15), in addition to iteration and flow control. Logical reasoning is required to deduce that the year of the birthday has to be updated to the current year to correctly execute the task.

1. Complex action evaluation instances comprising diverse, realistic queries annotated with programs requiring compositional use of multiple primitives are required for evaluation. Existing resources do not fully satisfy this requirement. SMCalFlow (Andreas et al., 2020) contains composi-

[1]The assistant library is a collection of functions and objects the assistant can use to compose plans which determine or change the user's device state. A *primitive* is any abstraction implemented in the library (e.g., a function or class).

tional queries but is annotated with a specialised domain-specific language (DSL) which hinders LLM performance (Bogin et al., 2024). DeCU (Jhamtani et al., 2024) is a dataset for evaluating plan generation for complex user queries but provides in-context examples (ICEs) demonstrating how to parse simple user requests to single-instruction programs in lieu of an assistant library; recent research has shown that task instructions and documentation, which DeCU lacks, can improve LLM performance on many other tasks (Lu et al., 2024; Srivastava et al., 2024; Hsieh et al., 2023). Styles et al. (2024) and Trivedi et al. (2024) develop simulated environments with comprehensively documented APIs, but limit action diversity by grounding queries in task templates.

2. Robust evaluation of complex action execution capability requires measuring task success, i.e. that the assistant actions satisfy the user goal. Jhamtani et al. (2024) note this to be an open problem, since functional correctness evaluation requires query-dependent databases and accounting for unwarranted side-effects[2]. Styles et al. (2024) tackle this by feeding databases to templated executable programs to annotate expected environment states. They propose strict database comparisons to estimate task success, and hence cannot evaluate queries with multiple outcomes and *information-seeking queries*[3]. Trivedi et al. (2024) address these limitations, but define environment states and evaluate task success via specialised programs implemented by domain experts for every task.

Contributions We propose ASPERA, a simulated environment supporting evaluation of agents capable of complex action execution with data generation capability. Given an assistant library simulation (§2.1), ASPERA enables a developer and an LLM to interact to generate diverse, high-quality complex user requests and programs which satisfy them. We show that robust task success estimation is possible for both synthesised and human-authored queries by prompting LLMs to generate programs which appropriately initialise the environment state and determine whether the executed action satisfies the user goal (§2.3.2 and 2.3.3). Using this system, we address the lack of complex actions execution data by generating *Asper-Bench*, a challenging collection of 250 tasks (§3). Evaluation on this dataset shows that (1) generating programs that satisfy complex action requests is a challenge for LLMs even when they are prompted with all the relevant information, despite their ability to generate plausible programs and (2) SOTA LLMs find it difficult to select all the primitives needed for composite tasks, adding a challenge to program generation (§5 and 6).

[2]This term describes an unintended action by the agent e.g., setting a meeting with the wrong attendees.

[3]Queries in which the assistant provides information to the user.

Module	Functions	Classes	Docs length (words)
time utils	22	11	986
work calendar	13	3	660
company directory	10	3	236
room booking	4	2	331
exceptions	-	1	209
Total	49	20	2,422

Table 1: Assistant library summary statistics. A module corresponds to a `.py` file. Docs length is the total length of the documentation strings defined inside the module. See Appendix B for details.

2 The ASPERA Framework

In ASPERA, a human developer initiates an interactive session in which an LLM is prompted to generate complex user requests grounded in a `python` library which can implement digital assistant use cases. In subsequent human-LLM interactions, two additional programs which enable success rate evaluation for arbitrary agents are generated. We now discuss how this works in practice.

2.1 The assistant library

ASPERA implements an assistant library which simulates a company in which employees in various teams (with a tree-based reporting structure) have meetings with one another under various conditions, managed by a room booking system. The library consists of 7 databases and 69 `python` primitives (Table 1). An extensive time utilities library, partially inspired by the SMCalFlow (Andreas et al., 2020), is implemented to test logical and arithmetic reasoning capabilities.

2.2 Components of an ASPERA task

A task generated by ASPERA has four elements: (1) the *user query*, a natural-language request for the assistant to execute an action (e.g., *Cancel my lunch with Jill*); (2) the *action execution program* (AEP), a program which satisfies the user request upon execution; (3) the *state initialisation program* (SIP), which uses the assistant library and simulation tools to set the environment state so that the query can be executed in `python` (i.e., establishing

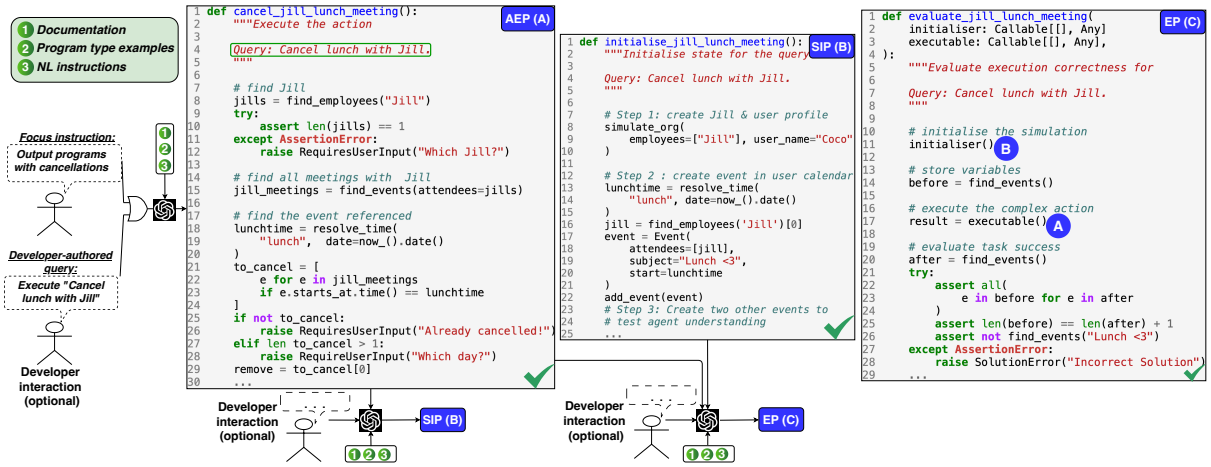


Figure 2: Sample ASPERA task, depicting action execution (A), state initialisation (B) and evaluation (C) programs. The task is generated in an interactive chat session (§2.4) which is initialised with AEP generation prompts (Appendix A.1 or A.2). To ground state initialisation, the chat history is extended with SIP generation prompt (Appendix A.3), which developers can customise with task-specific instructions. Finally, the chat history is extended with the EP generation prompt (Appendix A.4) which can also be customised by developers via instructions. At each step, the developer can execute and edit the generated programs to ensure data quality.

the existence of an employee named Jill and some meetings scheduled with her); (4) the *evaluation program* (EP) which runs the AEP in the initialised environment and determines its correctness (i.e., checks that the correct meeting has been deleted). Figure 2 depicts a simple ASPERA task.

2.3 ASPERA task generation

Figure 2 shows that the three programs which comprise a task are generated given: (1) assistant library documentation; (2) ICEs demonstrating the program format; and (3) natural language instructions. The instructions describe the assistant policy, environment assumptions and/or program structure information (depending on the program type to be generated).

2.3.1 Query and AEP generation

The user query can be authored by the human developer or synthesised by the LLM with the AEP (as part of the AEP docstring)[4]. By prompting the LLM with the documentation of the assistant library and with suitable examples, diverse and complex AEPs are generated. The complexity of the generated AEPs is characterised by: (1) number of primitives; (2) a variety of compositional patterns (Figure 2 AEP, lines 8 & 15, 18 - 20); (3) flow control and iteration (lines 21 - 24) and; (4) complex date-time reasoning (l. 18 - 20). Moreover, by prompting the LLM with exceptions, the AEPs model advanced assistant capabilities such as disambiguation (lines 9 - 12, 27 - 28) and determining if the requested action cannot be satisfied (lines 25 - 26). The AEP examples contain *planning steps*, that outline a possible decomposition of the task (lines 7, 14, 17) to encourage step-by-step thinking and to improve generation quality.

2.3.2 SIP generation

After AEP generation, the LLM is prompted[5] to generate an SIP which initialises the simulation state necessary to evaluate the query. In ASPERA, the SIP re-uses the primitives implemented for action execution (Figure 2 SIP, lines 13 - 22). This obviates the need for handcrafting databases, using templates to define the user query or prompting the LLM with database schemata. While statically defined databases model a single user's behaviour, ASPERA's dynamic database generation allows it to model multiple users. To simplify generation of complex environment states (e.g., an organisation reporting structure) the LLM can call ASPERA simulation tools (lines 8 - 10; see Appendix A.5).

2.3.3 EP generation

The final step is to generate an EP[6], which enables ASPERA to evaluate the functional correctness of an AEP to be evaluated. The EP takes as positional arguments the reference SIP and the AEP (Figure 2 EP, lines 2 - 4) and executes them in this order

[4] See query and AEP prompts in Appendix A.1 and A.2.

[5] See prompt listing in Appendix A.3.

[6] See prompt listing in Appendix A.4

Id	Query	Length (words)	Cyclomatic complexity	# primitives	Max. AST depth
1	Assistant, schedule lunch with my entire team tomorrow at noon.	12	1	7	6
2	Assistant, schedule lunch with a different team member each day next week at 12:30 PM.	17	3	8	10
3	Assistant, add a 1-hr strategy review with the CFO and the COO one week from today at 2:30.	23	5	13	9
4	Assistant, check my boss' calendar Wednesday to Friday next week, can they meet?	18	7	6	11
5	Assistant, I need to know which of Bill or Bob is busiest next week so I can allocate work.	21	7	7	14
6	Assistant, reorganise my diary on the fifth so that the important meetings come first.	16	9	10	11
7	Assistant, cancel the second meeting with Alice tomorrow if she declined.	13	8	5	10
8	Assistant, when in August when everyone from finance is off?	12	10	7	11
9	Assistant, set up a status update meeting with my manager every last Friday of the month at 2 PM till the end of the year. Skip his holidays.	33	10	16	10
10	Assistant, edit the attendee list for our fortnightly team planning on Wednesdays at 1 PM to remove Jack and Amy and add the newest sales hire.	28	13	11	10

Table 2: *Asper-Bench* sample queries (see §3)

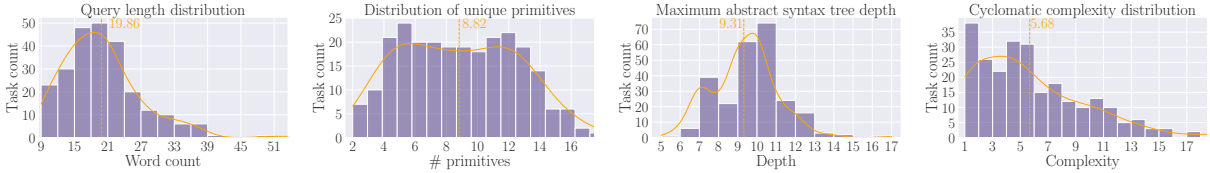


Figure 3: Distributions of key complexity measures in the *Asper-Bench* reference AEPs

(lines 11 & 17) to initialise the environment and execute the user action. Prior to action execution, one or more variables (line 14) store the initial state relevant to assessing side-effects and user goal completion. After AEP execution, the variables are compared with their expected values in assertion statements (lines 22 - 26). These verify the user goal was met without unexpected side effects.

The EPs thus implement goal-oriented agent evaluation (Budzianowski et al., 2018; Nekvinda and Dusek, 2021) even though the environment state is implicit in the queries and SIPs. Furthermore, the EPs generalise database comparison functions implemented in other environments (Lu et al., 2024; Styles et al., 2024) because they can evaluate information-seeking queries by comparing the AEP returned value against its expected value. Finally, evaluation of queries with multiple allowable outcomes[7] is supported in ASPERA by comparing captured state with a range of accepted values in assertion bodies.

