# TOOL-PLANNER: TASK PLANNING WITH CLUSTERS ACROSS MULTIPLE TOOLS

**Yanming Liu[1], Xinyue Peng[2], Jiannan Cao[3], Shi Bo, Yuwei Zhang, Xuhong Zhang[1]*, Sheng Cheng[1], Xun Wang[1], Jianwei Yin[1], Tianyu Du[1]***
[1]Zhejiang University, [2]Southeast University
[3]Massachusetts Institute of Technology
{oceann24, zhangxuhong, zjradty}@zju.edu.cn, zjuyjw@cs.zju.edu.cn, jiannan@mit.edu, xinyuepeng@seu.edu.cn, yuweizhang@tongji.edu.cn

## ABSTRACT

Large language models (LLMs) have demonstrated exceptional reasoning capabilities, enabling them to solve various complex problems. Recently, this ability has been applied to the paradigm of tool learning. Tool learning involves providing examples of tool usage and their corresponding functions, allowing LLMs to formulate plans and demonstrate the process of invoking and executing each tool. LLMs can address tasks that they cannot complete independently, thereby enhancing their potential across different tasks. However, this approach faces two key challenges. First, redundant error correction leads to unstable planning and long execution time. Additionally, designing a correct plan among multiple tools is also a challenge in tool learning. To address these issues, we propose Tool-Planner, a task-processing framework based on toolkits. Tool-Planner groups tools based on the API functions with the same function into a toolkit and allows LLMs to implement planning across the various toolkits. When a tool error occurs, the language model can reselect and adjust tools based on the toolkit. Experiments show that our approach demonstrates a high pass and win rate across different datasets and optimizes the planning scheme for tool learning in models such as GPT-4 and Claude 3, showcasing the potential of our method. Our code is public at https://github.com/OceannTwT/Tool-Planner.

## 1 INTRODUCTION

Large Language Models (LLMs) (Brown et al., 2020; Chowdhery et al., 2023; Touvron et al., 2023; Zeng et al., 2023; Zhang et al., 2022) have demonstrated outstanding performance across multiple domains, leveraging parameterized knowledge to exhibit powerful reasoning and planning capabilities (Bang et al., 2023). Tool learning (Schick et al., 2023; Yao et al., 2022; Lu et al., 2024; Qin et al., 2024) harnesses the planning prowess of LLMs by decomposing complex problems through understanding tools and having LLMs generate plans for tasks, thus leveraging external tools (APIs) to handle intermediate steps, achieving the goals of completing complex tasks. By utilizing tools, LLMs can significantly improve limitations in certain tasks, such as enhancing accuracy in mathematical reasoning problems (Hao et al., 2023b; Lee et al., 2024; Gou et al., 2024b), answering up-to-date news queries (Song et al., 2023; Liu et al., 2024b), executing commands, or invoking other models (Shen et al., 2024). Consequently, tool learning has become one of the potential paradigms for solving complex real-world scenarios.

The existing study of tool learning focuses on two main crucial aspects: **(1) how to better utilize tools for task planning and execution**, and **(2) how to adjust planning based on the results of tool invocation.** During the process of invoking tools, chain-like invocations may encounter situations of failure (Qin et al., 2024). This necessitates timely adjustments of tools and task re-planning when encountering tool errors. DFSDT (Qin et al., 2024) generates new plans by searching states when encountering API call errors, thus solving the original problem with tools according to new reasoning paths. ToolChain* (Zhuang et al., 2023) employs heuristic search algorithms to choose the direction

[*]Corresponding author.

| Feature | **Tool-Planner**<br>(this work) | **ReAct**<br>(Yao et al., 2022) | **Reflexion**<br>(Shinn et al., 2023) | **AdaPlanner**<br>(Sun et al., 2023a) | **DFSDT**<br>(Qin et al., 2024) | **Toolchain***<br>(Zhuang et al., 2023) |
|---|---|---|---|---|---|---|
| Tool Calling | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Tool Refinement | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Tool Expansion | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| Explainable Planning | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ |
| Tool Integration | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Task Nodes Size | ↘ | ↗ | ↗ | ↗ | ↗ | ↗ |

Table 1. A comparison of our **Tool-Planner** to notable planning and tool interaction frameworks. Our method shows significant advantages in real-world tool integration, multi-tool support, and efficient solution sizes.

most likely to yield answers during the planning search process. Tree-Planner (Hu et al., 2023) presets multiple planning paths and merges nodes with the same preceding tool for deep search. These methods all involve adjustments and optimizations of search algorithms based on tree structures. However, when a tool is called and an error occurs, previous methods typically choose to discard that planning path directly, despite the potential existence of other tools offering similar or identical functionality to accomplish that task. Furthermore, the new planning methods proposed by LLMs are generally more complex, leading to a higher likelihood of tool errors on new planning paths. Methods like CRITIC (Gou et al., 2024a) and AnyTool (Du et al., 2024) integrate feedback mechanisms into tool learning and provide error messages to aid in task re-planning. Nonetheless, the integration of this information still faces issues of inefficiency while lacking effective exploration of the current planning path. Multiple re-planning cycles also result in inefficient task resolution. Therefore, in tool learning, the rational planning of tasks and tools becomes increasingly important.

To address these challenges, we propose Tool-Planner, an efficient framework for task planning and tool invocation in tool learning with LLMs. Tool-Planner conceptualizes the problem-solving process as a decision tree. Unlike the previous methods, Tool-Planner views each node as a set of tools rather than a single tool, when a tool invocation error occurs, it prioritizes solutions within the same toolkit. Tool-Planner effectively tackles the inefficiency issue of previous methods in utilizing task planning solutions and significantly improves the pass rate of tool learning in task resolution. We construct sets of similar tools utilizing the SimCSE (Gao et al., 2021) to evaluate the distance of different tools for tool clustering based on tool APIs document and description. We conduct extensive experiments on two different datasets: ToolBench (Qin et al., 2024), which uses APIs selected from RapidAPI Hub (Rapid, 2023), and APIBench (Patil et al., 2023), which fetches APIs from various open-source models. Compared to various prior tool-learning search methods, Tool-Planner achieves a +8.8% increase in pass rate and +9.1% increase in win rate on ToolBench, as well as a +6.6% increase in pass rate and +14.5% increase in win rate on APIBench when tested with GPT-4. It also demonstrates outstanding performance in terms of re-planning frequency and computational speed. Extensive experimental results highlight the advancements of Tool-Planner.

**Our Contributions.** Our main contributions are summarized as follows.

- We propose a novel framework, Tool-Planner, which integrates external tools with LLMs. This framework enables task planning and tool invocation based on toolkits, addressing the inefficiencies in planning found in previous approaches.
- Tool-Planner categorizes tools into toolkits with similar or identical functionalities by clustering tool embeddings generated by SimCSE. The setting of toolkits allows thorough exploration of tools along a planning path, ensuring that each node is fully utilized and maximizing the information from that path.
- Extensive experimental results demonstrate the effectiveness of the Tool-Planner framework, highlighting the importance of thorough tool planning exploration on a single path compared to DFSDT's multiple attempts at different planning paths, and providing an ideal paradigm for future tool learning solution space search methods.

