Generating Tool-Usage Tests from Minimal Developer Input for Custom Tool-Calling Agents

GIACOMO CALCATERRA, University of Twente, The Netherlands

Recent advances in architecture and scale have enabled LLMs to invoke external tools, greatly expanding what an agent can do. Yet, as the number and complexity of those tools grow, agents can still produce malformed or incorrect calls. Many existing studies focused on testing LLMs capabilities to assemble such tool calls, mostly focusing on the comparison between different foundation models. However, most of the recently released models are increasingly able to do so for specific tasks thanks to dedicated fine-tuning, shifting the cause of the issue to custom prompting, tool complexity, and tools combination and description. To test such specific situations it is necessary to perform manual tests or produce hand-curated testing scenarios. The aim of this paper is to show how, combining existing generation and evaluation techniques, it is possible to generate and execute a set of tailored tests for custom tools, tools combinations and prompt only requiring minimal additional information, and removing the need for manual intervention during the process. Our code is available at this URL.

Additional Key Words and Phrases: Large Language Models, Agents, Tools, Automated Generation, Testing

1 INTRODUCTION

Large Language Models (LLMs) are extremely powerful Natural Language (NL) generators, able to produce syntactically and logically correct sentences in most cases [16] [5]. Since their capabilities have significantly improved in the past years [14] [6] [4], experts have leveraged their pattern recognition and understanding capabilities to introduce the concept of agents and tool-calling [19] [13]. Tools are functions that can receive input and return output. Tool-calling for agents refers to the ability to ask an LLM to generate calls to these functions. These calls can then be parsed and executed, inserting the resulting output into the initially generated response. This allows a certain degree of agency to LLMs, enabling them to interact with external resources in multiple iterations.

However, tools can be very complex, and their use usually requires advanced planning capabilities [17] [21]. Although foundation models are becoming increasingly better at planning [18], a successful agent also needs clear prompt instructions and tool descriptions. Any modification in the prompt, tool descriptions, or foundation model used for inference can alter overall performance. Since the output is NL, evaluating the consequences of those changes can be very challenging and requires significant manual work.

Recent academic work has developed several strategies to test the ability of foundation models to make correct use of tools, but the majority neglected the complexity of custom agents with specific prompts and tool descriptions, focusing instead on predefined lists of tools and hand-curated datasets. Additionally, the evaluation of the results is not always sufficiently detailed to draw meaningful conclusions, thus requiring further manual analysis.

The main research question that guides this work is the following:

> How can a comprehensive set of tests be automatically generated from minimal developer input to evaluate whether a custom agent invokes the intended tools with the correct arguments, in the correct order, and produces contextually appropriate output in realistic conversational scenarios?

To address it, the following sub-questions need to be investigated:

SQ1 What developer-supplied information is minimally required to provide sufficient context to generate valid ground truth for tests?

SQ2 What noise-injection strategies can produce realistic conversation scenarios that effectively cover broader edge cases?

SQ3 Does T-Eval's six-dimensional scoring provide meaningful and actionable metrics for evaluating complete multi-turn conversations?

In this paper, in Section 2 we first give an overview of the available literature relevant to the investigated issue. In the following section, we describe the preliminary work necessary to address the sub-question **SQ1**. Section 4 illustrates the implementation of the pipeline, explaining in detail the approach used. In Section 5 we present the results produced by the newly introduced framework and in Section 6 we discuss the implications, potential limitations and the proposed future steps. Finally, Section 7 presents the answers to the proposed research question and the overall result of this work.

2 RELATED RESEARCH

With the increasing use of LLMs as tool-calling agents, many research groups have focused on the development of testing methodologies for this application. Previous work has focused mostly on single interaction tool calling [11] sometimes allowing for multiple tool calls [15]. More recent papers analyzed increasingly complex scenarios [9] but still performed tests on a given set of tools provided in their work. However, for many business applications it is necessary to develop custom tools tailored to a specific sector or company.

The work by Arcadinho et al. [1] focuses on automating the generation of tests dynamically using LLMs. They use intermediate graphs to reduce LLMs' tendency to hallucinate and to increase the coverage of diverse conversation scenarios. With this approach, they address the issue of a static dataset limited to a specific set of tools. While their generation process primarily delegates tool ideation to the LLM, based on user requests, they explicitly mention that existing tools can be easily included in the pipeline. Unfortunately, they do not provide any specific evaluation methodology and they require manual verification of generated tools and conversations.

TScIT 43, July 4, 2025, Enschede, The Netherlands

Several papers provide relatively simple evaluation techniques, focusing on the correctness of the tool choice and arguments [10] or on the final response quality [11]. Alternatively, work by Yu et al. [20] provides a more nuanced evaluation, but this comes with a high computational cost and higher instability.

The work by Chen et al. [3] focuses attention on all the intermediate steps necessary for a correct tool call. It introduces T-Eval, a six-dimensional evaluation framework aimed at decomposing the evaluation into sub-tasks, to deepen the understanding of potential issues. While T-Eval metrics use Sentence-BERT embeddings and the Hopcroft-Karp matching algorithm, incurring some transformer and graph-matching computational overhead, they avoid full LLM inference on each example, making them substantially lighter and more stable than LLM-based evaluators.

2.1 Gaps

After analyzing the previous work, the main gaps identified are:

(1) Inability to test custom tools and tool combinations.
(2) Lack of detailed evaluation metrics.
(3) Necessity of substantial manual curation.

To address the first gap, the work by Arcadinho et al. [1] provides a generation pipeline that can be used to produce tests, and it can be used with any set of tools. This process does not require any manual intervention, thereby addressing the third gap. However, it does not provide any specific evaluation methodology.