2.4 Developer-LLM interaction in ASPERA

Figure 2 shows how AEP, SIP, and EP generation is sequential and moderated by a developer. The developer can seed the AEP generation with a focus instruction (top left) to provide guidance about attributes of tasks to be generated (e.g., action types, program length and complexity) or author the query and supervise AEP generation (bottom left).

After AEP generation, the chat history is automatically extended with the SIP generation prompt. The developer can optionally instruct the LLM to customise the environment state to be generated, define multiple SIPs or implement new simulation tools the LLM can use to write the SIPs. The interactive loop is repeated to enable EP generation. At any point, the developer can execute the programs in the simulated environment and edit them (or the queries) accordingly to ensure data quality.

3 The *Asper-Bench* Dataset

We generate an evaluation dataset of 250 tasks using GPT-4o[8], given five ICEs for each program type (§2.2). 71 tasks are information-seeking, while the remainder mutate one or more databases. We include both LLM- and human-authored queries. A single SIP and EP are generated for each query, except for conditional queries (Table 2, line 7) where state initialisation and evaluation are defined to test each AEP branch. Our annotations contain 9k, 13k and 17.5k lines across execution, initialisation and evaluation programs respectively.

Asper-Bench AEPs are diverse in their complexity (Figure 3). The distribution of maximum abstract syntax tree (AST) depth indicates AEPs satisfying the queries require compositional use of multiple primitives[9]; LLMs must interpret extensive documentation across multiple modules and demonstrate strong coding ability to generate AEPs which complete *Asper-Bench* tasks.

[7] Multiple outcomes are defined for *When is Bob free next Friday?* since both the upcoming Friday or Friday the following week are valid interpretations of the date mentioned.

[8] `gpt-4o-2024-05-13`.

[9] For comparison, the maximum AST depth of an AEP containing a call where all slot values are strings (e.g., `find_events(subject="Paper Review")` is 5.)

As further shown in Appendix C, the queries pose challenges ranging from parsing complex time expressions and date/time arithmetic (Table 2, rows 3, 8 - 10) to logical reasoning and interpretation of additional instructions (rows 3 - 5, see Appendix C.1). Hence, the diversity of the dataset arises from the complexity of the tasks rather than through, for example, paraphrasing. Representing such complex queries as programs requires iteration and flow-control patterns. This increases a program's *cyclomatic complexity* (CC), defined as the number of independent paths that can be traversed during execution (McCabe, 1976). Tasks with higher CC involve non-trivial operations to resolve people, events or dates (Table 3, rows 8, 10), complex rescheduling (row 6) and scheduling events subject to constraints (row 9). Lower CC tasks test fine-grained documentation understanding and programming ability (row 1); occasionally, these tasks require branching to follow instructions which provide relevant information about the environment that does not naturally fit in the documentation (row 3) or describe the assistant policy.[10]

Asper-Bench programs follow a policy for interrupting execution to interact with the user: the `RequiresUserInput` exception is raised if the entities mentioned by the user cannot be retrieved from the databases[11] or the task cannot be completed (e.g., a room is unavailable; see Appendix C.3).

4 ASPERA Evaluator

ASPERA provides an interface which enables arbitrary agents to execute AEPs and observe execution outcome. To support ongoing comparison of the baseline complex action execution capability of LLMs independent of the agent prompt, we provide two implementations of this interface.

1. Complete codebase knowledge (CCK) The agent prompt (Figure 17, Appendix D) contains the documentation for the entire assistant library (Table 1) alongside the five AEP example used to generate *Asper-Bench*. The prompt also includes instructions for: an events scheduling policy; information about environment constraints[12]; and the output format. For information-seeking queries, the type of the object to be returned to the caller is also included in the prompt.

2. Primitives selection (PS) The primitives are not known when the user invokes the assistant. Including the entire assistant library documentation in the prompt (as in the CCK prompt) may be impractical due to context window and latency limitations. In such a case, the assistant must inspect the library to determine which primitives are needed to execute the action requested by the user. To evaluate how well agents perform under these constraints, we provide a simple interface in which AEP generation is conditioned on primitives selected by the LLM prior to generation. This involves an iteration through an extended assistant library[13]. At each step, the agent is prompted with the documentation for an ASPERA module (viz Table 1) alongside the user request and is asked to issue `import` statements to select relevant primitives or `None` if the module is not relevant for executing the requested action (Figure 18a, Appendix D). Upon iteration completion, the selected primitives replace the full application library listings in the CCK prompt.

As opposed to the 5 ICEs in the CCK prompt, the AEP generation prompt for PS contains just one example demonstrating the solution format. Had the CCK examples been included, the success rate of agents with poor primitive selection recall would have been overestimated because the primitives used by the ICEs and their documentation would be listed despite not having been purposefully imported.

Metrics We report task success. A task is completed if the generated AEP executes without error and all assertions pass in all reference EPs.

5 *Asper-Bench* Evaluation

Complete assistant library knowledge (CCK Setting) AEP generation is challenging for both proprietary and open-source LLMs even when they can directly observe all the knowledge relevant for planning (Table 3). Despite performing well on standard code generation benchmarks (Chen et al. (2021), Austin et al. (2021a)), and their ability to consistently generate syntactically correct AEPs (Table 3, column 5), the most widely used general-purpose assistants successfully exe-

[10] For details, see Figure 6c in Appendix A.1.

[11] Given the complexity of our tasks, we always simulate these entities; we leave adversarial user behaviour robustness evaluation (e.g., the user deliberately requests updating an event that is not in the calendar) to future work.

[12] These include e.g. company information (e.g., *The leadership team is formed of a CEO, COO and CFO.*).

[13] The extension contains documentation for the `ai_assistant`, `contacts`, `files`, `messaging`, `navigation`, `user_device_settings` modules in addition to those reported in Table 1, to be implemented in a future release.

Model name	Checkpoint	Size	Task success (%)	Syntax err. (%)
o1	o1-preview-2024-09-12	-	80.13	-
o1-mini	o1-mini-2024-09-12	-	51.40	0.13
GPT-4o	gpt-4o-2024-05-13	-	45.33	-
GPT-4o-mini	gpt-4o-mini-2024-07-18	-	21.07	-
3.5-turbo	gpt-3.5-turbo-0125	-	10.80	1.20
1.5-pro	gemini-1.5-pro-002	-	33.73	0.40
1.5-flash	gemini-1.5-flash-002	-	27.87	0.40
1.0-pro	gemini-1.0-pro-002	-	12.67	0.53
Mistral L	Mistral-Large-Instruct-2407	123B	38.00	-
Qwen2.5	Qwen2.5-72B-Instruct	72B	28.80	-
Gemma2	gemma-2-27b-it	27B	14.40	0.4
CodeGemma	codegemma-7b-it	7B	2.40	6.0

Table 3: CCK *Asper-Bench* task completion rates (5-shot). Rates for proprietary models are average of 3 runs with different seeds. We use greedy decoding for all models models except o1 where the API only allows setting the temperature to 1.

Model name	Setting	# ICE	Micro F1	P	R	Task success (%)
o1	CCK	5	-	-	-	80.13
	CCK	1	-	-	-	72.80
	PS	1	0.63	0.60	0.67	28.40
GPT-4o	CCK	5	-	-	-	45.33
	CCK	1	-	-	-	36.53
	PS	1	0.56	0.56	0.55	11.46

Table 4: PS task success. Rows 1 and 4 are repeated from Table 3, # ICE denotes the number of AEP examples in the prompt. Precision and recall are computed with respect to the *Asper-Bench* reference AEPs.

cute only 45.33% (GPT-4o) and 33.73% (Gemini 1.5 Pro (Reid et al., 2024)) of actions. Task success correlates with model size (Table 3, r. 9-13). However, the improved task success of o1-mini compared to larger LLMs such as GPT-4o (+6.1%) and Gemini 1.5 Pro (+17.67%) suggests that both code generation proficiency and step-by-step reasoning prior to program generation may be key for implementing advanced digital assistants with LLMs.

Primitive selection (PS setting) Despite its AEP generation capability when conditioned on the documentation of the entire ASPERA library, o1 retrieves just 67% of the primitives relevant for AEP implementation and achieves a modest 28.4% task completion rate as a result (Table 4). Hence, while identifying which primitives are relevant for executing a given action is relatively simple for human developers, we find that SOTA LLMs have limited ability to perform in this setting.

6 Analysis and discussion

6.1 CCK error analysis

We begin with an in-depth analysis of programs generated by agents prompted with the documentation of the entire ASPERA library. A breakdown of the errors observed is presented in Figure 4. We

Statistic	Model name				
	GPT-3.5-turbo	GPT-4o-mini	GPT-4o	o1-mini	o1
Lines of code Δ to reference AEPs	-12.15	-7.3	-5.48	3.22	8.72
`RequiresUserInput` usages	52	93	170	360	291
Average planning steps (viz. Figure 1)	4.83	5.63	5.41	6.15	9.16
Helper functions count	0	2	11	29	65
Average cyclomatic complexity	2.92	3.82	4.44	5.95	6.80

Table 5: Key generated AEPs statistics

Model name	Programs debugged	Programs analysed	Errors labelled	Could recover (%)
GPT-4o	33	125	41	48.39
GPT-3.5-turbo	66	125	100	24.62

Table 6: Execution error analysis statistics.

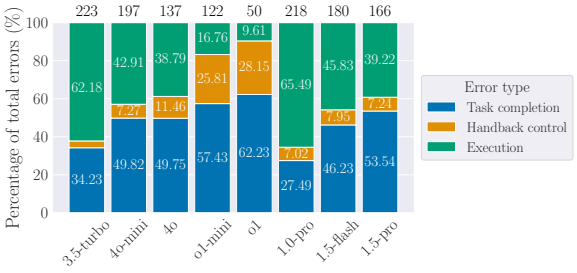


Figure 4: Assistant error types for OpenAI and Gemini model families. Top row displays total error counts.

make three key observations.

First, for both OpenAI and Gemini models, more capable[14] variants produce a larger proportion of *task completion errors*, in which programs execute successfully but fail an assertion in evaluation. Such an error indicates that the model can successfully use and combine primitives, but fails to understand some nuance in the user request and therefore takes the wrong action. Table 13 (Appendix E.2) shows concrete examples of this.

Second, less capable models incur relatively more *execution errors*, in which programs are syntactically correct but trigger a runtime exception. An in-depth error analysis of 141 such errors from GPT-3.5-turbo and GPT-4o[15] shows that both models have a tendency to hallucinate in situations where multi-step reasoning is required, generating shorter AEPs compared to the reference annotations (Table 5, row 1). Additionally, we find that execution errors often occur with task completion errors; in other words, the solution is incorrect even if the execution error is manually fixed (Table 6, column 5). While self-reflecting agents (Shinn et al., 2023) could achieve higher task success, our evaluation considers complex action execution in the single trial setting since, in practice, self-debugging iterations increase latency and trial execution might

[14] As ranked by performance on standard general, math, reasoning and code benchmarks.

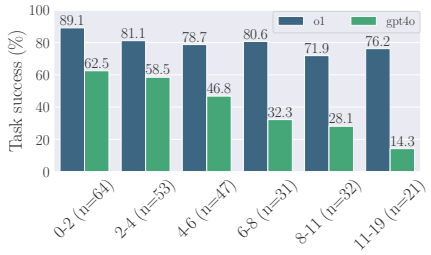
[15] Detailed in Appendix E.1.