## 2 Preliminaries

**LLMs Reasoning with Tools.** Given a task $x$ input in natural language and a pre-trained model $\rho_\theta(x)$, the naive generation output of an LLM is $y \sim \rho_\theta(x)$, corresponding to the answer predicted by the model. In the context of tool learning with LLMs (Lu et al., 2024; Qin et al., 2024), given the

API documentation (or demos) for tools $\mathcal{D} = \{d_i\}_{i=1}^{N}$ and their API descriptions $\mathcal{M} = \{m_i\}_{i=1}^{N}$, we first generate a multi-step decision-making plan for the tools $\mathcal{P}_t = \{p_1, p_2, \ldots, p_N\} \sim \rho_\theta(x, \mathcal{D}, \mathcal{M})$, where $N$ is the number of tools in the plan. Thus, for each intermediate reasoning step $x_i$, it can be generated through a series of corresponding tool calls. Setting the result of the API function call for each tool as $c_i = F_i(p_i)$, we have $x_i = \rho_\theta(x_i|\{x_h\}_{h=1}^{i}, \{c_{\mathcal{T}_h}\}_{h=1}^{i-1}, \mathcal{D}, \mathcal{M}, x)$. Consequently, the final output result or behavior for the task can be expressed as $y = \rho_\theta(y|\{x_h\}_{h=1}^{K}, \{c_{\mathcal{T}_h}\}_{h=1}^{K}, \mathcal{D}, \mathcal{M}, x)$. Such a process of tool invocation significantly enhances the adaptability of LLMs to various tasks.

**Tree Search on Planning Space.** In the chain-like calls of tool learning (Yao et al., 2022), the use of tools is linear. When encountering the hallucination problem of LLMs leading to repeated API calls and parameter errors, or when the tool itself is unavailable, the exploration of the solution space $\mathcal{S}$ is insufficient. This requires us to adjust the planned path $\mathcal{P}_t$ accordingly. The exploration of the planned path can be seen as a behavior tree, formalized as $G(c) = (V, \mathcal{E})$. Whenever re-planning is needed, all previous plan paths $\mathcal{P}$ and the original plan are used to generate a new plan tree chain $\mathcal{P}_{t'} = \{p'_h\}_{h=1}^{K} \sim \rho_\theta(x, \mathcal{D}, \mathcal{M}, \mathcal{P}, E)$ based on the returned error information $E = \sum\{F_e(p_e)\}$, and priority is given to finding the solution states that can be reached after correcting the current state. However, such multiple iterative calls and exploration of the solution space may repeatedly encounter hallucinations and tool issues, leading to excessively high costs in exploring solutions.

## 3 Methodology

When LLMs utilize various tools to generate answers, inefficiencies often emerge from a suboptimal selection process among multiple tools and frequent alterations in problem-solving strategies. To address this issue, we propose the Tool-Planner framework, aiming at enhancing the efficiency of tool calls while maintaining relative consistency in planning. Our framework includes clustering and categorizing tools, as well as planning processes for tool invocation paths.

### 3.1 Tool Clustering

In the realm of tool learning, comprehending the functionalities of application programming interfaces (APIs) is essential. However, the functionalities of most APIs are often inadequately categorized. For instance, platforms like RapidAPI group APIs with different functions into the same general category without categorizing them specifically for particular tasks. Consequently, problem-solving may require searching through multiple paths and invoking similar functionalities from different categories, leading to increased complexity and higher error rates in implementation.

To tackle this issue, we need to categorize a large number of tools into multiple classes $\mathcal{T}_i$ with similar or identical functionalities, providing functionality explanations for each class, as shown in Figure 1. Through annotated categorization, each class is assigned to a specific functionality $f(i)$, we can accurately differentiate APIs with different functionalities, enabling the selection of alternative APIs from the same category if the chosen API fails.

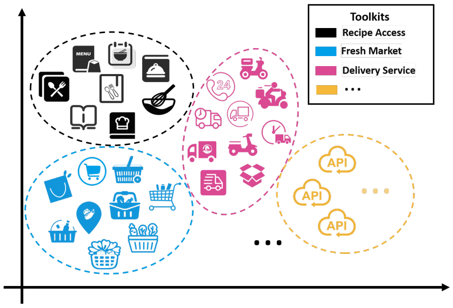

Figure 1: The process of tool clustering.

However, as the number of APIs may continue to increase, annotating each API is costly, and accurately identifying categories from multiple classes poses a challenge. It has been proven effective to use LLMs to guide text and assist in clustering different texts (Viswanathan et al., 2024; Tipirneni et al., 2024; Zhang et al., 2023). Therefore, we develop an automated classification method. We extracted tool documentation $\mathcal{D}$ and descriptions $\mathcal{M}$ of candidate APIs and provided them to LLMs to generate brief explanations $H = \{h_i\}_{i=1}^{N}$ of their functionalities. Upon obtaining API explanations, we utilize the SimCSE (Gao et al., 2021) model to compute text embeddings for these explanations, which can be seen as tool embeddings $e_i$ for the API. To understand which tools might have similar functionalities, we need to classify tool embeddings.

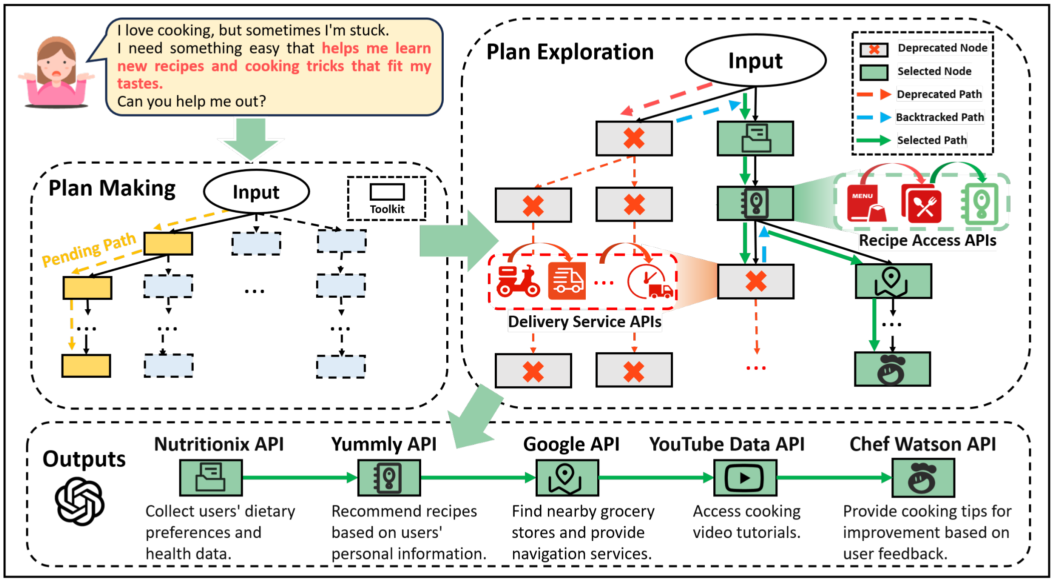
I love cooking, but sometimes I'm stuck. I need something easy that helps me learn new recipes and cooking tricks that fit my tastes. Can you help me out?
Plan Making
Input
Toolkit
Pending Path
Plan Exploration
Input
Deprecated Node
Selected Node
Deprecated Path
Backtracked Path
Selected Path
Recipe Access APIs
MENU
Delivery Service APIs
Outputs
Nutritionix API
Yummly API
Google API
YouTube Data API
Chef Watson API
Collect users' dietary preferences and health data.
Recommend recipes based on users' personal information.
Find nearby grocery stores and provide navigation services.
Access cooking video tutorials.
Provide cooking tips for improvement based on user feedback.

Figure 2: Overview of Tool-Planner. The user inputs a problem, the LLM devises a plan with API functionalities as nodes. The plan is then explored step by step. If an error occurs, other APIs within the same toolkit are tried. If all APIs in the toolkit fail, the process moves to other nodes until the final plan is determined and output.