To address the second gap, the T-Eval framework by Chen et al. [3] provides a nuanced set of metrics to evaluate the correctness of the tool calls, including the thought process behind the decision, the action to be performed, and the review of the result. However, it is not designed to evaluate the correctness of the conversation as a whole, but rather to evaluate the correctness of a single query. To address this, we will further extend the T-Eval framework to create aggregated metrics providing additional insights.

By combining the two approaches, we can produce a fully automated pipeline that generates tests for custom tools and tool combinations, and evaluates the results using a detailed set of metrics, addressing all the gaps identified.

3 PRELIMINARY ASSESSMENT

Sub-question **SQ1** investigates the need to require additional information about the context to successfully generate a valid ground truth. This necessity comes from the fact that the focus of this research is on custom-developed tools and prompts, which may include specific contextual information critical for generating a meaningful conversation.

Since there is no relevant literature available about this specific issue, to address it, we decided to rely on the evaluation of sector experts to validate what the minimally required information that should be available in the LLM's context is. We were able to interview three experts. The validation process involved multiple iterations of feedback and refinement.

The first step was to define a set of questions to ask the experts, which focused on what the most common problems faced with tool-calling assistants were and what the missing pieces of information in the LLM's context were, especially related to tool usage. This information was used to craft the first version of the structured object that is used as input for the intent generation step.

In the second iteration, the experts were asked to review the structured object and provide feedback on its completeness and relevance. The feedback was used to refine the object, removing some fields and restructuring some of the entries.

The third iteration served as final validation of the structured object, where the experts were asked to review the final version and confirm its adequacy. The final result of these consultations is explained in the subsection below.

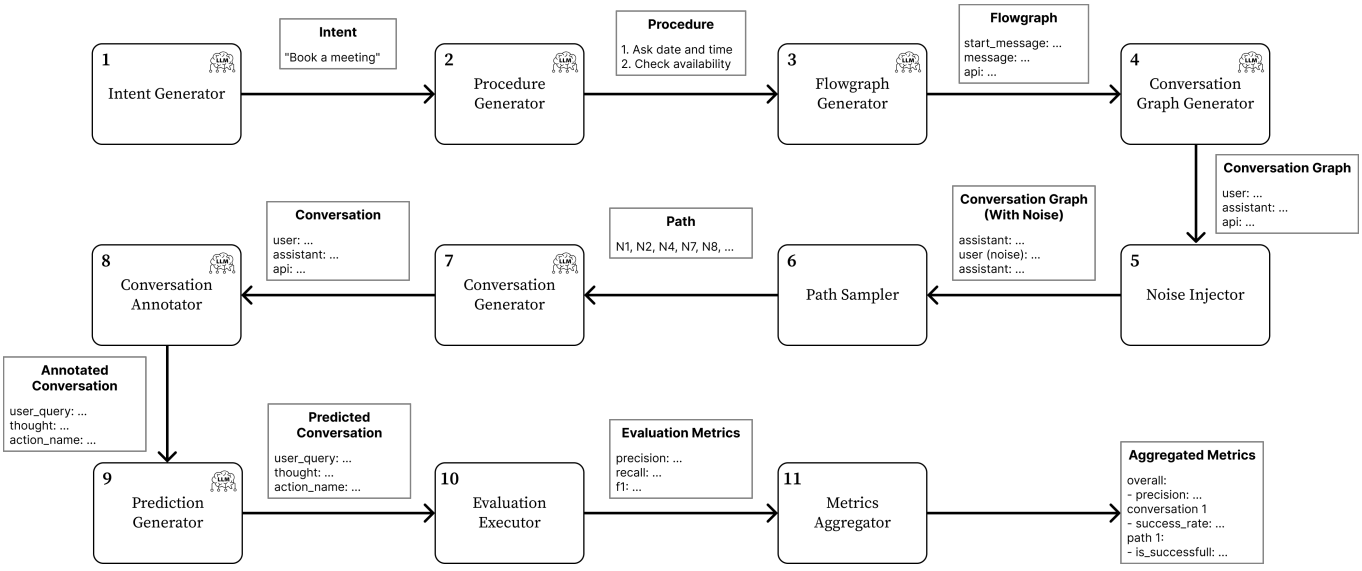

Fig. 1. A diagram overview of the steps of the pipeline

3.1 Minimal Developer Input

The first relevant set of information is related to the operative context. Each agent is intended to fulfill a specific role in a given environment. To produce relevant testing scenarios, the experts reported that the role and the assumptions that hold true in such an environment need to be made explicit. This avoids producing a ground truth that ignores these preconditions and thus does not represent a realistic conversation. It is also necessary to know the full agent prompt, which may contain additional business logic, relevant for tool usage.

When assembling a custom combination of tools, it is common to have groups of tools that have a common purpose, similarly to what happens with API endpoints dedicated to Create Read Update Delete (CRUD) operations on the same entity. From the experts' feedback emerged that a clear definition of such groups can significantly improve the quality of the intents, providing a deeper understanding of the intended use of the collection of tools.

All the information mentioned is part of the minimal required input used to increase the quality and relevance of the ground truth generated using LLMs. Since the pipeline has some other configuration information necessary, these have been merged into one unique configuration object. The structured object containing all these values is shown in the Appendix B.

4 METHODOLOGY

The pipeline developed, illustrated in Figure 1, is composed of eleven steps, each of which is implemented as a class. Seven steps involve the use of an LLM to generate the necessary information, indicated with an icon in Figure 1, while the remaining four steps are implemented as simple functions.

The first eight steps are used to generate the ground truth, following the methodology outlined in the paper by Arcadinho et al. [1]. Step nine is when the predictions to evaluate are generated, and step ten is when the evaluation results are produced using the T-Eval framework by Chen et al. [3]. Finally, step eleven aggregates the results of the evaluation to produce a meaningful summary of the agent's performance.

Throughout this methodology, the terms "tools" and "APIs" are used interchangeably, following the terminology established in the foundational frameworks by Arcadinho et al. [1] and Chen et al. [3], since tools are the equivalent of APIs for agents.