Subset	CC	AST depth	o1(%)	GPT-4o (%)	Example
Simple	1.9	7.3	100	100	Table 2, row 1
Constrained scheduling	7.1	9.6	86.67	46.67	Table 2, row 9
Complex time expressions	5.4	9.2	63.33	20.00	Table 2, r. 4 & 10
Policy / instruction following	6.0	9.2	80.00	20.00	Table 2, r. 2 & 3
Advanced problem solving	9.2	10.6	56.67	26.67	Table 2, row 5

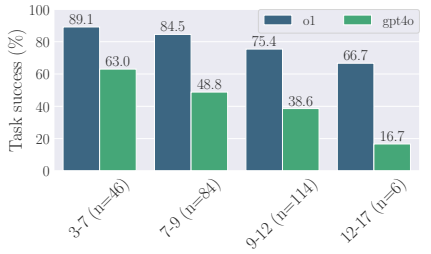
Table 7: Task success for query subsets. Each subset has 10 queries, see Appendix E.4 for complete listings.

have unintended consequences (e.g., some events are scheduled before the program execution fails).

Third, more capable models generate a greater proportion of *handback control errors*. These errors are linked to more frequent use of the `RequiresUserInput` exception (Table 5, row 2), used to handle cases in which the assistant cannot complete a task or cannot disambiguate between some entities at runtime. The errors occur when this exception triggers unexpectedly, indicating that the agent has made an incorrect assumption or misidentified an edge case. Such errors provide insight into the types of queries which challenge even SOTA models.[16]



(a) Cyclomatic complexity



(b) Maximum AST depth

Figure 5: Task success as a function of reference AEP complexity (n denotes bucket size)

6.2 Handling complexity

Asper-Bench requires models to perform various complex compositions of primitives and control flow sequences. Figure 5 shows that o1 can successfully complete a much larger proportion of tasks which require generating complex programs compared to GPT-4o. As seen in Table 5, o1 is more capable in this regard due to its ability to break down the task into fine-grained steps (row 3), make use of helper functions to encapsulate complex functionality (row 4) and to more effectively employ flow control and iteration (row 5).

To further demonstrate the challenges in *Asper-Bench*, we select 5 subsets of 10 queries which test different aspects of assistant understanding and reasoning capabilities. Table 7 (row 1) shows that both o1 and GPT-4o can equally handle simple problems (e.g., scheduling an event on a given date, or deleting events) but a large gap is observed in the completion rate of advanced tasks. Compared with o1, GPT-4o is significantly challenged by scheduling with constraints and resolving difficult relative time expressions (rows 2 & 3), which require flow control, primitive composition and arithmetic reasoning. The same is true of generating AEPs constrained by additional instructions in the prompt (row 4) and solving very challenging examples from the above categories (row 5).

6.3 Primitives selection

The primitives selection setting proved challenging for both models evaluated, as shown in Table 4.

The LLMs analysed show limited ability to reason about dependent primitives. Using the `work_calendar` module, for example, requires knowledge about properties of the `Event` primitive. We find this relation is not recognised during selection; o1 fails to import both the relevant `work_calendar` API and `Event` in 29 out of 67 occurrences of `find_events`, 16 out of 69 occurrences of `add_event` and 8 out of 19 occurrences of `delete_event`.

The ability of an LLM to use a primitive listed in the prompt can be weakly associated with selection performance for that same primitive. Consider the function `add_event`. In our baseline setting (CCK, 1-shot), o1 achieves 66% task success rate on the subset of queries whose reference AEP uses this primitive. In selection of `add_event`, however, o1 shows a comparatively poor recall of 0.41 and a F1 score[17] of 0.58. This suggests that selecting a complete set of fine-grained primitives to execute a complex user request is a challenging problem for these LLMs.

[16] For error examples, see Table 14, Appendix E.3.

[17] A detailed breakdown of retrieval metrics per primitive can be found in Appendix E.5.

7 Related Work

Task-oriented parsing Parsing natural language queries into DSL programs interpretable by execution engines (Zelle and Mooney, 1996; Liang et al., 2013; Berant et al., 2013; Gupta et al., 2018, *inter alia*) is challenging for program structures unseen in training (Yao and Koller, 2022). Bogin et al. (2024) and Jhamtani et al. (2024) show that representing targets as *programming languages* improves LLMs' few-shot semantic parsing ability; we build on this by employing program synthesis to collect complex, high-quality, task-oriented queries and to evaluate agents executing them.

Tool-augmented LLMs & LLM Agents An alternative is query synthesis at scale by prompting LLMs with documentation of sampled synthetic- (Tang et al., 2023) or real-world APIs (Xu et al., 2023; Song et al., 2023; Qin et al., 2024) and query examples. Because the relations between the sampled APIs are sparse, the resulting programs are linear sequences of often unrelated API calls. As such, tool-use corpora mostly evaluate LLMs' ability to parse API call sequences rather than complex reasoning with multiple tools. By grounding queries in a library with primitives sharing type relations, we generate challenging tasks that require multi-step, arithmetic and logical reasoning, building on work by Shen et al. (2023), who ground queries in handcrafted task relation graphs.

Synthetic data generation at scale comes with both quality (Iskander et al., 2024) and evaluation (Guo et al., 2024) challenges. To tackle the former, human-authoring and manual curation have been increasingly employed (Huang et al., 2024; Jhamtani et al., 2024; Trivedi et al., 2024; Styles et al., 2024; Yan et al., 2024). Instead, we propose an interactive data generation engine to ensure data quality and reduce human cost. Like Styles et al. (2024) and Trivedi et al. (2024) we tackle evaluation challenges by executing agent actions in a simulated environment and determining whether they satisfy the user goal. While both Styles et al. (2024) and Trivedi et al. (2024) template user queries and resort to program templates (Styles et al., 2024) or high-fidelity task simulators (Trivedi et al., 2024) to annotate environment state, ASPERA does not constrain the format of the query or of the program grounding it. Like Trivedi et al. (2024) we generalise the strict database comparisons of Styles et al. (2024), but generate the evaluation programs in LLM interactions as opposed to manually implementing them for every task.

Code generation LLM ability is measured by benchmarks (Chen et al., 2021; Austin et al., 2021b; Hendrycks et al., 2021) which test algorithmic ability via generation of self-contained functions with contextual dependencies limited to standard libraries. To address this, other resources encompass narrow-domain dependencies on external data-science libraries (Lai et al., 2023; Wang et al., 2023) or a broader set of domains (Zhuo et al., 2024). ASPERA focuses on program generation with project-runnable dependencies (Yu et al., 2024) of custom primitives in the assistant library, which is very challenging but receives limited coverage in existing resources (Siddiq et al., 2024). Moreover, ASPERA tasks represent high-level user goals requiring the assistant to reason about primitive relevance, while the aforementioned benchmarks test program generation given precise function specifications and knowledge about external libraries acquired during pre-training. Evaluation robustness is guaranteed by execution of human-authored tests for all the above benchmarks except Zhuo et al. (2024) who, like our work, use human-LLM interaction to generate data and robustly evalute general software task competence.

8 Conclusion

This work evaluated the ability of LLMs to parse complex natural language queries into executable programs that involve non-trivial primitive composition and flow control. We have addressed key limitations in existing work regarding dataset availability and evaluation by devising an environment where LLMs and human developers interact to collect evaluation data and code for environment state initialisation and execution outcome verification. We found that generating programs which satisfy intricate user queries grounded in custom assistant libraries is challenging for a wide range of SOTA LLMs which are otherwise proficient at code generation. Our initial results also showed that, while SOTA LLMs can compose primitives to execute difficult tasks, they are limited in their understanding of whether a given primitive is needed given the query alone, which is of concern to practical digital assistants. Hence, *Asper-Bench* and the ASPERA framework enable future study of action execution in the challenging setting where the primitives are not known to the agent and must be retrieved or discovered via environment interaction.

9 Limitations

Interactive code generation Humans write code in an interactive manner (Yang et al., 2023), occasionally relying on execution feedback to correct errors, resolve ambiguities and decompose tasks iteratively. The majority of existing code generation benchmarks, including the current work, consider a non-interactive instruction-to-code sequence transduction process which has the potential for error propagation and a disconnect between the generated code and its execution environment. While the ASPERA environment supports interactive code generation grounded in environment feedback and observations, we have focused on evaluating LLMs' fine-grained understanding and ability to compositionally use multiple primitives and curated the tasks such that that they are solvable without interaction. In doing so, we have increased the difficulty of certain types of tasks (e.g., scheduling subject to constraints, tasks involving re-scheduling and diary re-organisation). Future work will focus on comparing the performance of interactive and non-interactive agents on *Asper-Bench*.

Scenario-based evaluation We have designed ASPERA such that each task can have multiple SIPs and corresponding EPs to support creating contrast sets (Gardner et al., 2020) for each task and comprehensively evaluate that the agent actions satisfy the user goal regardless of the initial state. However, unlike in domains such as customer resource management (Styles et al., 2024) or online ordering (Trivedi et al., 2024) where the user may not know the state of the environment, we assume that the user has complete knowledge of the state of their calendar. Consequently, scenario-based evaluation is very limited in *Asper-Bench* and concerns only queries involving the calendars of other actors in the environment (e.g., other employees) or the room booking system. Moreover, we do not generate states where entities are ambiguous (e.g., two employees share the same surname and the user attempts to schedule a meeting with one of them without further identifying them). Future work could thus extend the SIP generation to support scenario-based evaluation.

Dataset size *Asper-Bench* is comparable in size to other popular code generation benchmarks such as HumanEval (Chen et al., 2021), NumpyEval (Zan et al., 2022b) , PandasEval (Zan et al., 2022b) and TorchDataEval (Zan et al., 2022a), but likely not sufficiently large for finetuning LLMs for digital assistant applications. Future work could focus on scaling the size of our data using the ASPERA data generation engine or by LLM-assisted paraphrasing of existing queries and refactoring of SIPs and EPs, similar to Zhuo et al. (2024). This would enable future work to study robustness of finetuned digital assistant models under non-trivial, semantics preserving transformations of the assistant library (e.g., refactoring).

Limited domain coverage The ASPERA assistant library supports parsing of complex time expressions and a simple simulation of a corporate calendar. Furthermore, the assistant library provides documentation for 6 domains (see §4, footnote 12). With more time investment, these domains could be simulated, along with any additional simulation and evaluation tools necessary to generate the environment state. The expansion could focus on evaluating requests which span multiple applications (e.g., *How long will it take me to drive to my next meeting this afternoon?*) which are not supported in the current release.

We note that, while the simulation and the current set of evaluation and simulation tools were developed offline by one of the authors with GPT-4o assistance, future releases could explore the use of LLMs for assisting the developer with auxiliary tool implementation during the ASPERA interactive session. We anticipate that the human effort required to scale to new domains depends on the LLMs available for data generation, the complexity of the domain considered and the complexity of the scenarios developers wish to simulate.

Multi-turn interactions In keeping with recent works focused on multiple tool use and LLM agents, our work considers a user which issues a complex request in a single-turn interaction. In practice, it is desirable that the digital assistant can handle complex requests at any point in a conversation. Moreover, multi-turn interaction is necessary when the assistant cannot perform entity disambiguation or has failed to solve the task. Future work could exploit the error handling sequences in the reference *Asper-Bench* AEPs to generate dialogues where complex action execution requires user interaction, similar to recent work by Lu et al. (2024).

References

Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan

DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H. Lin, Ilya Lintsbakh, Andy McGovern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Andrew Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. 2020. Task-oriented dialogue as dataflow synthesis. *Trans. Assoc. Comput. Linguistics*, 8:556–571.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021a. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021b. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1533–1544.

Ben Bogin, Shivanshu Gupta, Peter Clark, and Ashish Sabharwal. 2024. Leveraging code to improve in-context learning for semantic parsing. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pages 4971–5012. Association for Computational Linguistics.