After obtaining tool embeddings, we employ the $k$-means++ (Arthur & Vassilvitskii, 2007) algorithm to find a $k$-partition of these tool embeddings and generate tool clusters in the solution space. This could formulated as an optimization problem:

$$\arg\min_{\mathcal{T}} \sum_{i=1}^{k} \sum_{e\in\mathcal{T}_i} \left\| e - \frac{1}{|\mathcal{T}_i|} \sum_{x\in\mathcal{T}_i} x \right\|_2^2 . \tag{1}$$

The number of clusters is adjusted through the value of $k$. Each API is assigned to a tool cluster with similar or identical functionalities, even if they are not in the same tool within RapidAPI. Similar functionalities enable rapid tool adjustment when addressing issues. We refer to a cluster as a toolkit, meaning that one toolkit can perform a specific task.

### 3.2 Task Planning

Given task input $x$, LLMs first provide a plan $\mathcal{P}$ for the task. This approach differs from previous methods that directly presented API documentation to LLMs, leading to excessively lengthy contexts. In contrast, Tool-Planner first utilizes in context learning (Brown et al., 2020) to generate brief explanations for all API functionalities within the toolkit, and then uses these explanations to generate the functionality descriptions of toolkit $\mathcal{T}_h$. Consequently, each toolkit is associated with a unique functionality description, denoted as $m_{\mathcal{T}_h} = f(\mathcal{T}_h)$. When planning tasks, we provide these toolkit functionality descriptions as context to LLM and let it design a plan $\mathcal{P}_{\mathcal{T}} = \{P_h\}_{h=1}^{N} \sim \rho_\theta(x, \{m_{\mathcal{T}_h}\}_{h=1}^{k})$ using different toolkits to solve the task. With this prompt, the LLM can generate a chained toolkit-based plan based on the functionality descriptions of the toolkits.

In solving problems for specific states, the model will choose any API within the toolkit for invocation, thus completing the problem-solving for that state and passing the output result to the next state. While the chosen API or tool is $t$, the return of each intermediate toolkit could be stated as $c_{\mathcal{T}_i} = F_t(t)$, we have the intermediate state for instruction:

$$x_i = \rho_\theta(x_i | \{x_h\}_{h=1}^{i-1}, \{c_{\mathcal{T}_h}\}_{h=1}^{i-1}, \{m_{\mathcal{T}_h}\}_{h=1}^{i}, \mathcal{D}_{\mathcal{T}}, x). \tag{2}$$

In this way, when APIs within each toolkit can function properly, this plan can choose any API for each state to complete this process, and provides multiple choices of APIs to complete a state.

### 3.3 Planning Exploration on Solution Space

When LLMs experience hallucinations resulting in problematic parameter information or tool unavailability, we need to replan to complete the original task. After categorizing various tools, since each node in our search plan represents a toolkit containing multiple available APIs, we can adjust the tools through the following adaptation. The behavior tree in the toolkit plan could be formalized as $G(\mathcal{T}) = (V_\mathcal{T}, \mathcal{E}_\mathcal{T})$.

**Task Planning Within Same Toolkits.** When the current API $t$ within a node $V_\mathcal{T}$ becomes unusable for any reason, we prioritize selecting another available API $t'$ within the same toolkit. By referring to the API documentation $d_{t'}$, we can generate the call parameters $param$ for the new API, and fetch the calling result $c'_{\mathcal{T}_h} = F_t(param)$. This allows us to complete the current state by selecting an alternative tool within the same toolkit, without altering the original task plan. In other words, keep $\mathcal{E}_\mathcal{T}$ unchanged. The results generated by the new API can then be used as input for the next state in the task plan, maintaining the relative stability of the plan. The intermediate state for instruction is:

$$x_i = \rho_\theta(x_i | \{x_h\}_{h=1}^{i-1}, \{c_{\mathcal{T}_h}\}_{h=1}^{i-1}, \{c'_{\mathcal{T}_i}\}, \{m_{\mathcal{T}_h}\}_{h=1}^{i}, \mathcal{D}_\mathcal{T}, x). \tag{3}$$

**Task Replanning Across Toolkits.** Given the limited number of identical or similar APIs within a toolkit, if all APIs fail to process the current state $v$, we provide all error information $E = \sum_{t \in \mathcal{T}_i} \{F_t(t_e)\}$ and the original task plan $\mathcal{P}_\mathcal{T}$ to the LLMs, instructing them to generate a new task plan $\mathcal{P}'_\mathcal{T}$ on subgraph $G(\mathcal{T}) \leftarrow G(\mathcal{T}) \setminus v$. The new task plan aims to retain as many results from the previous state as possible. This process is similar to DFSDT (Qin et al., 2024), but while DFSDT searches at the API level, we search at the toolkit level. Once a new plan is generated, we select APIs within the new toolkit and attempt to complete the new state $v'$ according to the new toolkit's APIs. In this way, we can switch to a new plan when the original plan completely fails to accomplish the task, while maintaining relative small times to generate a new plan.

$$\mathcal{P}'_\mathcal{T} = \{P'_h\}_{h=1}^{K} \sim \rho_\theta(x, \mathcal{D}, \{m_{\mathcal{T}_h}\}_{h=1}^{K}, \mathcal{P}_\mathcal{T}, E), \text{on } G(\mathcal{T}) \leftarrow G(\mathcal{T}) \setminus v. \tag{4}$$

By setting these two possibilities, we can greatly optimize the task planning process, thereby improving the pass rate and effectiveness of task completion. We demonstrate more details on Appendix C.

## 4 Experiment

To evaluate the capabilities of our framework, we conducted a series of experiments and assessments, demonstrating the superiority of our approach from various aspects.

### 4.1 Experiment Setup

**Dataset.** We utilize the ToolBench (Qin et al., 2024) and APIBench(Patil et al., 2023) as our experimental dataset. ToolBench comprises 16,464 APIs, which are categorized into different tools and categories. In ToolBench, there are three different datasets for prompt generation, namely G1, G2, and G3, which represent single-tool instructions, intra-category multi-tool instructions, and intra-collection multi-tool instructions, respectively. In APIBench, datasets are constructed by selecting corresponding 1,645 APIs and their descriptions from three platforms as tools, and a series of questions are used to evaluate their performance. More details are described in Appendix B. We use the API interfaces selected by ToolBench and APIBench along with their corresponding documentation and descriptions to extract and generate functional explanations of the APIs. Subsequently, we generate tool embeddings based on their functionalities $\{m_{\mathcal{T}_h}\}_{h=1}^{k}$.

**Evaluation Metrics.** For the ToolBench dataset, we adopt two metrics from ToolEval (Qin et al., 2024) to evaluate our framework, covering different aspects of the task. The first metric is **Pass Rate**, calculated based on the proportion of tasks successfully completed. The second metric is **Win Rate**, where we compare the solution generated by our method with the plan generated by GPT-3.5+ReACT and make LLMs judge which solution is better. When our framework performs better, we mark it as a win. If our framework is the same or inferior to the GPT3.5+ReACT solution, we mark it as a tie or loss. The win rate reflects the quality of our generated solutions and their ability to solve problems. We also evaluate APIBench in **Halluation Rate**. Since most of the call processes in APIBench involve only a few tools, we can make comparison on the hallucination situation of the reasoning process.