4.1 Implementation Details

We developed the testing pipeline in TypeScript to benefit from enhanced type safety, making the codebase clearer and easier to maintain. As a runtime environment, we selected Deno, primarily because it natively supports TypeScript and includes a Jupyter notebook kernel, which significantly improves the experimentation process.

To simplify the implementation of the pipeline, we used the following libraries:

- `LangChain`: provides high-level abstractions for prompt construction and LLM interaction, reducing implementation complexity
- `Zod`: enforces JSON schema validation on LLM outputs, ensuring that every generated object strictly conforms to the expected structure
- `graphlib`: handles creation, validation, and traversal of graphs, providing a simple way to check the correctness of the generated graphs

Each pipeline step is implemented as its own TypeScript class, encapsulating the logic and making it easier to manage dependencies and state. The only exception to this object-oriented structure is the evaluation step explained in Section 4.11. Since the T-Eval evaluation framework is implemented in Python, we used a bash script to execute this step. The script takes the exported JSON data from the previous step as input, orchestrates the evaluator execution, collects the output, and organizes it into structured directories for the next step.

4.2 Intent Generation

The first step of the pipeline involves the generation of one or more intents, which will be used as the goal of the conversation. An intent is a sentence in NL that describes the desired outcome of the conversation.

To generate a meaningful intent, it is necessary to provide some additional information about the context in which the agent operates. The work by Arcadinho et al. [1] is focused specifically on customer support scenarios, so their prompt is tailored to that specific use case. However, the goal of this work is to produce a more general framework that can be used in any context, so we designed this step to receive contextual information as input parameter.

The simplest way we identified is to require a structured object defining what should be provided. The structure and the reasoning behind the contextual information part have been explained in Section 3.1.

The other input for the intent generation is the list of available tools. As previously mentioned in Section 2, Arcadinho et al.'s framework allows the use of existing tools, but it is not clearly stated how to provide them. To standardize its definition, recently one of the major AI laboratories, Anthropic, introduced the Model Context Protocol (MCP), which is a communication standard focused on agentic applications. This standard provides a simple way to fully describe a tool. Since the industry adoption is already significant [8], we decided to use their structured definition to ingest the custom list of tools.

4.3 Procedure Generation

After the generation of an intent to guide the conversation, we use an LLM to generate a procedure, which is an ordered list of actions, described in one or two sentences, necessary to address the intent given a list of available tools. For each tool, the name, a description, and the list of arguments that can be used to call it are provided. The procedure is a more structured version of the intent, covering all the necessary steps to achieve the desired outcome. The inputs provided to the LLM to generate the procedure are:

- the previously generated intent
- the list of available tools

The prompt used explicitly states that the procedure should not contain any vague statements, conditionals are allowed but must be resolved within the procedure, and procedure steps may contain actions solely based on the list of available tools.

4.4 Flowgraph Generation

From the procedure it is now possible to produce a flowgraph, which provides a structure of the flow and better encapsulates the conditionals in the procedure. The flowgraph is composed of nodes, which represent the agent messages or actions, and edges, which represent the user responses. Each node has a unique `id`, a `type` and a `description`. The nodes can be of four types:

- `start_message`: the initial message sent by the agent
- `message`: any other message sent by the agent
- `api`: any API call performed by the agent
- `end_message`: the final message by the agent ending the interaction

The edges also have a unique `id`, a `parent_node_id`, a `child_node_id` and a `description`, which contains the user response in a narrative format (e.g., *"The user provides a location"*).

In the prompt we enforce the flowgraph to have only one root node of type `start_message`, and that it should have no incoming edges, and to add in the messages all the details available in the procedure. As presented also in the original paper, we build the graph using a graph library, and we use it to check the construction conditions mentioned above; if they are not met, we retry the generation up to three times, if they all fail, we discard the graph. We also use one-shot prompting by providing an example flowgraph in the prompt, since this is a well-established technique to improve the output [2].

To better enforce the output structure of the LLM, we used the structured output functionality provided by Langchain, which, given a Zod schema definition, injects the corresponding JSON schema in the prompt and parses the LLM response checking for any misalignment. This approach is considered by industry experts as very effective for ensuring valid JSON objects generation [12], and it will be used in every subsequent step that involves the use of an LLM. However, in some situations this may not be a sufficiently robust strategy, thus in Section 6.2 we discuss how to further improve the output consistency.

4.5 Conversation Graph Generator

The goal of this step is to convert the flowgraph, which only represents the conversation from the agent's perspective, into an actual conversation, where every message, both from the agent and the user is a node. In the conversation graph there are three types of nodes:

- `assistant`: containing the messages sent by the assistant
- `user`: containing the messages sent by the user
- `api`: containing the API call performed by the assistant

Every node has an `id`, a `type`, and a `description`, which is either a message or an API call. Edges have an `id`, a `parent_node_id`, a `child_node_id`, and a `description`, which contains the API response or is left empty when no description is needed.

We use one-shot prompting here as well, providing an example of a flowgraph and the corresponding conversation graph. We also use the prompt to enforce other construction rules to prevent malformed graphs, and, similarly to the previous step, we use a graph library to check condition such as the root node being of type `assistant` and having no incoming edges, all the leaf nodes being of type `assistant`, and the edges being correctly connected to the nodes. If any of these conditions are not met, we retry the generation up to three times, and if they all fail, we discard the graph.