Pawel Budzianowski, Tsung-Hsien Wen, Bo-Hsiang Tseng, Iñigo Casanueva, Stefan Ultes, Osman Ramadan, and Milica Gasic. 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, Brussels, Belgium, October 31 - November 4, 2018*, pages 5016–5026. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Jianpeng Cheng, Devang Agrawal, Héctor Martínez Alonso, Shruti Bhargava, Joris Driesen, Federico Flego, Dain Kaplan, Dimitri Kartsaklis, Lin Li, Dhivya Piraviperumal, Jason D. Williams, Hong Yu, Diarmuid Ó Séaghdha, and Anders Johannsen. 2020. Conversational semantic parsing for dialog state tracking. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 8107–8117. Association for Computational Linguistics.

Matt Gardner, Yoav Artzi, Victoria Basmova, Jonathan Berant, Ben Bogin, Sihao Chen, Pradeep Dasigi, Dheeru Dua, Yanai Elazar, Ananth Gottumukkala, Nitish Gupta, Hannaneh Hajishirzi, Gabriel Ilharco, Daniel Khashabi, Kevin Lin, Jiangming Liu, Nelson F. Liu, Phoebe Mulcaire, Qiang Ning, Sameer Singh, Noah A. Smith, Sanjay Subramanian, Reut Tsarfaty, Eric Wallace, Ally Zhang, and Ben Zhou. 2020. Evaluating models' local decision boundaries via contrast sets. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1307–1323. Association for Computational Linguistics.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 11143–11156. Association for Computational Linguistics.

Sonal Gupta, Rushin Shah, Mrinal Mohit, Anuj Kumar, and Mike Lewis. 2018. Semantic parsing for task oriented dialog using hierarchical representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2787–2792, Brussels, Belgium. Association for Computational Linguistics.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models. *CoRR*, abs/2308.00675.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. 2024. Metatool benchmark for large language models: Deciding whether to use tools and which to use. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Shadi Iskander, Sofia Tolmach, Ori Shapira, Nachshon Cohen, and Zohar Karnin. 2024. Quality matters:

Evaluating synthetic data for tool-using llms. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 4958–4976. Association for Computational Linguistics.

Harsh Jhamtani, Hao Fang, Patrick Xia, Eran Levy, Jacob Andreas, and Ben Van Durme. 2024. Natural language decomposition and interpretation of complex utterances.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Percy Liang, Michael I Jordan, and Dan Klein. 2013. Learning dependency-based compositional semantics. *Computational Linguistics*, 39(2):389–446.

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.

Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.

Tomás Nekvinda and Ondrej Dusek. 2021. Shades of bleu, flavours of success: The case of multiwoz. *CoRR*, abs/2106.05555.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew M. Dai, Katie Millican, Ethan Dyer, Mia Glaese, Thibault Sottiaux, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, James Molloy, Jilin Chen, Michael Isard, Paul Barham, Tom Hennigan, Ross McIlroy, Melvin Johnson, Johan Schalkwyk, Eli Collins, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Clemens Meyer, Gregory Thornton, Zhen Yang, Henryk Michalewski, Zaheer Abbas, Nathan Schucher, Ankesh Anand, Richard Ives, James Keeling, Karel Lenc, Salem Haykal, Siamak Shakeri, Pranav Shyam, Aakanksha Chowdhery, Roman Ring, Stephen Spencer, Eren Sezener, and et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *CoRR*, abs/2403.05530.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. 2023. Taskbench: Benchmarking large language models for task automation. *CoRR*, abs/2311.18760.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. 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*.

Mohammed Latif Siddiq, Simantika Dristi, Joy Saha, and Joanna C. S. Santos. 2024. The fault in our stars: Quality assessment of code generation benchmarks. *CoRR*, abs/2404.10155.

Yifan Song, Weimin Xiong, Dawei Zhu, Cheng Li, Ke Wang, Ye Tian, and Sujian Li. 2023. Restgpt: Connecting large language models with real-world applications via restful apis. *CoRR*, abs/2306.06624.

Pragya Srivastava, Satvik Golechha, Amit Deshpande, and Amit Sharma. 2024. NICE: to optimize in-context examples or not? 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 5494–5510. Association for Computational Linguistics.

Olly Styles, Sam Miller, Patricio Cerda-Mardini, Tanaya Guha, Victor Sanchez, and Bertie Vidgen. 2024. Workbench: a benchmark dataset for agents in a realistic workplace setting. *CoRR*, abs/2405.00823.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *CoRR*, abs/2306.05301.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. 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 16022–16076. Association for Computational Linguistics.

Boshi Wang, Hao Fang, Jason Eisner, Benjamin Van Durme, and Yu Su. 2024. Llms in the imaginarium: Tool learning through simulated trial and error. 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 10583–10604. Association for Computational Linguistics.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 1271–1290. Association for Computational Linguistics.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. 2023. On the tool manipulation capability of open-source large language models. *CoRR*, abs/2305.16504.

Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Berkeley function calling leaderboard. https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. 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*.

Yuekun Yao and Alexander Koller. 2022. Structural generalization is hard for sequence-to-sequence models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5048–5062, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 37:1–37:12. ACM.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022a. When language model meets private library. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 277–288. Association for Computational Linguistics.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022b. CERT: continual pre-training on sketches for library-oriented code generation. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 2369–2375. ijcai.org.

John M Zelle and Raymond J Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pages 1050–1055.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *CoRR*, abs/2406.15877.

A ASPERA dataset generation prompts

A.1 Joint query and AEP generation

```
My team needs your help with generation a wide variety of complex
programs that can be implemented with our application backend.
We care to generate only programs that would be generated by our
large language model when interacting with our application via a
voice interface.

Here is our application code.

```python
{{ code }}
```

Here are some examples of high quality programs that we wrote to
help you understand the task.

```python
{{ query_solution_examples }}
```

Guidelines:

1. Please limit yourself to generating programs involving complex
combinations of the members of our codebase. It is not helpful to
assume scenarios that our application cannot implement or assume
unknown details about methods implementations - focus on the
interfaces and read our documentation carefully.

2. Diversity is key. Focus on user requests that can be parsed
to a fairly complex program implemented with the codebase above.
Just put yourself in the shoes of the user wanting to get a lot
done with our application. Some ways to achieve diversity may be:
    - imagine scenarios using for loops
    - imagine scenarios based on user conditions
    - imagine scenarios requiring filtering operations
    - imagine many scenarios where multiple dataclasses and their
      methods are required to support a complex user goal
    - scenarios imagined should always be compositional (ie always
      have diverse combinations of object attributes and methods
      operating on them)

3. To reiterate, diversity (2) should not come at the expense of
imagining scenarios our codebase cannot support (1). We will
discuss how to improve our codebase in the future.

Program structure guidelines
The examples above follow the
{{ guidelines.generation_labelling | length }} structure guidelines
listed below. Do the same, clearly stating when you follow them in
your comments, as demonstrated above.
{% for instruction in guidelines.generation_labelling %}
{{ loop.index }}. {{ instruction }}
{%- endfor %}
```

(a) System turn. In the above the field `code` is replaced with the documentation of the assistant library and `query_solution_examples` is replaced with 5 AEP examples. See Figure 6c for guidelines definition.

```
You have done a stellar job generating some brilliant programs and
user queries already. To remind you of work you completed and keep
things brief, we only show the queries extracted from the docstrings
of programs you generated:
{% for q in queries %}
{{ loop.index }}. {{ q }}
{%- endfor %}

Now we have to generate more programs representing complex user
utterances. Crucially, these should represent a complex set of
new user queries, where the user tries to complete different tasks
from the ones you generated above. *Do not merely paraphrase the
queries you already generated* when synthesizing programs - think
of new and original complex user tasks that our application backend
supports.
{% if focus -%}
{{ focus }}

{% endif -%}

Let us generate {{ n_programs }} programs.

```python"""
```

(b) User turn. To encourage diversity, we optionally. include the history of the queries generated in the prompt, similar to Wang et al. (2024). If `n_programs` is set to values greater than 1, multiple programs are generated. The `focus` field can be changed after each round of interaction, to encourage diversity of generated queries and programs.

```
- Employee names are generally assumed unique, so you may use
  find_employee(name)[0] for resolving a name to an Employee
  object. Use this sparingly; even though there may be multiple
  employees with the same name, the user query might give additional
  information which resolves the ambiguity (eg specify the meeting
  time). If you decided to make this assumption add a 'by structure
  guideline #1' comment.
- Work meetings can start after 9:06 AM and should end before 5:10
  PM. They don't happen at the weekend unless the user explicitly
  mentions so.
- Type annotate the return for programs which have a return type
  which is not None.
- Do not call functions with default optional values.
```

(c) Guidelines used to populate `{{instruction}}` in the bottom loop of (a). The first guideline enforces a unique entity name environment constraint, which grounds 0-indexing `find_employee` results. We make this design decision to decrease task difficulty for our initial release, but note the LLM is instructed to mark this assumption with `# by structure guideline 1` to support future LLM-based annotations of AEPs which handle disambiguation. The second guideline encodes a simple events scheduling policy to be followed when explicit constraints are not provided by the user and when rescheduling events. The third guideline prompts for return type annotation for information-seeking queries and the final guideline encourages concise coding.

Figure 6: Prompt templates for joint program and task generation (Section 2.3.1)
```

A.2 AEP generation given human-authored request

```
You are an expert programmer working with my team which is specialising
in developing AI assistants. Your current task is to translate a series
of complex user requests into executable `python` programs using our
application backend below:

```python
{{ code }}
```

Here are some examples your colleagues shared with you to help you
generate your response in a style that is compatible with our
infrastructure:

```python
{{ query_solution_examples }}
```

{% if guidelines.generation_labelling -%}
Program structure guidelines
The examples above follow the
{{ guidelines.generation_labelling | length }} structure guidelines
listed below. Do the same, clearly stating when you follow them in
your comments, as
demonstrated above.
{% for instruction in guidelines.generation_labelling %}
{{ loop.index }}. {{ instruction }}
{%- endfor %}
{%- else %}
{%- endif %}
```

(a) System turn. The `code` and `query_solution_examples` fields are populated with the assistant library documentation and 5 AEP examples like in the joint AEP and query generation prompt depicted in Figure 6

```
Now it's your turn. Please translate the queries below into `python`
programs using the examples above to guide your response format.
The response should be inside a Python markdown block.
{% for q in queries %}
{{ loop.index }}. {{ q }}
{%- endfor %}

```python"""
```

(b) User turn. The framework supports AEP generation for query batches.

Figure 7: Prompt template used for AEP generation given a human-authored request (Section 2.3.1)
```

A.3 SIP generation

```
For testing purposes, we need to generate the underlying runtime  state of the user device. Your task is to carefully analyse
`{{ plan_name }}` along with the application code above and assist our testing team in setting up the runtime environment such
that `{{ plan_name }}` can be executed and its outputs verified. To do so, you will need to generate a `python` function named `{{ setup_function_name }}`.

We have implemented additional tooling you may find helpful for completing this task:

```python
{{ setup_code }}
```

You may use additional knowledge and create your own functions if needed - custom functions should be defined inside the
`{{ setup_function_name }}` function. Note how we import modules in the standard python library locally inside the
`{{ setup_function_name }}` and how our application code does not need to be imported (we automatically do so when we run the code).

Here are some comprehensive examples your testing team colleagues shared to help you generate a high quality program that sets up
the runtime environment correctly.

```python
{{ runtime_setup_examples }}
```

{% if guidelines.runtime_setup -%}
Runtime environment setup guidelines
The examples above follow the {{ guidelines.runtime_setup | length }} setup guidelines listed below. Do the same, clearly stating when
you follow them in your comments, as demonstrated above.
{% for instruction in guidelines.runtime_setup %}
{{ loop.index }}. {{ instruction }}
{%- endfor %}
{%- else %}
{%- endif %}

Let's now write `{{ setup_function_name }}`, our developers wrote some TODOs to get you started.