Table 2. The results of different models and baselines of Pass Rate (%), Win Rate (%), and Halluation Rate (%). We evaluate various methods on three sub-dataset with their APIs derived from APIBench(Patil et al., 2023).

| **Model** | **Method** | **TorchHub** | | | **HuggingFace** | | | **TensorFlow** | | | **Average** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Pass.(↑) | Win.(↑) | Hallu.(↓) | Pass.(↑) | Win.(↑) | Hallu.(↓) | Pass.(↑) | Win.(↑) | Hallu.(↓) | Pass.(↑) | Win.(↑) | Hallu.(↓) |
| GPT-3.5 | ReACT | 62.6 | - | 23.2 | 35.6 | - | 31.6 | 53.7 | - | 13.6 | 50.6 | - | 22.8 |
| | Reflexion | 70.2 | 59.0 | 21.7 | 46.0 | 55.2 | 28.0 | 55.0 | 56.5 | 12.7 | 57.1 | 56.9 | 20.8 |
| | AdaPlanner | 69.5 | 62.1 | 23.7 | 48.7 | 53.4 | 32.7 | 57.2 | 53.6 | 14.2 | 58.5 | 56.4 | 23.5 |
| | DFSDT | 74.5 | 72.0 | 9.6 | 57.2 | 60.7 | 18.2 | 70.9 | 69.6 | **11.2** | 67.5 | 67.4 | 13.0 |
| | Tool-Planner | **78.2** | **77.6** | **8.2** | **66.5** | **75.4** | **13.5** | **72.5** | **72.4** | 12.1 | **72.7** | **75.1** | **11.3** |
| GPT-4 | ReACT | 68.6 | 58.0 | 19.2 | 43.8 | 59.2 | 23.4 | 61.9 | 61.1 | 15.8 | 58.1 | 59.4 | 19.4 |
| | Reflexion | 74.4 | 68.4 | 16.5 | 51.9 | 58.2 | 16.8 | 60.2 | 55.5 | 12.1 | 62.2 | 60.7 | 15.1 |
| | AdaPlanner | 75.3 | 78.5 | 15.9 | 50.2 | 56.8 | 15.3 | 60.1 | 53.9 | 12.7 | 61.8 | 63.1 | 14.6 |
| | DFSDT | 80.3 | 78.1 | 11.2 | 58.4 | 62.0 | 12.3 | 72.1 | 69.0 | 10.9 | 70.3 | 67.7 | 11.5 |
| | Tool-Planner | **83.2** | **87.0** | **7.1** | **70.3** | **77.0** | **10.7** | **77.2** | **82.6** | **10.5** | **76.9** | **82.2** | **9.4** |
| Claude-3 | ReACT | 69.5 | 56.5 | 21.6 | 42.1 | 56.9 | 30.1 | 58.9 | 46.8 | 13.2 | 56.8 | 53.4 | 21.6 |
| | Reflexion | 76.8 | 68.5 | 20.1 | 48.8 | 67.1 | 20.4 | 60.3 | 59.1 | 12.4 | 62.0 | 64.9 | 17.6 |
| | AdaPlanner | 75.2 | 69.8 | 18.4 | 49.5 | 65.6 | 16.5 | 62.7 | 61.3 | 13.8 | 62.5 | 65.6 | 16.2 |
| | DFSDT | 80.2 | 72.6 | 10.0 | 55.8 | 69.2 | 14.9 | **73.4** | 65.0 | 11.5 | 69.8 | 68.9 | 12.1 |
| | Tool-Planner | **81.5** | **83.5** | **8.6** | **68.2** | **75.9** | **12.4** | 72.1 | **74.5** | **11.1** | **73.9** | **78.0** | **10.7** |

**Baselines.** We compare Tool-Planner with the following baseline methods. (1) **ReACT** (Yao et al., 2022) operates by having an LLM execute an action based on the previous state, and then reason based on the result of that action, repeating this process iteratively. This can be considered a linear tool usage process. (2) **Reflexion** (Shinn et al., 2023) introduces a feedback mechanism during decision-making. In tool-learning scenarios, when an error occurs with an intermediate tool, Reflexion searches for the next node based on previous nodes using the error information. (3) **AdaPlanner** (Sun et al., 2023a) is similar to Reflexion, which explicitly corrects an error at a specific point in the path and adjusts by selecting the correct tool. (4) **DFSDT** (Qin et al., 2024) employs a deep search mechanism. Each time an error node is encountered, it provides the model with all previous error paths and information, allowing the model to re-select and re-plan the path, thereby maximizing the expansion of possible solutions.

**Model.** We utilize the following foundational models for model planning generation and tool learning problem-solving: GPT-3.5 (Ouyang et al., 2022) (*gpt-3.5-turbo-0125*), GPT-4 (Achiam et al., 2023) (*gpt-4-turbo-2024-04-09*), and Claude-3 (*claude-3-sonnet*). We use SimCSE (Gao et al., 2021), an effective method for sentence representation learning that leverages contrastive learning to calculate the tool embeddings. We set $k$ as 1800 in ToolBench and $k$ as 65 in APIBench for experiments.

### 4.2 Main Experiment

**For APIBench.** Tool-Planner outperforms other methods across all three API platforms with significant improvements in Pass Rate, Win Rate, and Hallucination Rate. Compared to DFSDT, Tool-Planner increases the pass rate by **+6.6%** and the win rate by **+14.5%**, while reducing the hallucination rate by **-2.1%** on GPT-4. This demonstrates that Tool-Planner's clustering-based tool planning approach better captures the nuances of API and model selection and task execution, leading to more robust and reliable performance.

**Model Invocation and Hallucination.** Tool-Planner significantly reduces the occurrence of hallucinations. This is due to the targeted optimization of the current model's performance during multiple attempts and plan revisions. For plans that fail to invoke the API successfully, the model discards those plans and selects another API that can be called. The plans generated through this approach effectively invoke the corresponding APIs of the model, successfully achieving the intended goals. Models that successfully invoke APIs also gain better interpretation within toolkit.

**For ToolBench.** Tool-Planner achieves state-of-the-art performance on five out of six datasets and demonstrates competitive performance on G1-Cat. Table 2 shows the comparison results between Tool-Planner and other baselines. Compared to the DFSDT, which has the best average performance, our method improves the pass rate by **+8.8%** and the win rate by **+9.1%**. This indicates that by clustering tools and planning on the clustered toolkits, task planning capabilities can be more effectively enhanced.

**Multi-tool instructions tasks.** For tasks that require the cooperation of multiple tools (such as G2 and G3), our method shows remarkable performance improvements. For multi-tool instructions tasks, the pass rate increases by **+8.9%** and the win rate by **+10.6%** on GPT4, which is more notable than

Table 3. The results of different models and baselines of Pass Rate (%) and Win Rate (%). We evaluate the results on six different sub-dataset derived from Toolbench with various methods.