4.6 Noise Injector

To produce more realistic scenarios covering less straightforward paths, the pipeline includes a noise injection step, where the conversation graph is extended with nodes going outside what is usually called the *"happy path"*. The paper by Arcadinho et al. (2024) identifies only two categories of noise: `out-of-procedure` and `attack`. To address sub-question **SQ2**, we decided to expand to three categories of noise, namely:

- `out-of-procedure`: a user request that goes outside the expected procedure
- `malicious`: a user request that is intended to cause confusion or disruption
- `misunderstanding`: a user request that shows a misunderstanding of the conversation context or the agent's capabilities

To generate the noise we provide to the LLM the past interactions in the conversation and the category of the noise, asking it to generate a user message matching the two inputs. Then we walk through the agent nodes and with a certain probability we insert an edge pointing to the noisy node, and another edge from the noisy node to an assistant node with a description stating that the assistant is available to help with the user's original request.

4.7 Path Sampler

From the noised conversation graph it is now possible to extract different paths traversing it. A path is represented as an array of node ids. Following the pseudocode provided in the original paper, we use an improved random-walk algorithm to traverse the conversation. Using a set of weights that are adjusted after every iteration, the goal is to increase the coverage by reducing the chance of visiting the same node multiple times. From each graph we will extract a given number of paths M provided as input. After the paths are extracted, we look for duplicates and remove them, since they do not provide any additional information.

4.8 Conversation Generator

With the available paths it is now possible to generate the actual conversations between the agent and the user. We use the ids in the path to retrieve the corresponding nodes and edges connections from the conversation graph, and we provide this and the list of available tools to the LLM. Each entry in the generated conversation is structured in the following way:

- a `role`: which can be one of `user`, `assistant`, `api` or `api_output`
- the `content`: representing the actual message

We instruct the LLM with some constraints to make sure the entries of type `api` should contain the API call, while the `api_output` entries should contain the output returned by the API.

For this step we use one-shot prompting as we did in previous steps, by providing to the LLM an example path, an example list of APIs and the expected conversation to be generated. We also define in the prompt other constraints like always generating an `api_output` entry after an `api` one, and an `assistant` entry after an `api_output`, to ensure the validity of the generated conversations.

4.9 Conversation Annotation

This step is the conjunction between the two main papers used for this research. The current conversation, which will be used as ground truth, only contains a `role` and a `content`. In order to evaluate the conversations using the metrics provided by T-Eval, we need to have a ground truth that follows the structure defined in their paper, which requires that for every query q we need a tuple (t, a, o, r) where:

- t is the thought preceding the decision of an action
- a is the action to be performed, which is the tool call
- o is the observation, meaning the result of the tool call
- r is the review of the result of the tool call, constrained to one of five categories: Success, Internal Error, Input Error, Irrelevant Response, and Unable to Accomplish.

Since the T-Eval annotation process was initially designed for single queries, we first need to break down the conversation into a list of interactions. This process corresponds to the last step described in the paper by Arcadinho et al. [1], and we decided to combine it with the annotation step since it is a very simple operation. A function loops over the conversation entries and aggregates the `content` values in an object with a key for each role. Since every interaction starts with a user message, when the next user message is found, the current object is added to the interactions list.

To generate the thought t, we provide the LLM the complete conversation history as context, plus the relevant user message and the following `api`, `api_output` and `assistant` messages, asking to produce a compatible thought explaining the decision to call the API. By providing the correct tool call in advance, we ensure that the thought serves as valid ground truth that fits the conversation flow. To generate the review r, we provide the same information as before plus the observation, which is obtained from the `api_output`. In the prompt, we ask the LLM to evaluate the quality and usefulness of the obtained observation, and return one of the five categories mentioned above according to their definitions.

The evaluation metrics by Chen et al. [3] are focused on queries that require a tool call to answer. However, in a real conversation like the one we generated, some interactions do not involve a tool call but only a response to ask the user more details. For such interactions, we cannot define a valid action, observation, and review since no tool call is needed. Nonetheless, it is still meaningful to generate a thought where the agent makes the decision to not call any tool and directly reply instead, to be then able to evaluate if the agent has enough awareness to make such a decision. We also generate a review that can be either *"Success"* or *"Input Error"*, depending on whether the decision not to call a tool is justified by the user request.

4.10 Prediction Generation

To generate the predictions, it is necessary to assemble an agent with the user-defined prompt and list of tools, the user query, and we add some additional information in the prompt to ensure that the output can be then parsed by T-Eval evaluator functions, which can accept either a string or stringified JSON as input. Since the purpose of this research is to test the agent's behaviour in a realistic scenario, we decided to limit the input to string type, asking the LLM to produce a thought containing the reasoning and tool calls in one sentence.

After the thought is generated, we can parse it to extract the tool calls to be able to provide the observation, which corresponds to the tool output. If the extracted call is valid, we can provide the resulting observation and then ask the LLM to generate a review. To

Table 1. Assistant A Results

Conversation	Correct Paths	Planning	Reasoning	Retrieval	Understanding	Instruction	Review	Mean
1	1/2	0.850	0.784	0.845	0.775	0.882	0.793	0.822
2	1/2	0.867	0.810	0.883	0.667	0.905	0.814	0.759
3	0/2	0.750	0.696	0.791	0.750	0.836	0.708	0.755
Overall	2/6	0.822	0.763	0.840	0.731	0.874	0.772	0.800

Table 2. Assistant B Results

Conversation	Correct Paths	Planning	Reasoning	Retrieval	Understanding	Instruction	Review	Mean
1	0/2	0.675	0.532	0.675	0.667	0.781	0.738	0.678
2	0/2	0.667	0.643	0.667	0.565	0.750	0.833	0.688
3	0/2	0.667	0.686	0.750	0.576	0.826	0.708	0.702
Overall	0/6	0.669	0.620	0.697	0.603	0.786	0.760	0.689

do so, we provide in the prompt what was mentioned before, plus the newly generated thought, and the observation, asking to return one of the five review categories mentioned in Section 4.9.