```python
def setup_env_{{plan_name}}():
    """Simulate the environment for the query:

    {{ query }}

    Note this means to create any persons, contacts, emails, events and everything that should exist
    in the user's virtual context when they make the query. You **should not** create new entities that are
    implied in the user request that the assistant has created in the `{{plan_name}}` function.
    """
'''
    {{ TODOs }}
```

(a) User turn for SIP generation. This turn is added to the chat history which contains the AEP generation system and user turns and assistant turn with the generated AEPs. `plan_name` is the name of the AEP function for which the state is to be initialised and the `setup_function_name` is the name of the SIP to be generated. `setup_code` is replacted by the documentation for additional tools the LLM can call to simulate complex environment state. One example is `simulate_org` in Figure 2 (program B, l. 9 - 11) which allows the LLM to simulate an organisation with a complex reporting structure by parametrising the simulation. The `runtime_setup_examples` field shows 5 SIP examples, which initialise the state for the 5 AEP examples in the chat history. Guidelines, shown in Figure 8b, state simulation assumptions. The LLM is prompted to mark these assumptions in comments to enable LLM-assisted refactoring of the SIPs. The `query` field is replaced by the user query. The `TODOs` fields marks instruction the developer may optionally specify. These are formatted on separate lines following `#TODO:` tags.

- Dates should be grounded using the tools in the time_utils library. When doing so, add a 'setup guideline #1' comment.
- Work meetings can start after 9:06 AM and should end before 5:10 PM. When doing so, add a 'setup guideline #2' comment.
- Events assumed to occur in the future should start after the date and time specified by time_utils.now_(), whereas events in the past should finish before time_utils.now_(). When doing so, add a 'setup guideline #3' comment.
- Employee names are assumed unique, so you may use find_employee(name)[0] for resolving a name to an Employee object. When doing so, add a 'setup guideline #4' comment.
- Ensure you follow all the TODOs with appropriate steps, but don't be afraid to do additional steps if you think it necessary - our developers may not write detailed enough TODOs.

(b) Guidelines used to populate `{{instruction}}` in the bottom loop of (a).

Figure 8: Prompt template used for runtime setup program generation (Figure 2, B).
```

A.4 EP generation

```
We need some test code to check that `{{ plan_name }}` executes correctly on the user device. After a careful analysis of `{{ plan_name }}` and
`{{ setup_function_name }}` (defined below), your task is to write a function `{{ test_function_name }}` to do so.

We have implemented additional tooling you may find helpful for completing this task:

```python
{{ setup_code }}
```

```python
{{ testing_code }}
```

You may use additional knowledge and create your own functions if needed - custom functions should be defined inside the `{{ test_function_name }}`
function. Note how we import modules in the standard python library locally inside the s`{{ test_function_name }}` and how our application code does not
need to be imported (we automatically do so when we run the code).

Here are some comprehensive examples your testing team colleagues wrote:

```python
{{ evaluation_examples }}
```
{% if guidelines.evaluation -%}
Testing guidelines
The examples above follow the {{ guidelines.evaluation | length }} setup guidelines listed below. Do the same, clearly stating when you
follow them in your comments, as demonstrated above.
{% for instruction in guidelines.evaluation %}
{{ loop.index }}. {{ instruction }}
{%- endfor %}
{%- else %}
{%- endif %}

 Here is the code that sets up the runtime environment for `{{ plan_name }}` execution:

 ```python
 {{ runtime_setup_program }}
 ```

Write `{{ test_function_name }}`:

```python
def evaluate_{plan_name}(
    query: str, executable: Callable[[], Any], setup_function: Callable[[], Any]
):
    """Validate that `executable` program for the query

    {{ query }}

    has the expected effect on the runtime environment.

    Parameters
    ----------
    query
        The query to validate
        executable
        The query execution function, `{plan_name}`
    setup_function
        `{setup_function_name}` function.
    """
'''
```

(a) User turn template for EP generation. This turn is added to the chat history, which contains at this point the user and system turns for AEP and SIP generation. `plan_name`, `setup_function_name` and `test_function_name` are formatted with the AEP, SIP and EP function names, respectively. `setup_code` is defined in Figure 8 and `testing_code` is replaced by documentation of other tools the LLM can use to verify AEP correctness (see Appendix A.5). The `evaluation_examples` field is replaced by 5 EP examples, which demonstrated how to evaluate the correctness of the AEPs in the interaction history given the SIPs examples. Guidelines, shown in Figure 9b, provide relevant assumptions for writing correct and concise test code (Appendix A.5). The LLM is prompted to mark these assumptions in comments to enable LLM-assisted refactoring of the EPs. The `runtime_setup_program` is the SIP, and `test_function_name` is name of the EP to be generated.

- fields of type list[Employee] of events returned by find_events are sorted alphabetically according to the name attribute. Sort attendees lists you create accordingly. When doing so, add a 'testing guideline #1' comment"
- For queries that have a return type, consider a range of possible alternative return types that could have been returned instead by the executable and check the result correctness in those cases too. Add a '#testing guideline #2' comment in this case.
- When checking events requested by the user were created, never test equality of the 'subject' attribute because variations in the meeting name can affect test robustness.
- When add_event is called without an ends_at parameter, a default duration of 16 minutes is assumed when writing the event to the underlying database. Check that the events for which end time or duration is not specified satisfy this constraint.
- SolutionError message is always 'Incorrect Solution'.
- Where possible, use the information in the runtime environment setup function below to simplify testing code.

(b) Guidelines.

Figure 9: Prompt template used for evaluation program generation (Figure 2, C).
```

A.5 Auxiliary ASPERA Tools

ASPERA defines auxiliary tools designed to aid SIP and EP generation (Table 8). These can be implemented by the developer interactively[18] or (before task generation begins).

Simulation Tools		
Module	**Tool**	**Functionality**
work_calendar	simulate_user_calendar	Adds a set of LLM-generated events to the user's calendar.
	simulate_employee_calendar	Adds a set of LLM-generated events to the calendar of a given employee.
company_directory	simulate_org_structure	Build an organisation structure given, employee names, team membership, user name and user role. Reporting relationships and employee profiles are simulated by ASPERA.
	simulate_vacation_schedule	Simulate the vacation schedule of a given employee.
	UserRole	Enum listing key company roles such as CEO and COO.
room_booking	simulate_conference_room	Add a conference room to the conference room database.
Evaluation Tools		
Module	**Tool**	**Functionality**
time_utils	repetition_schedule	Create a recurrence schedule for a meeting or reminder.
work_calendar	assert_user_calendar_shared	Check that a calendar has been shared between a list of employees.

Table 8: ASPERA auxiliary tools

Simulation tools Simulation tools are included in SIP generation prompts to allow the LLM to create entities stored in environment databases. These tools differ in their implementation complexity. Some tools (e.g., `simulate_user_calendar`) simply write LLM-defined entities to the environment databases whereas others can be used to invoke more advanced simulations implemented by developers (possibly with LLM assistance) in ASPERA (e.g., `simulate_org_structure`). The LLM uses information in the query and the AEP to parametrise the simulation and generates complex entities as a result.

Evaluation tools EP generation prompts include evaluation tools to support robust evaluation and access to environment state that is not possible with the tools the assistant uses to compose AEPs. To understand why this is necessary, consider the query *Remind me to check arxiv on Wednesdays.* To execute this action, the assistant must create an `Event` instance and set the `repeats` property to a correctly parametrised recurrence rule (a `RepetitionSpec` instance, shown in Figure 10). Because the recurrence always inherits the parent event parameters, setting `which_weekday=[2]` in this case is optional. More generally, complex recurrences admit multiple parametrisations which are difficult to enumerate for developers. For this reason, we include the `repetition_schedule` tool in the prompt so that the LLM can use it to compare the event instances it returns rather than comparing generator object properties. This ensures robust comparison independent of `RepetitionSpec` parametrisation.

```python
class RepetitionSpec(BaseModel):
    frequency: EventFrequency
    period: int = 1
    recurs_until: datetime.date | datetime.datetime | None = None
    max_repetitions: int | None = None
    which_weekday: list[int] | None = None
    which_month_day: list[int] | None = None
    which_year_month: list[int] | None = None
    bysetpos: list[int] | None = None
    exclude_occurrence: list[datetime.datetime] | None = None
    occurrence_on_date: datetime.datetime | None = None
```

Figure 10: Definition of `RepetitionSpec`, an object used for generating recurring event instances. Documentation omitted for brevity.

[18] The developer is prompted to implement simulation tools after AEP generation and evaluation tools after SIP generation. The implemented tools are displayed in the subsequent SIP/EP generation prompts.

B Assistant library

	time_utils	work_calendar	company_directory	room_booking
Functions				
	now_	get_default_preparation_time	get_current_user	find_available_time_slots
	get_weekday	add_event	find_employee	room_booking_default_time_window
	this_week_datestextsuperscript*	find_events	find_team_of	search_conference_room
	get_weekday_ordinaltextsuperscript*	find_past_events	find_reports_of	summarise_availabilitytextsuperscript*
	parse_time_stringtextsuperscript*	get_calendar	find_manager_of	
	time_by_hmtextsuperscript*	delete_event	get_assistant	
	date_by_mdytextsuperscript*	get_search_settings	get_vacation_schedule	
	get_next_dowtextsuperscript*	find_available_slotstextsuperscript	get_employee_profile	
	get_prev_dowtextsuperscript*	share_calendar	get_all_employees	
	parse_duration_to_calendartextsuperscript*	summarise_calendar	get_office_location	
	parse_durations_to_date_intervaltextsuperscript*	provide_event_details		
	parse_date_stringtextsuperscript*			
	sum_time_unitstextsuperscript*			
	compare_with_fixed_duration			
	modifytextsuperscript*			
	combine			
	intervals_overlaptextsuperscript*			
	replacetextsuperscript*			
Objects				
	Duration	Eventtextsuperscript	EmployeeDetails	ConferenceRoom
	TimeInterval	CalendarSearchSettings	Employee	RoomAvailability
	DateRange			
	RepetitionSpec			
Enums				
	TimeExpressions	ShowAsStatus	Team	
	DateRanges			
	DateExpressions			
	TimeUnits			
	DateTimeClauseOperators			
	ComparisonResult			
	EventFrequency			

Table 9: The ASPERA assistant library defines 62 primitives across 4 domains, implemented by a single developer with GPT-4o assistance. Primitives marked with * were implemented interactively with the LLM using the ChatGPT graphical user interface. For each primitive, the LLM was prompted with the docstring describing the primitive functionality, and its output subsequently refined until the specification was correctly implemented, if necessary. Unit tests were generated in addition to developer-authored tests to verify complex functionality. Primitives marked with †were implemented with partial LLM assistance, where the developer described the functionality to be implemented to the LLM, but substantially refactored and enhanced the code. The LLM was also used for generating unit tests for †primitives.