| Model | Method | G1-Inst. | | G1-Tool. | | G1-Cat. | | G2-Inst. | | G2-Cat. | | G3-Inst. | | Avg. | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Pass. | Win. | Pass. | Win. | Pass. | Win. | Pass. | Win. | Pass. | Win. | Pass. | Win. | Pass. | Win. |
| GPT-3.5 | ReACT | 39.5 | - | 40.5 | - | 40.0 | - | 33.5 | - | 30.5 | - | 21.0 | - | 34.2 | - |
| | Reflexion | 45.5 | 55.8 | 55.0 | 58.8 | 53.5 | 55.5 | 58.0 | 56.3 | 66.0 | 57.3 | 55.0 | 61.3 | 55.5 | 57.5 |
| | AdaPlanner | 46.0 | 56.3 | 53.5 | 60.5 | 55.0 | 57.3 | 60.5 | 65.8 | 67.5 | 55.8 | 59.0 | 59.5 | 56.9 | 59.2 |
| | DFSDT | 48.5 | 58.3 | 62.5 | 59.8 | 58.0 | 56.8 | 71.5 | 70.5 | 70.5 | 59.3 | 61.0 | **68.0** | 62.0 | 62.1 |
| | Tool-Planner | **58.5** | **63.3** | **71.0** | **63.5** | **66.0** | **61.3** | **75.5** | **72.3** | **78.0** | **63.5** | **66.0** | 67.5 | **69.2** | **65.2** |
| GPT-4 | ReACT | 50.5 | 59.8 | 46.5 | 59.3 | 51.5 | 63.0 | 64.5 | 62.8 | 67.5 | 58.5 | 42.0 | 73.5 | 53.8 | 62.9 |
| | Reflexion | 49.5 | 60.8 | 56.5 | 65.3 | 68.0 | 68.5 | 74.0 | 71.8 | 67.0 | 55.3 | 55.0 | 76.5 | 61.7 | 66.4 |
| | AdaPlanner | 52.5 | 62.5 | 60.5 | 64.8 | 68.5 | **73.3** | 75.5 | 73.5 | 68.0 | 52.8 | 70.0 | 79.5 | 65.9 | 67.7 |
| | DFSDT | 57.0 | 65.8 | 72.0 | 69.3 | 64.5 | 65.3 | 77.5 | 72.0 | 69.5 | 56.8 | 71.0 | 81.5 | 68.5 | 68.4 |
| | Tool-Planner | **66.0** | **75.5** | **78.5** | **75.8** | **75.0** | 71.8 | **83.5** | **79.8** | **77.5** | **70.3** | **83.0** | **92.0** | **77.3** | **77.5** |
| Claude-3 | ReACT | 48.5 | 58.8 | 47.5 | 57.3 | 50.0 | 62.5 | 63.0 | 61.5 | 62.5 | 54.8 | 38.0 | 68.5 | 51.6 | 60.6 |
| | Reflexion | 51.5 | 62.3 | 58.5 | 65.8 | 55.5 | 64.3 | 73.0 | 72.8 | 67.0 | 54.3 | 61.0 | 73.0 | 61.1 | 65.4 |
| | AdaPlanner | 52.0 | 63.5 | 62.5 | 64.3 | 57.0 | 63.8 | 71.5 | 73.3 | 66.5 | 53.3 | 62.0 | 75.5 | 61.9 | 65.6 |
| | DFSDT | 55.5 | 62.3 | 72.5 | 68.5 | **61.5** | 67.5 | 74.5 | 70.3 | 68.0 | 56.8 | 69.0 | 79.0 | 66.8 | 67.4 |
| | Tool-Planner | **64.0** | **73.8** | **77.0** | **76.3** | 59.5 | **73.8** | **79.5** | **79.3** | **76.5** | **68.3** | **78.0** | **87.5** | **72.4** | **76.5** |

the improvement in single-tool scenarios. This demonstrates that our method is better able to adapt to and leverage the advantages of different toolkits in multi-tool scenarios. The setup of the toolkits ensures that tool coordination in multi-step tasks focuses more on the execution of each step and can find suitable solutions within similar tools when encountering errors, ensuring the relative stability of the plan.

### 4.3 Analysis

**Ablation study on tool clustering.** In Section 3.1, we introduced the method of tool clustering, which is applied during the phase of generating plans by the model. At each planning step, we obtain a variety of APIs with similar functionalities through clustering as alternatives. To validate the effectiveness of tool clustering in the Tool-Planner, we combine the baselines ReACT and AdaPlanner with the toolkit obtained through clustering, and conducted experiments on GPT-4. Subsequently, we calculated their pass rate and win rate, with detailed results shown in Table 4.

Table 4. Results of Pass Rate (%) and Win Rate (%) improvement with tool clustering algorithm integration using GPT-4 across various baselines.

| Method | G1-Tool. | | G2-Inst. | | G2-Cat. | | G3-Inst. | |
|---|---|---|---|---|---|---|---|---|
| | Pass. | Win. | Pass. | Win. | Pass. | Win. | Pass. | Win. |
| ***w/o Toolkit Integration*** | | | | | | | | |
| ReACT | 46.5 | 59.3 | 64.5 | 62.8 | 67.5 | 58.5 | 42.0 | 73.5 |
| AdaPlanner | 60.5 | 64.8 | 75.5 | 73.5 | 68.0 | 52.8 | 70.0 | 79.5 |
| DFSDT | 72.0 | 69.3 | 77.5 | 72.0 | 69.5 | 56.8 | 71.0 | 81.5 |
| ***with Toolkit Integration*** | | | | | | | | |
| ReACT | 55.5 | 62.3 | 72.5 | 69.0 | 68.0 | 61.3 | 59.0 | 77.8 |
| AdaPlanner | 72.5 | 71.3 | 78.5 | 76.0 | 70.0 | 57.5 | 73.0 | 83.0 |
| Tool-Planner | **78.5** | **75.8** | **83.5** | **79.8** | **77.5** | **70.3** | **83.0** | **92.0** |

The experimental results indicate that the performance significantly improved after applying our tool clustering algorithm, even under a single-chain approach. **Tool clustering helps provide alternative solutions to problems.** ReACT with Toolkit outperformed ReACT in all metrics. For instance, G1-Tool's pass rate increases by 9.0%, and G3-Inst's win rate rises by 4.3%. These results demonstrate significant performance enhancements. Moreover, the overall performance of ReACT with Toolkit is comparable to AdaPlanner. For example, in G2-Inst, the pass rate of ReACT with Toolkit is only 3.0% lower than that of AdaPlanner, while in G2-Cat, its win rate is 8.5% higher. Overall, ReACT with Toolkit significantly improves model performance through tool integration, narrowing the gap with better-performing algorithms and showing higher pass and win rates.

Similarly, the performance of AdaPlanner significantly improved after integrating the toolkit. Meanwhile, the pass and win rates of AdaPlanner with Toolkit are slightly higher than those of DFSDT. For instance, in G2-Inst, its win rate exceeds that of DFSDT by 4.0%, with similar improvements observed in other datasets. This indicates that after applying our tool clustering algorithm and using the toolkit as node states in the search process, the initially underperforming AdaPlanner can surpass

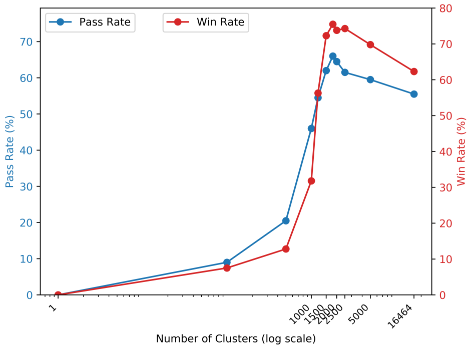

Figure 3: Evaluation of the effects of different numbers of clusters on G1-inst.

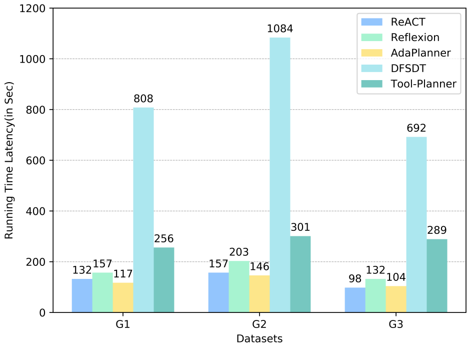

Figure 4: The average running time (sec) of different methods in various datasets.

DFSDT, which doesn't use any classification methods, fully demonstrating the superiority of the clustering algorithm in tool planning.

Our Tool-Planner method derives upon DFSDT by replacing its API node with the toolkit. As seen from the table, the performance of DFSDT improved with the adoption of our clustering algorithm. The win rate surges by 13.5% in G2-Cat and soares to 92.0% in G3-Inst. This demonstrates that our innovatively proposed toolkit exhibits significant enhancements in both single and multiple-tool usage scenarios, confirming its effectiveness in practical applications and heralding a new advancement in the field of tool utilization.