However, if the extracted call mentions a non-existent tool, we insert a fixed observation stating that the tool called does not exist to still produce a valid review for evaluation. It can also happen that the call has incorrect arguments; in this case, we still call the tool and provide the resulting output that contains the error message, to see if the review provided identifies the mistake in assembling the call.

Since the next step is to perform the evaluation of the test results using T-Eval, which is implemented in Python, we added to this class an export method that fits the input format expected by T-Eval evaluators. To maintain a clear organization, we create a directory for each conversation, containing a directory for each path. Inside this directory we store four JSON files, one for each evaluator, since each one requires a slightly different input format.

4.11 Evaluation Execution

At this point we have all the necessary data to run the T-Eval evaluators. To do so, we created a bash script that uses the exported data as input and runs individually the four evaluators. When execution is completed, it checks that the files containing the results exist; if not, it lists in the shell the failed ones. To avoid losing this information it also creates a summary file containing this information.

The output files are also arranged in the same structure as the input files, with a directory for each conversation and a subdirectory for each path. This allows us to easily aggregate the results in the next step.

4.12 Metrics Aggregation

The results provided by the evaluators are generated per individual interaction, since T-Eval is focused on single query-answer pairs. However, our goal is to test a complete conversation with different paths; thus, to make the results meaningful we need to aggregate them to provide a broader overview.

We denote a conversational path by p, a conversation by c, and index our six evaluation dimensions by $d = 1, \ldots, 6$.

(1) **Path-level aggregation.** Within each path p, we average all evaluator ratings along each dimension d. The result is a single score $\bar{x}_{p,d}$ for path p on dimension d.
(2) **Conversation-level aggregation.** For a given conversation c, we gather its constituent paths and compute the mean of their $\bar{x}_{p,d}$ values for each dimension. This gives conversation-level scores $\bar{x}_{c,d}$. To condense these into one overall conversation score, we then take

$$\bar{x}_c \;=\; \frac{1}{6}\sum_{d=1}^{6} \bar{x}_{c,d}.$$

(3) **Overall aggregation.** Finally, we average the conversation-level metrics $\bar{x}_{c,d}$ across all conversations c to obtain each dimension's corpus-level mean, and average $\bar{x}_c$ to produce the grand mean.

Since the scope of the research is to analyze the assistant behaviour in a complete conversation, we decided to introduce another metric capturing if a path had all successful interactions. This value is boolean and is calculated by comparing each interaction metric with the maximum value (complete correctness). If they coincide for every interaction in the path, the path is considered correct. We then use this boolean value to calculate the fraction of correct paths for each conversation.

All the metrics are stored in a JSON file, which is then used to generate a final report in markdown, including the most relevant evaluation results presented in a table format and a more detailed part below it.

An example of the table results of this aggregation process can be seen in Table 1.

Below the table we export the individual results for each path, to facilitate a more granular analysis of the agent's performance. Here we include the boolean values indicating whether the path had all successful interactions, the six evaluation dimensions, and a boolean indicating if the path included noise nodes. An example of the individual results can be seen in Appendix C.

5 VALIDATION

To test the pipeline, we designed two assistants with the same scope but with different tools description, in particular one of them has more vague descriptions that can lead to ambiguity in their usage. The reason behind this validation strategy is that the assistant with lower quality tool description should have worse performance than the other assistant, and this should be reflected in the metrics. The tool descriptions of the two assistants can be found in Appendix D. The assistant with better tool description is referred to as `assistant_a` and the second one `assistant_b`.

We used the same prompt and the same set of tools for both assistants, to ensure that the only difference in the evaluation is the quality of the tool descriptions. To generate the complete set of tests, we used the `claude-3.7-sonnet` model from Anthropic. The results for each assistant aggregated in table format are shown in Table 1 and Table 2.

6 DISCUSSION

The results of the validation presented in the previous section show that the evaluation results for the two assistants are significantly different, with `assistant_a` achieving a higher overall correctness compared to `assistant_b`.

In particular, if we look at the `Correct Paths` column, we can see that `assistant_a` has two conversations with one correct path each, while `assistant_b` has no correct paths in any conversation. This is consistent with the expectation that the assistant with better tool descriptions should have better performance, since the tool descriptions are more precise.

We can also observe that the overall value of the planning column, which corresponds to the correctness of the tool selection, for assistant A is above the overall average, while assistant B is below this value. This difference can be interpreted as a sign that assistant A is better at selecting the appropriate tools for the task at hand, given the availability of more precise tool descriptions.

By performing a deeper inspection in the individual path evaluation results, we tried to identify if the noise injected in the conversation graph affected the agent performance. We noticed that for both assistants, the presence of noise in a path led to a decrease in performance. This suggests that the noise-injection strategy is effective in testing the agent's ability to handle unexpected user behavior, by covering a wider range of possible interactions.

While this result is promising, the scale of the testing is relatively small, and the difference in results can be the consequence of the stochastic nature of LLMs. In the following subsection we discuss this and the other limitations of this work more in depth.

6.1 Limitations

The objective of this research is to automatically generate and execute a set of tests for custom tool-calling agents. However, it must be taken into consideration that without any human supervision or review, LLMs can produce incorrect output, undermining the validity of the test results. This limitation can be addressed by allowing the user to manually review the generated conversations and discard the faulty ones, or by introducing additional verification steps performed by dedicated LLMs. Since the first option would compromise the automation aspect of the pipeline, in Section 6.2 we discuss potential LLM-based verification approaches.

One of the necessary inputs for the pipeline is the minimal developer input structured object, which is used to provide the context necessary to generate valid ground truths. Due to the lack of literature on this specific topic, the structured object is based on the feedback of a limited number of experts. This may not be sufficient to successfully identify all the necessary information to generate valid ground truths, potentially leading to a lower quality of the generated conversations.