C Dataset characterisation

C.1 Examples of challenging tasks

```python
def check_boss_availability() -> bool:
    """Check the boss' calendar from Wednesday to Friday next week for availability."""

    # find the current user's manager
    current_user = get_current_user()
    manager = find_manager_of(current_user)

    # calculate the dates for Wednesday to Friday next week
    next_week_monday = get_next_dow("Monday", after=now_().date())
    next_week_wednesday = get_next_dow("Wednesday", after=next_week_monday)
    next_week_thursday = get_next_dow("Thursday", after=next_week_monday)
    next_week_friday = get_next_dow("Friday", after=next_week_wednesday)

    # get the manager's calendar events between Wednesday to Friday next week
    all_events = get_calendar(manager)
    relevant_events = [
        event for event in all_events
        if next_week_wednesday <= event.starts_at.date() <= next_week_friday
    ]
    # check if there are some events that may cover the entire interval
    for e in all_events:
        if (
            (e.starts_at.date() < next_week_wednesday <= e.ends_at.date())
            or (next_week_wednesday <= e.starts_at.date() <= next_week_friday)
            and (next_week_friday < e.ends_at.date())
        ):
            relevant_events.append(e)
    available_slots = []
    for date in [next_week_wednesday, next_week_thursday, next_week_friday]:
        available_slots += find_available_slots(relevant_events, date=date)
    return bool(available_slots)
```

(a) *Assistant, check my boss' calendar Wednesday to Friday next week, are they available for a meeting?* Solving this query involves reasoning about time and having the common sense to account for events spanning multiple days.

```python
def schedule_strategy_review():
    """Schedule a strategy review with the CFO and the COO."""

    # find the CFO and COO
    all_employees = get_all_employees()
    leadership = [
        e
        for e in all_employees
        if get_employee_profile(e).team == Team.Leadership
    ]
    # ceo does not report to anyone
    cfo_coo = [e for e in leadership if find_manager_of(e)]
    # determine the event start time and duration
    one_week_from_today = get_next_dow("Tuesday")
    meeting_time = time_by_hm(hour=2, minute=30, am_or_pm="pm")
    starts_at = combine(one_week_from_today, meeting_time)
    duration = Duration(number=1, unit=TimeUnits.Hours)
    ends_at = modify(starts_at, duration, operator=DateTimeClauseOperators.add)
    # add the event to the calendar
    event = Event(
        subject="Strategy review with CFO and COO",
        starts_at=starts_at,
        ends_at=ends_at,
        attendees=cfo_coo,
    )
    add_event(event)
```

(b) *Assistant, add a strategy review with the CFO and the COO one week from today at 2:30 PM, for 1 hr.* Solving this query involves clever tool use to find the leadership team while taking care to exclude the CEO.

```python
def who_is_busiest_next_week() -> str:
    """Determine which of Bill or Bob is busiest next week."""

    from collections import defaultdict

    def calculate_duration(
        duration_map: dict[datetime.date, list[Duration]]
    ) -> Duration:

        def to_minutes(d: Duration) -> float:
            """Convert the Duration to minutes."""
            if d.unit == TimeUnits.Hours:
                return float(d.number * 60)
            elif d.unit == TimeUnits.Minutes:
                return float(d.number)
            elif d.unit == TimeUnits.Days:
                return float(d.number * 24 * 60)
            elif d.unit == TimeUnits.Months:
                raise TypeError("Cannot convert variable durations to minutes!")
            else:
                raise ValueError(f"Unsupported time unit: {d.unit}")
        total_minutes = 0
        for day, durations in duration_map.items():
            # the largest unit of time is returned for the sum, need
            # to make sure the units are consistent
            this_day_total = to_minutes(sum_time_units(durations))
            total_minutes += this_day_total
        return Duration(total_minutes, unit=TimeUnits.Minutes)

    # Find the employees named Bill and Bob
    bill = find_employee("Bill")[0]  # by structure guideline #1
    bob = find_employee("Bob")[0]  # by structure guideline #1

    # Get their events for next week
    next_week = parse_durations_to_date_interval(DateRanges["NextWeek"])
    bill_events = get_calendar(bill)
    bob_events = get_calendar(bob)

    # Create look-ups for relevant events in the next week
    bill_events_by_day = defaultdict(list)
    for e in bill_events:
        if next_week.start <= e.starts_at.date() <= next_week.end:
            bill_events_by_day[e.starts_at.date()].append(e.duration)

    bob_events_by_day = defaultdict(list)
    for e in bob_events:
        if next_week.start <= e.starts_at.date() <= next_week.end:
            bob_events_by_day[e.starts_at.date()].append(e.duration)

    bill_total_duration = calculate_duration(bill_events_by_day)
    bob_total_duration = calculate_duration(bob_events_by_day)

    # Compare durations and return the name of the busiest person
    if bill_total_duration.number > bob_total_duration.number:
        return "Bill"
    elif bob_total_duration.number > bill_total_duration.number:
        return "Bob"
    else:
        return "Both are equally busy"
```

(c) *Assistant, I need to know which of Bill or Bob is busiest next week so I can allocate work.* Here, summing the event duration involves careful unit conversion in order to provide the correct answer.

Figure 11: Challenging queries from lines 3 -5 of Table 2 as particularly challenging. Figures 11a, 11c and 11b show the sample solutions for these queries respectively, with explanations of their difficulty.

C.2 Further corpus descriptive statistics

Here, we present some further descriptive statistics of *Asper-Bench*. Tables 10 and 11 show some example queries organised according to their complexity, whereas Figures 12 to 15 show how key program complexity measures vary with query length and the distribution of *Asper-Bench* reference AEPs.

Query	Cyclomatic complexity	σ from mean
Assistant, can you tell me when are my manager and skip manager both available on Friday?	1.00	-1.14
Assistant, schedule an urgent meeting with my manager now.	1.00	-1.14
Assistant, schedule a project meeting with my team next Wednesday at 2 PM and block 30 minutes right before for preparation.	1.00	-1.14
Assistant, schedule a project update meeting with my manager before 3 PM tomorrow.	5.00	-0.16
Assistant, schedule a meeting in the afternoon with my engineering colleagues, avoiding any engineering management.	6.00	+0.08
Assistant, remove my second holiday notification from the calendar, something came up.	7.00	+0.32
Assistant, send out a meeting invite to the entire team for a company update next Monday at 2 PM, but exclude those who are on vacation.	7.00	+0.32
Assistant, see if my boss' boss and Jane have accepted my meeting request for tomorrow. If anybody declined, reschedule to take place later but at the earliest available time for everyone, I'm free all day.	19.00	+3.24
Assistant, tell me which days is Sally in office in the third week of August?	20.00	+3.48
Assistant, is there a time in August where everyone from finance is off?	21.00	+3.72

Table 10: Sampling of queries according to cyclomatic complexity of sample solution

Query	# unique primitives	σ from mean
Assistant, how many meetings with Jianpeng are in my calendar at the moment?	2	-1.79
Assistant, cancel everything but the important meetings.	2	-1.79
Assistant, find the names of our assistants please.	2	-1.79
Assistant, schedule a meeting with my manager tomorrow at 10 AM if I have no other meetings then.	9	+0.04
Assistant, provide a summary of my manager's calendar for the next two weeks.	9	+0.04
Assistant, invite the entire sales department to a meeting today from 3 to 5.	9	+0.04
Assistant, schedule a team meeting next Monday at 10 AM, and book a conference room for it. Also, schedule a follow-up meeting one week later at the same time and book the same room.	18	+2.39
Assistant, can you schedule a 30 mins recurring weekly meeting with the engineering team on Fridays at 3 PM for the next two months? If there are clashes, tell me their dates, don't double book.	19	+2.65

Table 11: Sampling of queries according to number of unique primitives in sample solution

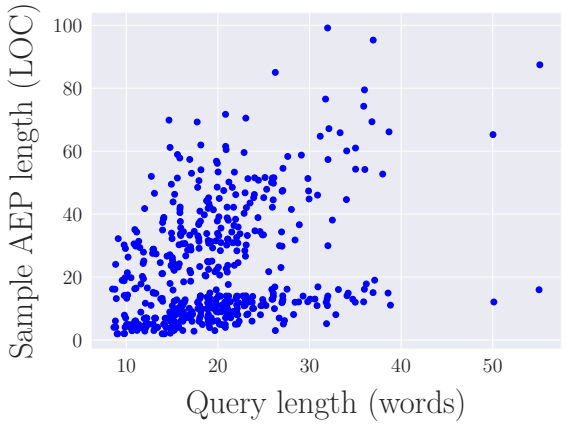


Figure 12: *Asper-Bench* AEP query length vs program length

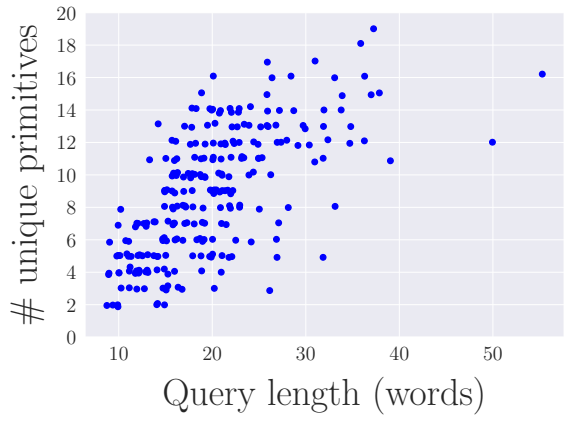


Figure 13: *Asper-Bench* AEP query length vs number of unique primitives

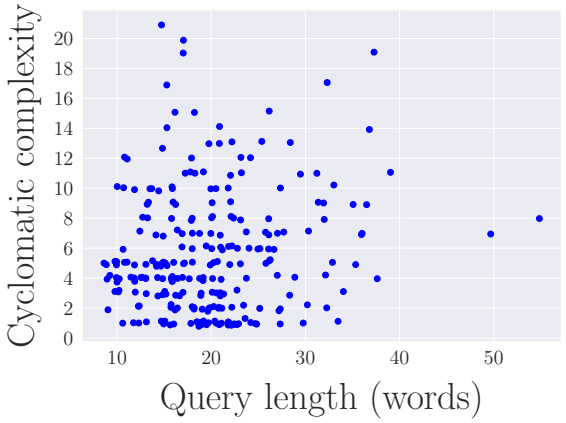


Figure 14: *Asper-Bench* AEP query length vs cyclomatic complexity

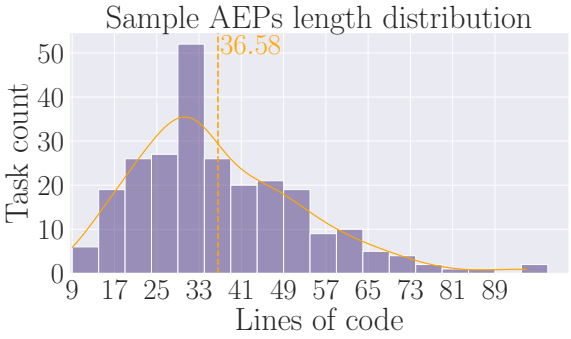


Figure 15: *Asper-Bench* AEP length distribution

C.3 ASPERA policy

```python
class RequiresUserInput(Exception):
    """An exception to be raised when an assumption has to be made in order
    to continue the program. Typical situations involve:

        - indexing search results when multiple or no search results are returned
        but the user query implies the uniqueness of the search results.
        - inform the user a task could not be completed (eg no suitable meeting
        room could be booked for a meeting as per the user request).


    For additional details, consult program structure guidelines below (displayed
     only if any special guidance applies).

    Notes
    -----
    1. Cases when no or multiple search results are returned should have distinct messages
     to indicate why the error was raised.
    2. The number of returned results should always be stated when the error is raised because
     a search returned more than one result.
    3. This error *should not* be raised to respond to user questions - an appropriate object
    should be returned instead. However, if the question cannot be answered (eg user asks
    about meeting which cannot be found), then this exception can be raised.
    4. If the user warns the assistant of possible conflicts when making their request (eg
    schedule something if it doesn't conflict with something else), this error should not be raised.
    5. Employee names can be assumed unique unless the user request implies this is not the case. In this
    case it is not necessary to raise this error and raising it will have no effect."""
```

(a) `RequiresUserInput` documentation

```python
def reschedule_monthly_sync():
    """Reschedule the monthly sync meeting."""

    # find the monthly sync meeting
    events = find_events(subject="Monthly Sync")
    # by structure guideline #2
    if len(events) == 0:
        raise RequiresUserInput("No monthly sync meetings found.")
    elif len(events) > 1:
        raise RequiresUserInput(
            f"{len(events)} monthly sync meetings found. Please provide more details."
        )
    else:
        monthly_sync = events[0]

        # from now_() we now it is Tuesday 25th, and 1st Monday of next month is actually next monday
        first_monday_next_month = get_next_dow("Monday")
        new_meeting_time = time_by_hm(hour=9, minute=0, am_or_pm="am")
        new_meeting_start = combine(first_monday_next_month, new_meeting_time)
        new_meeting_end = modify(
            new_meeting_start,
            monthly_sync.duration,
            operator=DateTimeClauseOperators.add
        )

        # update the event
        monthly_sync.starts_at = new_meeting_start
        monthly_sync.ends_at = new_meeting_end

        # add the updated event to the calendar
        add_event(monthly_sync)
```

(b) LLM-generated system policy for error handling and disambiguation

Figure 16: ASPERA employs exceptions to generate reference AEPs which follow a simple policy according to which the assistant can raise to inform a user a certain task could not be completed due to environment constraints or when disambiguation is required to identify entities mentioned in the user request. The sample solutions contain 144 `RequiresUserInput` usages across 78 programs. In addition, the top two guidelines in Figure 6c enforce a simple events scheduling policy.