**Impact on different numbers of clusters.** For Tool-Planner, the size of $k$ value in tool clustering has a crucial impact on overall performance. Considering that tools with identical function have closely related tool embeddings, an appropriate number of clusters helps in properly categorizing tools by their functionality. When performing task planning, the model focuses more on the problems that the toolkit can solve rather than the specific details of each API. We set a range of $k$ values and conduct experiments on pass rate and win rate to understand the relationship between configuration size and model performance.

On ToolBench, G1-inst achieves the best results when the average size of a cluster is about 9. As shown in Figure 3, while $k$ is large, the performance of our framework gradually declined. This indicates that with insufficiently classified tools, the time consumption for solution space search and re-planning is considerable. Additionally, due to insufficient exploration of the solution space, some feasible solutions are not obtained, resulting in a lower overall pass rate and win rate. On the other hand, when $k$ is too small, the clustering effect significantly deteriorates, causing the clustering of different methods to mix together. This leads to incorrect tool calls and the generation of unsolvable next-step information. Therefore, a reasonable number of clusters is crucial for problem-solving. This is closely related to the distribution of tools and datasets.

**Text embeddings model on tool clustering.** Tool clustering simulates the human process of categorizing tools. In Tool-Planner, the effectiveness of clustering and the similarity of the resulting toolkits are crucial to our method. Tool clustering learns from tool demonstration and documentation, generating sentence embeddings based on their respective functions. In the tool clustering process, we use the SimCSE model to calculate the similarity between tools. To compare the clustering effectiveness of different similarity algorithms and their impact on the final results, we experiment with various similarity algorithms, including different text embedding models like RoBERTa-base (Liu et al., 2019), Contriever (Izacard et al., 2022), and *text-embedding-ada-02*. We evaluate these algorithms based on pass rate metrics.

Table 5. The pass rate (%) results of different text clustering models on Tool-Planner.

| **Model** | **G1-inst.** | **G2-inst.** | **G3-inst.** |
|---|---|---|---|
| RoBERTa-base | 60.5 | 76.5 | 73.0 |
| Contriever | 61.0 | 78.5 | 76.5 |
| *text-embedding-ada-02* | 63.5 | 81.5 | 78.0 |
| SimCSE | **66.0** | **83.5** | **83.0** |

Table 5 shows the impact of different text embedding models on the final results. **SimCSE shows robust ability to generate tool embeddings.** It can be seen that the SimCSE model achieves the best performance among the four text embedding models, indicating that SimCSE can better understand the informational knowledge of tool functions. Meanwhile, the performance of different text embedding models is generally similar, but task-specific embeddings generated through fine-tuning may perform better in more suitable scenarios.

**Efficiency evaluation.** To comprehensively understand the overhead of different algorithms in practical applications, we compare their execution speeds and test them on various datasets. This comparison not only demonstrates the efficiency differences between the methods but also helps us understand the distinct planning processes and the number of tool invocations for each algorithm. The evaluation results are shown in Figure 4.

Tool-Planner demonstrates a significant efficiency improvement compared to DFSDT. When an error occurs, Tool-Planner immediately selects another tool with the same function from the toolkit, quickly completing the tool replacement without affecting the original plan. It fully explores the feasible area, attempting alternative paths only after all APIs in the toolkit have been tried. On the other hand, DFSDT tries to find an API when an error occurs. If it determines there are no feasible APIs, it abandons the current state and continues searching. the new planning methods proposed by LLMs are generally more complex, leading to a higher likelihood of tool errors on new planning paths. This repeated plan adjustment not only inadequately explores feasible solutions but also wastes previous computation results. Moreover, we can see that, compared to path search solutions like ReACT and AdaPlanner, the latency of Tool-Planner is only twice as much, while DFSDT's latency is 6-8 times higher. This indicates that Tool-Planner can effectively find feasible tools and select appropriate plans for problem reasoning.

### 4.4 Error Analysis

In this section, we summarize and categorize the issues directly leading to failure or having potential failure risks in each step to conduct a more specific analysis. The types and distribution of failures are shown in Table 5. We also provide examples of each error type in Appendix H.

In the Figure 5, we can see that *Invalid Input Parameters* is the most frequent error type. There are primarily two reasons for this error: **(1)** When calling the API, the parameters provided to it do not meet the expected content, resulting in invalid input. **(2)** Users may misunderstand the content of the parameters required. Similar error types include *False API Call Format* and *Miss Input Parameters*, both of which arise during API invocation. Methods to mitigate these errors include providing users with more understandable prompts during the information input stage and stricter validation and filtering of input data during the model inference stage to ensure data integrity and accuracy.

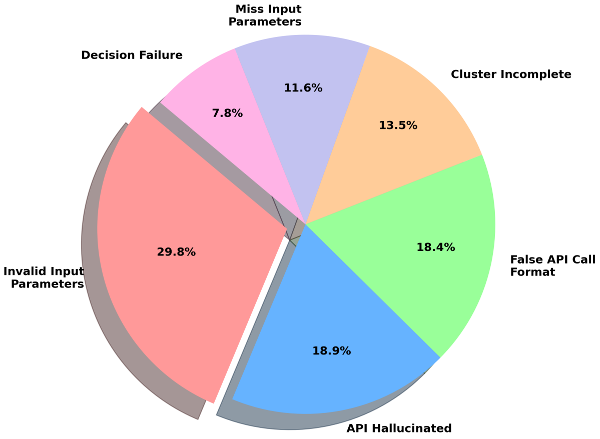

Figure 5: Distribution of reasoning errors.

The second most common error type is *API Hallucinated*, which is a prevalent mistake. This occurs when the model attempts to call an API as proposed in the plan but cannot find an API with a matching name. Methods to mitigate the hallucination in LLMs include providing clear and accurate API documentation to enable the model to interpret and use the information correctly, and timely updating the model with information about APIs to avoid hallucination issues caused by API updates.

Furthermore, *Cluster Incomplete* error occurs when the model overlooks some APIs due to incomplete clustering, resulting in the failure to identify APIs that could solve the problem. This type of error occurs due to the newly introduced clustering algorithm in this paper, but the occurrence rate is low, at only **13.5%**. Methods to mitigate this error include improving the clustering algorithm to enhance its performance and providing clear and accurate API documentation to facilitate clustering.

## 5 Related Work

**Task planning with LLMs.** Trained on extensive corpora, LLMs encompass a wealth of common-sense knowledge for task planning (Pallagani et al., 2023; Sun et al., 2023b). Consequently, generative methods have emerged as a hot topic in recent years. In considering the utilization of LLMs, some studies directly generate entire plans without executing them in the environment (Singh et al., 2023; Wu et al., 2023; Liang et al., 2023a; Lin et al., 2023; Yang et al., 2023; Zeng et al., 2022). However, these studies ignore the mechanism to correct decisions, which could result in a chain of errors starting from the initial ones. Reflexion (Shinn et al., 2023) mitigates this issue by requiring LLMs to reflect on past failures. The DFSDT proposed by ToolLLM (Qin et al., 2024) extends Reflexion to a more general method by allowing LLMs to evaluate different reasoning paths and select the most promising one. Our approach creatively utilizes toolkits for plan generation.