Another relevant feature of this research is to generate detailed six-dimensional metrics, building on the T-Eval framework by Chen et al. [3]. To produce such detailed analysis, it is necessary to force the agent being tested to generate the output responses following a specific structure. This structure includes explicit planning, tool selection, argument definition, function call composition and output review based on five predefined categories. While most of these elements are very commonly part of an agent output, their enforcement alters the behavior of the agent in the testing environment, making the evaluation results potentially less aligned with real environment behavior.

Some design choices constitute an intrinsic limitation in scope. The decision to use the MCP, for example, requires having an MCP server running to be able to retrieve the list of tools and query them to obtain the necessary observations. Another limitation comes from the decision to generate ground truths assuming the tool response is almost never returning a network error, making it less effective in testing the assistant behavior in this edge case situation. In Section 6.2 we discuss in more depth the further research to address the limitations mentioned.

As mentioned in the introduction to this section, the validation was performed on a small scale, which limits the statistical power of our results. This is due to the limited computational and human resources available for this research. However, this can be addressed in future iterations over this work. In the next subsection we present a list of possible next steps that can improve the scientific relevance of these initial results.

6.2 Future Work

Given the aim to fully automate the testing process, future work should focus on increasing the validation procedures for the output of each step. An example is to improve the efficacy of the current output parsing by handling an error situation with an agent loop, where the error message is sent back to the LLM together with the response, asking it to resolve the issue. This should guarantee a higher success rate.

Given the limited availability of experts, the structured object used to generate the ground truths is based on a small set of interviews. To improve the quality of the generated ground truths, it is necessary to expand the number of expert interviews and to include experts from different sectors, since the current set may not capture the full diversity of perspectives and use cases. This will help further enhance the quality of the generated conversations.

Another area that can be further improved is the noise injection strategy. As highlighted in Section 6.1 some edge cases connected to the tool failure are not properly covered. To address this issue, a simulated error response from the tools related to an internal error can be introduced to test the assistant's ability to recover.

Although the focus of this research is not on the efficiency of the pipeline, it is still important to consider the temporal and computational cost of the various steps. All the intermediate steps aim to produce a valid ground truth, but this creates significant overhead. To address this issue, recent research introduced the concept of *prompt caching*, which allows optimization of the inference by reusing attention states from previous runs [7]. Since conversations exhibit a high degree of contextual overlap across successive turns, prompt caching is particularly convenient in this scenario.

Finally, the evaluation criteria could be expanded, creating additional metrics providing more detailed insights on the specific situation of failure, for example providing a ranking of the most misused tools, or a list of the non-existing tools called.

7 CONCLUSIONS

This research addresses a significant gap in tool-calling agent evaluation by introducing a fully automated pipeline for generating and executing test scenarios for custom tool-calling agents, requiring only minimal developer input and delivering detailed, six-dimensional evaluation metrics. By combining a nuanced conversation generation pipeline with the T-Eval framework, we demonstrate that comprehensive test suites can be produced without manual scenario curation and that they effectively identify weaknesses in agent performance.

To answer sub-question **SQ1** about minimal developer input, we conducted expert consultations that identified four essential components: (1) the agent's operative context and role definition, (2) the complete agent prompt containing business logic, (3) environment assumptions that constrain tool usage, and (4) logical groupings of related tools with their intended purposes. These findings suggest that this structured input allows LLMs to generate contextually

appropriate test scenarios without requiring extensive manual curation, providing a balance between developer effort and test validity.

For sub-question **SQ2** on noise-injection strategies, we extended the existing approach by introducing three distinct perturbation categories: out-of-procedure (deviations from expected workflow), malicious (intentional disruption attempts), and misunderstanding (user confusion about context or capabilities). The results of our initial validation indicate that these expanded noise-injection strategies generate realistic conversation scenarios that effectively cover broader edge cases, demonstrating improved coverage of realistic user behavior variations.

Concerning sub-question **SQ3** about T-Eval's effectiveness for conversation-level evaluation, we effectively integrated the thought-action-observation-review schema into multi-turn dialogues and extended it through metric aggregation at path, conversation, and overall levels. In our validation experiment, T-Eval's six-dimensional scoring proved to provide meaningful and actionable metrics for evaluating complete multi-turn conversations, detecting quality differences between agents with varying tool description clarity. This demonstrates that T-Eval's single-query metrics can be meaningfully aggregated to assess complete conversational flows while also maintaining computational efficiency.

Our initial validation shows that the pipeline can detect differences in tool-description quality by identifying weaker descriptions with lower evaluation scores. While these results are promising, given the limited scale of our validation, they represent a preliminary proof-of-concept that requires further validation. Future work could include more extensive experiments, automated validation mechanisms, prompt caching, and additional evaluation metrics.

REFERENCES

[1] Samuel Arcadinho, David Aparicio, and Mariana Almeida. 2024. Automated test generation to evaluate tool-augmented LLMs as conversational AI agents. arXiv:2409.15934 [cs.CL] https://arxiv.org/abs/2409.15934

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL] https://arxiv.org/abs/2005.14165

[3] Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, and Feng Zhao. 2024. T-Eval: Evaluating the Tool Utilization Capability of Large Language Models Step by Step. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 9510–9529. https://doi.org/10.18653/v1/2024.acl-long.515