D ASPERA evaluator prompt templates

```
You are an expert programmer working with my team which is
specialising in developing AI assistants. Your current task is to
translate a complex user request into a `python` program using
our application backend below:

```python
{{ code }}
```

Here are some examples your colleagues shared with you to help you to
understand the solution format and some assumptions about our
application backend.

```python
{{ query_solution_examples }}
```

The examples above follow the
{{ guidelines.generation_labelling | length }} structure
guidelines listed below. You must adhere to these when writing your
solution.
{% for instruction in guidelines.generation_labelling %}
{{ loop.index }}. {{ instruction }}
{%- endfor %}
```

(a) Prompt template for AEP generation, shared by CCK and PS agents. See `guidelines` below.

```
- Unless the user explicitly states, meetings should not be scheduled on or recur during weekends.
- Work meetings can only happen during the times prescribed in the time_utils library unless the user explicitly states otherwise.
- The leadership team is formed of a CEO, COO, CFO. Department heads report to either the COO or the CFO.
- Use the tools in the time_utils library to reason about time. Hence, current date and time on the user device should be found using the tools
  and documentation in this library and not the datetime library.
- Information-seeking queries should return an appropriate object to the caller; avoid simply printing the information inside your solution.
- If you need to format dates in a string, use strftime('%Y-%m-%d'). For datetime objects use strftime('%Y-%m-%d %H:%M:%S').
- Make sure to escape \n characters.
- Type annotate the return for programs which have a return type which is not None
- Only the first Python markdown block will be executed, so if you wish to use helper functions, these should be defined locally inside your
  solution.
- Only import modules from the standard library that you need for your programs (eg import collections). Imports from our application backend will
  be automatically done when we execute the program you generate.
```

(b) The first two guidelines implement a simple events schedule policy. The third provides additional information about the environment, required to solve a range of queries involving the organisation leadership. The remainder of the guidelines are concerned with various aspects of the AEP structure such as time grounding, return type, function nesting and importing. These guidelines were designed to minimise execution errors due to mismatches between the simulation environment and model behaviour following detailed error analyses on initial agent development iterations.

Figure 17: ASPERA AEP generation prompt template

```
You are a programmer using a Python library of personal assistant
tools in order to write a program that executes a user query.
You will be shown signatures from a Python module and a query, and will
be asked to formulate Python import statements importing any tools
that might be relevant to writing a program that executes the user
query.

When writing the program, you will be asked to follow the
{{ guidelines | length }} structure guidelines listed below.
{% for instruction in guidelines %}
{{ loop.index }}. {{ instruction }}
{%- endfor %}
Use this additional information to guide your import decisions.

Module:
{{ module }}
Query: {{ query }}

Think carefully, and output the relevant Python import statements,
or None. Any code you write must be included in a Python markdown
block (ie start with a "```python" sequence and end with "````").
If there are no relevant tools in the current module being shown,
simply output None.
```

(a) Primitives selection prompt template.

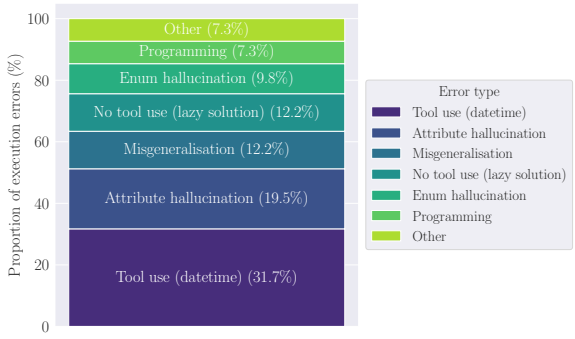
- Use the tools in the time_utils library to reason about time. Hence, current date and time on the user device should be found using the tools and documentation in this library and not the datetime library.
- Work meetings can only happen during the times prescribed in the time_utils library unless the user explicitly states otherwise.
- The leadership team is formed of a CEO, COO, CFO. Department heads report to either the COO or the CFO. Appropriate tools will have to be imported and combined to resolve these employees to Employee objects required by all APIs.

(b) Guidelines presented to the agent during at each primitive selection iteration step. These are a subset of the guidelines defined for the CCK prompt in Figure 17b, including only the instructions which can influence primitive selection.

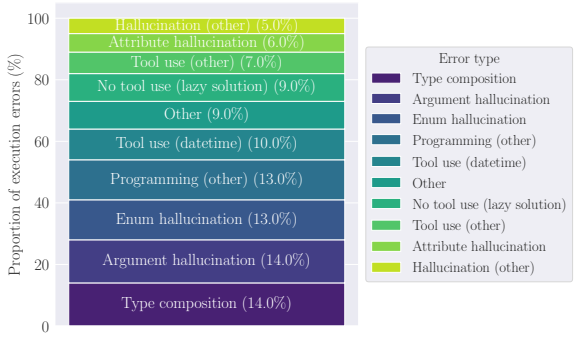
E Analysis supplementary material

E.1 Execution errors

We debug the AEPs generated by the the best GPT-4o and GPT-3.5-turbo runs[19] for the first 125 queries in our corpus (50% of the data), analysing a total of 141 execution errors (Table 6) which we classify into several categories depicted in Figure 19 and for which representative examples are shown and explained in Table 12. We find execution errors occur because the LLMs hallucinate in preference to performing additional problem solving steps. While GPT-4o fails to appropriately combine the primitive to perform non-trivial compositions for date and time reasoning (row 1, Table 12) or simple arithmetic reasoning (row 5), GPT-3.5-turbo additionally fails to appropriately exploit type relations to compose primitives (row 6) and often hallucinates API arguments (row 7), demonstrating very limited ability to program according to complex set of constraints defined by an assistant library.



(a) GPT-4o



(b) GPT-3.5-turbo

Figure 19: Execution error classification for the first 125 ASPER queries

[19] Success rate of 46% and 11.2%, respectively.

Id	Query	Error Snippet

1	Assistant, schedule our team Christmas party 10 days before Christmas. Should start in the morning and end at 10 PM.	`def schedule_team_christmas_party():`
		
		`  # find the user's team to determine event attendees`
		`  ...`
		
		`  # resolve the date for 10 days before Christmas`
		`  christmas_day = parse_date_string(DateExpressions.ChristmasDay)`
		`  party_date = modify(`
		`  christmas_day,`
		`  Duration(10, TimeUnits.Days),`
		`  operator=DateTimeClauseOperators.subtract,`
		`  ).date()`

Tool use (datetime): Line 9 contains a `TypeError`, `modify` only accepts `datetime` objects. A correct solution requires an additional reasoning step: pass `christmas_day` and one of specified times to the `combine` library function to get the correct type.		

2	Assistant, set up a training session for all employees from the Engineering team next Monday from 2 PM to 5 PM. Send out invites and book a conference room that fits 20 people.	`def schedule_engineering_training_session():`
		
		`  # Find all employees in the Engineering team`
		`  engineering_team = [`
		`  emp for emp in get_all_employees()`
		`  if emp.team == Team.Engineering`
		`  ]`

Attribute hallucination: In, line 6 the `.team` attribute access raises an error because the `Employee` objects returned by `get_all_employees` only have `name` as attribute. The `Employee` object should be passed instead to the `get_employee_profile` library function to return an object which has `team` as an attribute.		

3	Assistant, can you schedule a 30 mins recurring weekly meeting with the engineering team on Fridays at 3 PM for the next two months? If there are clashes, tell me their dates, don't double book.	`from typing import list`
		
		`def schedule_weekly_meeting_with_engineering_team() -> (`
		`  list[datetime.date] | None`
		`):`

Misgeneralisation: The assistant triggers an import error in line 1. The pretraining data contains `from typing import List`, a common idiom for static typing prior to PEP 585 (2019). When prompted to return an object of type `list[datetime.date] | None`, the model does not make this distinction and misgeneralises by generating line 1.		

4	Assistant, put 45 minutes in the calendar, back-to-back, with Engineering and Marketing starting at 10 AM tomorrow... Actually, add a 10-minute buffer between each meeting.	`def schedule_back_to_back_meetings():`
		
		`  # find the user's profile`
		`  ...`
		
		`  # find the teams`
		`  engineering_team = find_team_of(Employee(name="Engineering"))`
		`  marketing_team = find_team_of(Employee(name="Marketing"))`

No tool use (lazy solution): The assistant hallucinates lines 7–8 instead of using relevant APIs to find the engineering team, in spite of documentation that states that `Employee` objects cannot be instantiated. The functions `get_all_employees`, `get_employee_profile` and the enumeration `Team.Engineering` should have been composed, similar to snippet in row 2.		

5	Assistant, mark my vacation from next Tuesday for 2 weeks and cancel all my meetings during this period.	`def mark_vacation_and_cancel_meetings():`
		
		`  # Determine the vacation start and end dates`
		`  next_tuesday = get_next_dow("Tuesday")`
		`  vacation_start = next_tuesday`
		`  vacation_end = modify(`
		`  vacation_start,`
		`  Duration(2, TimeUnits.Weeks),`
		`  operator=DateTimeClauseOperators.add,`
		`  ).date()`

Enum hallucination: The assistant uses the enum value `TimeUnits.Weeks` (line 8), which is undefined. The library deliberately defines the `TimeUnits` members as "Hours", "Minutes", "Days", "Months" so that assistants have to perform simple unit conversions.		

6	Assistant, notify me of overlapping meetings this week.	`def notify_overlapping_meetings_this_week() -> list[Event] | None:`
		`  # Find all future events in the user's calendar`
		`  ...`
		`  # Create a list to store overlapping meetings`
		`  ...`
		`  # Get the dates for the current week`
		`  ...`
		`  # Create a dictionary to store events by date`
		`  events_by_date = defaultdict(list)`
		
		`  # Populate the events_by_date dictionary with events happening`
		`  ...`
		
		`  # Check for overlapping meetings`
		`  for date, events in events_by_date.items():`
		`  for i, event1 in enumerate(events):`
		`  for event2 in events[i + 1 :]:`
		`  if intervals_overlap(event1, event2):`

Type composition: The assistant calls `intervals_overlap` with `Event` instead of `TimeInterval` types (line 18). The latter must be instantiated from the event properties.		

7	Assistant, block time for preparation before important meetings.	`def block_preparation_time_for_meetings():`
		`  # Find the user's upcoming important meetings`
		`  user = get_current_user()`
		`  important_meetings = find_events(attendees=[user], event_importance="high")`

Argument hallucination: The assistant calls `find_event` with `event_importance` keyword (line 4). Valid `find_event` arguments are `attendees` and `subject`.		