**Tree-based modeling for inference in LLMs.** Most LLM-based agents employ either open-loop or closed-loop systems, relying on linear reasoning or planning structures. To explore multiple branches in the action space, Self-consistency (Wang et al., 2022) samples multiple chains of thought, which can be seen as multiple i.i.d. solution paths in the decision space, and selects the best answer through majority voting. Some works (Yao et al., 2024; Long, 2023) propose an alternative solution of chains of thought, called “tree-of-thought”. These studies focus on reasoning tasks without involving interaction between the internal steps of the tree and the environment. Additionally, RAP (Hao et al., 2023a) combines world models with rewards in advanced MCTS search methods. To avoid exhaustive exploration like MCTS, Toolchain* (Zhuang et al., 2023) proposes a method that integrates efficient A* search with the effective reasoning capability of LLM, and Tree-Planner (Hu et al., 2023) samples different paths once and aggregated them into an action tree. Most methods fail in multi tools scenarios, but our Tool-Planner effectively addresses the issue of tool usage efficiency.

**LLMs for tool use.** The latest research in language modeling explores the use of external tools to complement the knowledge stored in model weights (Qin et al., 2023). This approach allows tasks like precise computation or information retrieval to be offloaded to external modules, such as Python interpreters or search engines (Mialon et al., 2023). These tools retrieve natural language knowledge from additional resources, as demonstrated by WebGPT (Nakano et al., 2021) and ReACT (Yao et al., 2022), which utilize search APIs to tap into these sources. Other methods, such as Toolformer (Schick et al., 2023), ART (Paranjape et al., 2023), ToolkenGPT (Hao et al., 2023b), leverage combinations of search APIs, question-answer APIs, machine translation APIs, calculators, and other tools to address various NLP tasks. ChatGPT Plugin[1] and TaskMatrix.AI (Liang et al., 2023b) show the potential of LLMs integrated with thousands to millions of APIs. LATM (Cai et al., 2023) and CREATOR (Qian et al., 2023) utilize GPT-4 to create API tools. Our Tool-Planner integrates APIs with the same or similar functions into toolkits, which significantly enhancing the ability to solve sub-problems.

## 6 Conclusion

In this paper, we present Tool-Planner, a framework for task planning based on tool clustering in tool learning. This framework enables flexible adjustments among tools with the same function. When errors occur in task planning, other tools within the same toolkit can be selected to maintain relative consistency of the plan and ensure effective and thorough exploration of the solution space. After all tools within a toolkit have been attempted, we switch to a new toolkit-based task planning to dynamically adjust the planning process. Compared to existing algorithms, our method finds the API to solve the current task more quickly, thus completing the task. Experiments show that our method has a higher pass rate and win rate compared to different baselines. Additionally, the ablation study demonstrates the effectiveness of the toolkit design. We also explore the impact of different clustering numbers and text similarity algorithms on clustering and planning effectiveness. Experimental results further confirm that our method can quickly complete task solutions. We believe this framework will contribute to the long-term development of the tool learning paradigm.

[1] https://openai.com/blog/chatgpt-plugins

## 7 Acknowledgements

This work was partly supported by the NSFC under No. 62402418 and No. 62102360. This work was also partly supported by the Key R&D Program of Ningbo under No. 2024Z115.

## References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Zakia Ahmad, Wahid Kaiser, and Sifatur Rahim. Hallucinations in chatgpt: an unreliable tool for learning. *Rupkatha Journal on Interdisciplinary Studies in Humanities*, 15(4):1–18, 2023.

David Arthur and Sergei Vassilvitskii. k-means++ the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 1027–1035, 2007.

Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, et al. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023*, 2023.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

Yu Du, Fangyun Wei, and Hongyang Zhang. Anytool: Self-reflective, hierarchical agents for large-scale api calls. *arXiv preprint arXiv:2402.04253*, 2024.

Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. Complexity-based prompting for multi-step reasoning. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=yf1icZHC-l9`.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. Simcse: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 6894–6910, 2021.

Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*, 2024a. URL `https://openreview.net/forum?id=Sx038qxjek`.

Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. ToRA: A tool-integrated reasoning agent for mathematical problem solving. In *The Twelfth International Conference on Learning Representations*, 2024b. URL `https://openreview.net/forum?id=Ep0TtjVoap`.

Nuno M Guerreiro, Duarte M Alves, Jonas Waldendorf, Barry Haddow, Alexandra Birch, Pierre Colombo, and André FT Martins. Hallucinations in large multilingual translation models. *Transactions of the Association for Computational Linguistics*, 11:1500–1517, 2023.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*, 2023a.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *Advances in Neural Information Processing Systems*, 36, 2023b.

Mengkang Hu, Yao Mu, Xinmiao Chelsey Yu, Mingyu Ding, Shiguang Wu, Wenqi Shao, Qiguang Chen, Bin Wang, Yu Qiao, and Ping Luo. Tree-planner: Efficient close-loop task planning with large language models. In *The Twelfth International Conference on Learning Representations*, 2023.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised dense information retrieval with contrastive learning. *Transactions on Machine Learning Research*, 2022. ISSN 2835-8856. URL `https://openreview.net/forum?id=jKN1pXi7b0`.

Jaewook Lee, Digory Smith, Simon Woodhead, and Andrew Lan. Math multiple choice question generation via human-large language model collaboration. *arXiv preprint arXiv:2405.00864*, 2024.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500. IEEE, 2023a.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. *arXiv preprint arXiv:2303.16434*, 2023b.

Bill Yuchen Lin, Chengsong Huang, Qian Liu, Wenda Gu, Sam Sommerer, and Xiang Ren. On grounded planning for embodied tasks with language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 13192–13200, 2023.

Yanming Liu, Xinyue Peng, Tianyu Du, Jianwei Yin, Weihao Liu, and Xuhong Zhang. Era-cot: Improving chain-of-thought through entity relationship analysis. *arXiv preprint arXiv:2403.06932*, 2024a.

Yanming Liu, Xinyue Peng, Xuhong Zhang, Weihao Liu, Jianwei Yin, Jiannan Cao, and Tianyu Du. Ra-isf: Learning to answer and understand from retrieval augmentation via iterative self-feedback. *arXiv preprint arXiv:2403.06840*, 2024b.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Jieyi Long. Large language model guided tree-of-thought. *arXiv preprint arXiv:2305.08291*, 2023.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*, 2023.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Vishal Pallagani, Bharath Muppasani, Keerthiram Murugesan, Francesca Rossi, Biplav Srivastava, Lior Horesh, Francesco Fabiano, and Andrea Loreggia. Understanding the capabilities of large language models for automated planning. *arXiv preprint arXiv:2305.16151*, 2023.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

Cheng Qian, Chi Han, Yi R Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. Creator: Disentangling abstract and concrete reasonings of large language models through tool creation. *arXiv preprint arXiv:2305.14318*, 2023.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, et al. Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*, 2023.

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. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=dHng2O0Jjr`.

Rapid. Rapid api, 2023. `https://rapidapi.com/`.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2023.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2023.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11523–11530. IEEE, 2023.

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

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. Adaplanner: Adaptive planning from feedback with language models. *Advances in Neural Information Processing Systems*, 36, 2023a.

Simeng Sun, Yang Liu, Shuohang Wang, Chenguang Zhu, and Mohit Iyyer. Pearl: Prompting large language models to plan and execute actions over long documents. *arXiv preprint arXiv:2305.14564*, 2023b.

Sindhu Tipirneni, Ravinarayana Adkathimar, Nurendra Choudhary, Gaurush Hiranandani, Rana Ali Amjad, Vassilis N Ioannidis, Changhe Yuan, and Chandan K Reddy. Context-aware clustering using large language models. *arXiv preprint arXiv:2405.00988*, 2024.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Vijay Viswanathan, Kiril Gashteovski, Kiril Gashteovski, Carolin Lawrence, Tongshuang Wu, and Graham Neubig. Large language models enable few-shot clustering. *Transactions of the Association for Computational Linguistics*, 12:321–333, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Zhenyu Wu, Ziwei Wang, Xiuwei Xu, Jiwen Lu, and Haibin Yan. Embodied task planning with large language models. *arXiv preprint arXiv:2307.01848*, 2023.