[4] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[6] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul R. Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Jack Krawczyk, Cosmo Du, Ed Chi, Heng-Tze Cheng, Eric Ni, Purvi Shah, Patrick Kane, Betty Chan, Manaal Faruqui, Aliaksei Severyn, Hanzhao Lin, YaGuang Li, Yong Cheng, Abe Ittycheriah, Mahdis Mahdieh, Mia Chen, Pei Sun, Dustin Tran, Sumit Bagri, Balaji Lakshminarayanan, Jeremiah Liu, Andras Orban, Fabian Güra, Hao Zhou, Xinying Song, Aurelien Boffy, Harish Ganapathy, Steven Zheng, HyunJeong Choe, Ágoston Weisz, Tao Zhu, Yifeng Lu, Siddharth Gopal, Jarrod Kahn, Maciej Kula, Jeff Pitman, Rushin Shah, Emanuel Taropa, Majd Al Merey, Martin Baeuml, Zhifeng Chen, Laurent El Shafey, Yujing Zhang, Olcan Sercinoglu, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, Alexandre Frechette, Charlotte Smith, Laura Culp, Lev Proleev, Yi Luan, Xi Chen, James Lottes, Nathan Schucher, Federico Lebron, Alban Rrustemi, Natalie Clay, Phil Crone, Tomas Kocisky, Jeffrey Zhao, Bartek Perz, Dian Yu, Heidi Howard, Adam Bloniarz, Jack W. Rae, Han Lu, Laurent Sifre, Marcello Maggioni, Fred Alcober, Dan Garrette, Megan Barnes, Shantanu Thakoor, Jacob Austin, Gabriel Barth-Maron, William Wong, Rishabh Joshi, Rahma Chaabouni, Deeni Fatiha, Arun Ahuja, Gaurav Singh Tomar, Evan Senter, Martin Chadwick, Ilya Kornakov, Nithya Attaluri, Iñaki Iturrate, Ruibo Liu, Yunxuan Li, Sarah Cogan, Jeremy Chen, Chao Jia, Chenjie Gu, Qiao Zhang, Jordan Grimstad, Ale Jakse Hartman, Xavier Garcia, Thanumalayan Sankaranarayana Pillai, Jacob Devlin, Michael Laskin, Diego de Las Casas, Dasha Valter, Connie Tao, Lorenzo Blanco, Adrià Puigdomènech Badia, David Reitter, Mianna Chen, Jenny Brennan, Clara Rivera, Sergey Brin, Shariq Iqbal, Gabriela Surita, Jane Labanowski, Abhi Rao, Stephanie Winkler, Emilio Parisotto, Yiming Gu, Kate Olszewska, Ravi Addanki, Antoine Miech, Annie Louis, Denis Teplyashin, Geoff Brown, Elliot Catt, Jan Balaguer, Jackie Xiang, Pidong Wang, Zoe Ashwood, Anton Briukhov, Albert Webson, Sanjay Ganapathy, Smit Sanghavi, Ajay Kannan, Ming-Wei Chang, Axel Stjerngren, Josip Djolonga, Yuting Sun, Ankur Bapna, Matthew Aitchison, Pedram Pejman, Henryk Michalewski, Tianhe Yu, Cindy Wang, Juliette Love, Junwhan Ahn, Dawn Bloxwich, Kehang Han, Peter Humphreys, Thibault Sellam, James Bradbury, Varun Godbole, Sina Samangooei, Bogdan Damoc, and Alex Kaskasoli. 2023. Gemini: A Family of Highly Capable Multimodal Models. *arXiv e-prints*, Article arXiv:2312.11805 (Dec. 2023), arXiv:2312.11805 pages. https://doi.org/10.48550/arXiv.2312.11805 arXiv:2312.11805 [cs.CL]

[7] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 325–338. https://proceedings.mlsys.org/paper_files/paper/2024/file/a66caa1703fe34705a4368c3014c1966-Paper-Conference.pdf

[8] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. 2025. Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions. arXiv:2503.23278 [cs.CR] https://arxiv.org/abs/2503.23278

[9] Shijue Huang, Wanjun Zhong, Jianqiao Lu, Qi Zhu, Jiahui Gao, Weiwen Liu, Yutai Hou, Xingshan Zeng, Yasheng Wang, Lifeng Shang, Xin Jiang, Ruifeng Xu, and Qun Liu. 2024. Planning, Creation, Usage: Benchmarking LLMs for Comprehensive Tool Utilization in Real-World Complex Scenarios. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 4363–4400. https://doi.org/10.18653/v1/2024.findings-acl.259

[10] 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. arXiv:2310.03128 [cs.SE] https://arxiv.org/abs/2310.03128

[11] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 3102–3116. https://doi.org/10.18653/v1/2023.emnlp-main.187

[12] Michael Xieyang Liu, Frederick Liu, Alexander J. Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J. Cai. 2024. "We Need Structured Output": Towards User-centered Constraints on Large Language Model Output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI EA '24)*. Association for Computing Machinery, New York, NY, USA, Article 10, 9 pages. https://doi.org/10.1145/3613905.3650756

[13] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. 2022. WebGPT: Browser-assisted question-answering with human feedback. arXiv:2112.09332 [cs.CL] https://arxiv.org/abs/2112.09332

[14] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[15] 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. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. arXiv:2307.16789 [cs.AI] https://arxiv.org/abs/2307.16789

[16] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[17] Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Guoqing Du, Shiwei Shi, Hangyu Mao, Ziyue Li, Xingyu Zeng, and Rui Zhao. 2023. TPTU: Large Language Model-based AI Agents for Task Planning and Tool Usage. arXiv:2308.03427 [cs.AI] https://arxiv.org/abs/2308.03427

[18] Yunhao Yang and Anshul Tomar. 2023. On the Planning, Search, and Memorization Capabilities of Large Language Models. arXiv:2309.01868 [cs.CL] https://arxiv.org/abs/2309.01868

[19] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629

[20] Qingchen Yu, Zifan Zheng, Shichao Song, Zhiyu Li, Feiyu Xiong, Bo Tang, and Ding Chen. 2025. xFinder: Large Language Models as Automated Evaluators for Reliable Evaluation. arXiv:2405.11874 [cs.CL] https://arxiv.org/abs/2405.11874

[21] Wenshuo Zhai, Jinzhi Liao, Ziyang Chen, Bolun Su, and Xiang Zhao. 2025. A Survey of Task Planning with Large Language Models. *Intelligent Computing* 4 (2025), 0124. https://doi.org/10.34133/icomputing.0124 arXiv:https://spj.science.org/doi/pdf/10.34133/icomputing.0124

A AI DISCLOSURE

During the preparation of this work, we used ChatGPT, Claude, and GitHub Copilot to support code development: analyzing design pattern alternatives, generating boilerplate implementations, and debugging errors. We also employed these tools to proofread text, identify grammatical inconsistencies, and optimize LaTeX formatting. We used Google Search and Overleaf, which employ AI technologies under the surface, to perform research and review Latex respectively. After using these tools, we thoroughly reviewed and edited the content as needed, taking full responsibility for the final outcome.