Table 12: Sample execution errors

E.2 Task completion error examples

Id	Query	Agent action
1	Assistant, Ari and James are on holiday next month, who's out for longer?	Sums duration of all vacations, month notwithstanding.
2	Assistant, reorganise my diary on the fifth so that the important meetings come first.	Sets the importance of the first low-priority meeting to "high" and all other events to "normal", without any further updates.
3	Assistant, is there a time in August where everyone from finance is off?	Returns `True` for the first employee whose vacation starts in August.
4	Assistant, book a conference room for the meeting with sales tomorrow at 2 PM.	Assumes the user is part of the sales team, scheduling a meeting with the wrong attendees as a result.
5	Assistant, add bi-weekly mentorship sessions with the reports of my reports starting next Monday at 2 PM to my calendar.	Hallucinates an end date for the recurrent event, scheduling instances only for six months.
6	Assistant, add a reminder 1 hour before all important meetings, with the meeting title in the subject.	Disregards `add_event` documentation according to which the user should not be a member of attendees lists for events in their own calendar.
7	Assistant, schedule by-monthly team training sessions on the first Monday at 10 am for hires who joined since the 1st of May, alternating between the Engineering and Sales and Marketing.	Cannot correctly resolve the meeting start dates scheduling two meetings which start at the same time in the first Monday of the current month, which has already passed.
8	Assistant, cancel all my meetings Wednesday next week and mark me out of office	Cancels meetings on Wednesday in the current week instead
9	Assistant, how many employees called John are in my team?	Exact matches the `name` attribute instead of calling `find_employee('John')` `and filtering to ensure returned employees are in user's team`
10	Assistant, what date did Joris and Pete meet last week?	Wrong information provided to the user because the model is looking for a meeting involving Joris and Pete in user's calendar as opposed to checking either Joris' or Pete's calendar.
11	Assistant, reschedule the meetings which overlap with "annual review" this afternoon to the same time tomorrow.	Adds copies of overlapping events tomorrow, instead of modifying existing events.
12	Assistant, schedule a 30 mins meeting with Frank from finance at 10 AM in any available meeting room.	Schedules a meeting in the wrong room, choosing the first room returned by the room search API without first checking availability for the entire duration specified by the user.
13	Assistant, can you find a room that can accommodate 20 people for a meeting on Thursday afternoon?	Incorrectly processes serch results, returning rooms that are not available during the stated interval
14	Assistant, who in our team has not booked any vacations yet?	Includes the user in the list of returned names, not expected since the user was asking about other team members, not themselves.
15	Assistant, reschedule all meetings from today to next Monday.	Reschedules all the meetings happening until next Monday to next Monday instead of rescheduling today's meetings.

Table 13: Sample task completion errors for gpt3.5-turbo (rows 1-3), gpt-4o-mini (4 - 6), gpt-4o (7 - 9), o1-mini (10 - 12) and o1 (13 - 15)

E.3 Handback control error examples

Id	Query	Agent action
1	Assistant, find a suitable conference room for a meeting with my team I wanna schedule later today.	Tries to schedule a meeting, handing back control because of incorrect diary checking.
Error cause: Distracted by irrelevant info. The agent is not required to schedule a meeting, not enough details are provided. Instead, it should have searched for a room that is available and has sufficient capacity to accommodate the user and their team.		
2	Assistant, can you find a time slot in my diary today when I could schedule something with the HR department to discuss my performance review?	Hallucinates a program attempting to find HR team, handing back control because it cannot determine it.
Error cause: Distracted by irrelevant info. The HR team is not defined in the simulation. The task requires the agent to find a slot in user's diary.		
3	Assistant, schedule our team Christmas party 10 days before Christmas. Should start in the morning and end at 10 PM?	Requires the user to provide an alternative date.
Error cause: Following policy. The agent follows the instruction *Unless the user explicitly states the date, meetings should not be scheduled on or recur during weekends.* which is irrelevant.		
4	Assistant, schedule a follow-up meeting two weeks after my last one-on-one with my manager.	Hand back control because it cannot find the 1:1 meeting.
Error cause: Documentation comprehension. The agent fails to follow a note according to which the user should not be specified as an attendee during search by convention. The note is included in `find_events` docs and referenced in `find_past_event` documentation.		
5	Assistant, move back my meeting with John from sales and Jane by one hour.	Hands back control because it determines two employees named John are part of the sales team.
Error cause: Unwarranted disambiguation. The event can uniquely determined by checking the calendar.		

Table 14: Examples of queries where o1 mistakenly hands back control to the user.

E.4 Problem categories

In Table 7, we report task success for five problem categories. Table 15 lists the queries which were used to estimate the performance per problem category. For each query, a model predicts three AEPs with different random seeds, so 30 task completion outcomes are considered when estimating subset performance.

Simple
Assistant, how many meetings with Jianpeng are in my calendar at the moment?
Assistant, plan a weekend trip to the beach with my work colleagues Alice and Bob starting Saturday morning.
Assistant, schedule lunch with my entire team tomorrow at noon.
Assistant, schedule a 3-hour workshop with my team next Monday starting at 1 PM.
Assistant, schedule a meeting with my manager at lunch tomorrow.
Assistant, schedule an urgent meeting with my manager now.
Assistant, share my calendar with my assistant.
Assistant, cancel everything but the important meetings.
Assistant, schedule a team event next Tuesday at 4 PM for 2 hours at the bowling alley.
Assistant, cancel my meeting with Pete and move my meeting with Jianpeng in that slot instead
Constrained scheduling
Assistant, schedule a project update meeting with my manager when I'm free, before 3 PM tomorrow.
Assistant, schedule a project update meeting with my manager when we're both free, before 3 PM tomorrow.
Assistant, set a 3 to 4 meeting in room z with any team members available then.
Assistant, set a 30 mins meeting with Jianpeng at the earliest time when we are both free today.
Assistant, reschedule today's meetings to Monday - keep the same time. If you detect clashes the rescheduled meetings should start as soon as possible after the end of existing events. No overlaps!
Assistant, find an available slot for a 30-minute meeting with my team two weeks from now.
Assistant, is it possible to schedule a team meeting tomorrow 10 am to 11:30 am or is any colleague from my team busy?
Assistant, check my boss' calendar Wednesday to Friday next week, are they available for a meeting?
Assistant, set up a status update meeting with my manager every last Friday of the month at 2 PM till the end of the year. Skip the ones on his holidays.
Assistant, my manager just told me of a clash with our 1:1 tomorrow, reschedule it to the latest free slot we're available.
Complex time expressions
Assistant, show me the last time I met with Alice.
Assistant, schedule a 45-minute team follow-up call two weeks after tomorrow's project deadline, keeping the start time.
Assistant, schedule our team Christmas party 10 days before Christmas. Should start in the morning and end at 10 PM.
Assistant, schedule a 1-hour meeting with my manager, then a 45-minute meeting with my team, followed by a 30-minute meeting with the sales team. Add a 15-minute buffer between each meeting starting tomorrow at 9 AM.
Assistant, put 45 minutes in the calendar, back-to-back, with Engineering and Marketing starting at 10 AM tomorrow... Actually, add a 10-minute buffer between each meeting.
Assistant, find an available conference room for my next meeting and schedule it there.
Assistant, book me out of office for the last two hours of the working day the day before my vacation in October.
Assistant, schedule a 1-hour review meeting with my sales team next Monday at 10, then one with finance right after that, and one with engineering after a 30 mins break.
Assistant, block the last hour of the working day for a catch-up with my team the day before any of their vacations start.
Assistant, change our weekly team meeting to happen on Thursday instead, with a update to say 'friday is a no-meeting day'?
Policy / instruction following
Assistant, schedule a meeting with my team every day next week at 3 PM.
Assistant, plan an off-site event with my team this weekend at Central Park starting at 10 AM.
Assistant, schedule lunch with a different team member each day next week at 12:30 PM.
Assistant, block 90 mins of focus time every morning at 8 AM for the next two weeks.
Assistant, I've got an urgent task that needs 3 hours starting at 1 PM tomorrow. Reschedule my existing meetings to fit this in, but try to keep the same day.
Assistant, schedule a meeting with my team late afternoon tomorrow. Mark Alice optional.
Assistant, reorganise my diary on the fifth so that the important meetings come first.
Assistant, add a strategy review with the CFO and the COO one week from today at 2:30 PM, for 1 hr.
Assistant, set 30 minutes tomorrow late afternoon with the department heads from engineering, finance and marketing.
Assistant, add a reminder 1 hour before all important meetings, with the meeting title in the subject.
Advanced problem solving
Assistant, find a suitable conference room for a meeting with my team I wanna schedule later today.
Assistant, see if my boss' boss and Jane have accepted my meeting request for tomorrow. If anybody declined, reschedule to take place later but at the earliest available time for everyone, I'm free all day.
Assistant, schedule a meeting in the afternoon with my engineering colleagues, avoiding any engineering management.
Assistant, find an available conference room for my next meeting and schedule it there.
Assistant, block 2 hours of free time for holiday preparation after dinner on the last working day before my next vacation.
Assistant, I will need to schedule an important retrospective sometime next week, how many rooms accommodating between 8 and 12 people do we have?
Assistant, add a finance manager to my meeting with the marketing manager.
Assistant, who in finance is yet to book a holiday this year?
Assistant, Ari and James are on holiday next month, who's out for longer?
Assistant, what's ratio of Diarmuid to Anders holidays from the start of the year till the second of July?

Table 15: Listing of queries for which task success is reported in Table 7.

E.5 Primitive selection

Below, we report primitive selection results broken down for three key ASPERA modules. "Task success" represents the task success rate for queries whose sample solution made use of the primitive in question. The final row shows the global precision, global recall, micro F1 and mean task success across primitives in the module.

`work_calendar`

Primitive	Precision	Recall	F1	CCK task success (1-shot)
find_past_events	0.83	0.91	0.87	0.73
RepetitionSpec	0.76	0.94	0.84	0.68
find_events	0.97	0.71	0.82	0.73
summarise_calendar	1.00	0.67	0.80	0.67
get_default_preparation_time	0.67	1.00	0.80	0.00
Event	0.73	0.78	0.76	0.64
delete_event	0.79	0.65	0.71	0.78
find_available_slots	0.73	0.64	0.68	0.76
get_calendar	0.73	0.55	0.63	0.69
add_event	0.97	0.41	0.58	0.66
CalendarSearchSettings	0.29	0.50	0.36	0.75
ShowAsStatus	0.33	0.12	0.18	0.56
get_search_settings	0.33	0.09	0.14	0.73
Overall	0.62	0.61	0.61	0.66

`company_directory`

Primitive	Precision	Recall	F1	CCK task success (1-shot)
get_all_employees	0.98	0.83	0.90	0.71
get_employee_profile	0.87	0.92	0.90	0.71
get_current_user	0.89	0.89	0.89	0.72
Team	0.97	0.79	0.87	0.67
find_reports_of	0.95	0.77	0.85	0.81
find_employee	0.92	0.77	0.84	0.74
find_team_of	0.98	0.68	0.81	0.74
get_vacation_schedule	0.73	0.83	0.77	0.76
get_assistant	1.00	0.60	0.75	1.00
find_manager_of	0.86	0.63	0.73	0.65
Employee	0.05	0.50	0.09	0.75
Overall	0.66	0.74	0.70	0.74

`room_booking`

Primitive	Precision	Recall	F1	CCK task success (1-shot)
room_booking_default_time_window	0.75	1.00	0.86	1.00
find_available_time_slots	0.50	0.50	0.50	0.50
search_conference_room	0.30	0.89	0.45	0.72
summarise_availability	0.06	1.00	0.12	0.67
Overall	0.27	0.42	0.33	0.72

Table 16: Primitive selection results broken down for three ASPERA modules. The final column shows o1's task success in the CCK setting for the subset of queries whose sample solution made use of the primitive in question. This can be thought of as a proxy for how well the model is able to make use of this tool, in contrast to how well it is able to select it.