B MINIMAL DEVELOPER INPUT

```
{
  // OpenAI-compatible API Configuration (loaded
      from environment)
  llm: {
    baseUrl: Deno.env.get("LLM_BASE_URL"),
    apiKey: Deno.env.get("LLM_API_KEY"),
    model: Deno.env.get("LLM_MODEL"),
    temperature:
        parseFloat(Deno.env.get("LLM_TEMPERATURE")
        || "0.7"),
    maxTokens:
        parseInt(Deno.env.get("LLM_MAX_TOKENS")
        || "2000")
  },

  // MCP Server Configuration (loaded from
      environment)
  mcpServer: {
    url: Deno.env.get("MCP_SERVER_URL"),
```

```
    timeout:
        parseInt(Deno.env.get("MCP_SERVER_TIMEOUT")
        || "30000")
  },

  // Specifications (embedded from specs.yaml)
  specs: {
    context: {
      agent_role: "AGENT_ROLE",
      agent_prompt: "AGENT_PROMPT",
      environment_assumptions: [/* array of
          strings */]
    },
    toolsets: [
      {
        "name":   "TOOLSET_NAME",
        "usage":  "TOOLSET_PURPOSE",
        "tools":  [/* list of tool-identifiers */]
      }
      /* additional toolsets */
    ]
  }
};
```

C INDIVIDUAL PATH RESULTS

```
{
  "intent_id": 1,
  "path_id": 1,
  "is_correct": false
  "normalized_scores": {
    "planning": 0.75,
    "instruction": 0.8125,
    "reasoning": 0.5442041158676147,
    "understanding": 0.75,
    "retrieval": 0.75,
    "review": 0.875
  },
  "is_noisy": true,
}
```

D TOOL DESCRIPTIONS

D.1 Assistant A Tools

```
[
  {
    "tool_name": "add_event",
    "description": "Adds an event to the
        calendar",
    "args": [
      { "name": "title", "type": "string",
          "description": "Title of the event",
          "required": true },
      { "name": "date", "type": "string",
          "description": "Date of the event in
          YYYY-MM-DD format", "required": true },
      { "name": "time", "type": "string",
          "description": "Time of the event in
          HH:MM format", "required": true },
      { "name": "location", "type": "string",
          "description": "Location of the event",
          "required": true },
      { "name": "description", "type": "string",
          "description": "Description of the
          event, which is optional", "required":
          false }
    ]
  },
  {
    "tool_name": "get_events",
    "description": "Gets events from the calendar
        for a specific date and location. It
        returns a list of events with their
        titles, times, duration and IDs.",
    "args": [
      { "name": "date", "type": "string",
          "description": "Date to check events
          for, in YYYY-MM-DD format", "required":
          true },
      { "name": "location", "type": "string",
          "description": "Location to filter
          events by.", "required": false },
      { "name": "limit", "type": "number",
          "description": "Maximum number of
          events to return.", "required": false }
    ]
  },
  {
    "tool_name": "delete_event",
    "description": "Deletes an event from the
        calendar given an ID. It can optionally
        include a reason for deletion.",
    "args": [
      { "name": "eventId", "type": "string",
          "description": "ID of the event to
          delete", "required": true },
      { "name": "reason", "type": "string",
          "description": "Reason for deleting the
          event.", "required": false }
    ]
  },
  {
    "tool_name": "get_opening_hours_by_location",
    "description": "Checks the opening hours for
        a specific location. The location must be
        provided.",
    "args": [
      {
        "name": "location",
        "type": "string",
        "description": "Specific location to
            check opening hours for. Provide city
            or address, the system will identify
            the closest office.",
        "required": true
      }
    ]
  }
]
```

D.2 Assistant B Tools

```
[
{
  "tool_name": "add_event",
  "description": "Adds an event to the
      calendar",
  "args": [
    { "name": "title", "type": "string",
        "description": "Title of the event",
        "required": true },
    { "name": "date", "type": "string",
        "description": "Date of the event",
        "required": true },
    { "name": "time", "type": "string",
        "description": "Time of the event",
        "required": true },
    { "name": "location", "type": "string",
        "description": "Location of the event",
        "required": true },
    { "name": "description", "type": "string",
        "description": "Description of the
        event", "required": false }
  ]
},
{
  "tool_name": "get_events",
  "description": "Gets events from the
      calendar.",
  "args": [
    { "name": "date", "type": "string",
        "description": "Date to check events
        for", "required": true },
    { "name": "location", "type": "string",
        "description": "Location to filter
        events by.", "required": false },
    { "name": "limit", "type": "number",
        "description": "Maximum number of
        events", "required": false }
  ]
},
{
  "tool_name": "delete_event",
  "description": "Deletes an event from the
      calendar.",
  "args": [
    { "name": "eventId", "type": "string",
        "description": "ID of the event",
        "required": true },
    { "name": "reason", "type": "string",
        "description": "Reason for deletion",
        "required": false }
  ]
},
{
  "tool_name": "get_opening_hours_by_location",
  "description": "Checks the opening hours.",
  "args": [
    {
      "name": "location",
      "type": "string",
      "description": "Specific location to
          check opening hours for.",
      "required": true
    }
  ]
}
]
```