Sherry Yang, Ofir Nachum, Yilun Du, Jason Wei, Pieter Abbeel, and Dale Schuurmans. Foundation models for decision making: Problems, methods, and opportunities. *arXiv preprint arXiv:2303.04129*, 2023.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Andy Zeng, Maria Attarian, Brian Ichter, Krzysztof Choromanski, Adrian Wong, Stefan Welker, Federico Tombari, Aveek Purohit, Michael Ryoo, Vikas Sindhwani, et al. Socratic models: Composing zero-shot multimodal reasoning with language. *arXiv preprint arXiv:2204.00598*, 2022.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Zhiyuan Liu, Peng Zhang, Yuxiao Dong, and Jie Tang. Glm-130b: An Open Bilingual Pre-trained Model. In *The Eleventh International Conference on Learning Representations*, 2023.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open Pre-trained Transformer Language Models. *ArXiv*, abs/2205.01068, 2022.

Yuwei Zhang, Zihan Wang, and Jingbo Shang. Clusterllm: Large language models as a guide for text clustering. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 13903–13920, 2023.

Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A Rossi, Somdeb Sarkhel, and Chao Zhang. Toolchain*: Efficient action space navigation in large language models with a* search. In *The Twelfth International Conference on Learning Representations*, 2023.

## A Broader Impact and Limitations

**Broader Impact.** Tool-Planner innovatively integrates toolkits, achieving efficient search in the solution space for task planning. The Tool-Planner paradigm not only performs well on datasets like ToolBench but can also be applied to complex real-world API task scenarios. This approach allows us to integrate APIs of different categories and information types by placing the same type of APIs into a toolkit, enabling multiple attempts with similar tools when addressing practical problems without extensive exploration across a wide solution space.

**Limitation.** Tool-Planner has some limitations. Firstly, it heavily relies on clustering effectiveness. When there are significant functional differences within a cluster, the model may fail to find the appropriate tool for reasoning. This necessitates setting an appropriate cluster size during clustering to merge tools with similar functions. Secondly, there is still room to explore better tool invocation schemes. In our scenario, compared to linear invocation methods like ReACT Yao et al. (2022), Tool-Planner still has twice the time delay. We hope to further study methods for tool selection within clusters in the future.

**Algorithm 1** Tool-Planner Exploration Search

```
Input: Query: $x_0$; API docs: $\mathcal{D}_{\mathcal{T}}$; Toolkits functionality: $\{m_{\mathcal{T}_i}\}_{i=1}^{K}$, Planning length: $s$, Toolkit planning: $\mathcal{P}_{\mathcal{T}} = \{P_i\}_{i=1}^{s}$
Output: Intermediate reasoning step: $\mathcal{X} = \{x_i\}_{i=1}^{s}$;
$P_{now} \leftarrow P_1$; $\mathcal{X} \leftarrow \{x_0\}$; $l \leftarrow 1$; $E \leftarrow \varnothing$;
while $l$ is not higher to $s$ do                          ▷ // Finding a Suitable Answer
    $c_{now} \leftarrow \varnothing$
    $d_{now} \leftarrow \varnothing$
    for $d_l \in \mathcal{D}_{\mathcal{T}_{P_{now}}}$ do                  ▷ // Task Planning within Same Toolkits
        $param \leftarrow$ LLMs$(x_{l-1}, d_l)$;
        $c_l \leftarrow F_{d_l}(param)$;                             ▷ // Making Function Call
        if $c_l$ is not error message then
            $c_{now} = c_l$;                                 ▷ // Fetching Valid Tool
            Break Loop;
        end if
    end for
    if $c_{now}$ is not $\varnothing$ then
        $x_l \leftarrow$ LLMs$(\mathcal{X}, \{c_i\}_{i=1}^{l}, d_{now})$;     ▷ // Generate the Intermediate State Result
        Add $x_l$ to $\mathcal{X}$ under $x_{l-1}$;
        $l \leftarrow l + 1$;
    end if
    if $c_{now}$ is $\varnothing$ then                      ▷ // Task Replanning across Toolkits
        $\mathcal{P}'_{\mathcal{T}} \leftarrow$ LLMs$(\mathcal{X}, \{c_i\}_{i=1}^{l}, \mathcal{P}_{\mathcal{T}}, E)$;
        Add $\mathcal{P}_{\mathcal{T}}$ to $E$;
        $l \leftarrow LCA(\mathcal{P}'_{\mathcal{T}}, \mathcal{P}_{\mathcal{T}})$;       ▷ // Restart from Lowest Common Ancestor Node
    end if
    $P_{now} \leftarrow P_l$;
end while
return $\mathcal{X} = \{x_i\}_{i=1}^{s}$.
```

## B Implementation Details

**Dataset.** ToolBench Qin et al. (2024) serves as a benchmark designed to evaluate the API calling capabilities of agents. The ToolBench team gathered 16,464 real-world APIs from RapidAPI Rapid (2023) and compiled multiple execution traces for use as a training corpus. This is the only large enough benchmark that contains enough APIs to shows the ability of tool clustering and simulate the real world APIs usage.

The ToolBench test set is categorized into six distinct groups: G1-instruction, G1-tool, G1-category, G2-instruction, G2-category, and G3-instruction. Groups labeled with "instruction" include test instructions that utilize tools from the training set, thereby representing in-domain test data. In contrast, groups labeled with "tool" or "category" feature test instructions that do not use tools from the training set, represent out-of-domain test data. Each group consists of 100 user instructions, totaling 400 instructions for the in-domain test set and 200 instructions for the out-of-domain test set.

**Environment.** In tool learning, we primarily rely on a series of API function calls to complete planning task processing. In this process, we use toolsets as nodes in the dynamic search tree of tool learning. The SimCSE Gao et al. (2021) model we adopt is a pre-trained supervised fine-tuning model based on RoBERTa-base Liu et al. (2019). We generate corresponding explanatory information for each API using the prompts provided in the appendix and embed this information into the KG class. We utilize the Kmeans++ Arthur & Vassilvitskii (2007) algorithm, which can quickly converge by pre-setting initial cluster nodes. Additionally, both the OpenAI API and Claude API[2] interfaces we use have an initial temperature setting of 0.3 for inference and planning. We choose $k = 1800$ in our experiment if no specific mention of its setting.

[2] https://www.anthropic.com/api

Table 6. Comparision of Win Rate of DFSDT, ToolChain* and Tool-Planner.

| Model | Method | Home Search | G1-Inst | G1-Tool | G1-Cat | G2-Inst | G2-Cat | G3-Inst |
|---|---|---|---|---|---|---|---|---|
| GPT3.5 | DFSDT | 71.0 | 58.3 | 59.8 | 56.8 | 70.5 | 59.3 | 68.0 |
| GPT3.5 | ToolChain* | 69.0 | 56.3 | 58.5 | 53.0 | 67.5 | 58.0 | 61.5 |
| GPT3.5 | Tool-Planner | 75.0 | 63.3 | 63.5 | 61.3 | 72.3 | 63.5 | 67.5 |
| GPT4 | DFSDT | 73.0 | 65.8 | 69.3 | 65.3 | 72.0 | 56.8 | 81.5 |
| GPT4 | ToolChain* | 76.0 | 38.5 | 70.5 | 64.0 | 73.3 | 54.0 | 75.5 |
| GPT4 | Tool-Planner | 79.0 | 75.5 | 75.8 | 71.8 | 79.8 | 70.3 | 92.0 |

## C Planning Exploration Details

When planning and exploring a task, we call upon the existing tools based on the current plan and the information contained in the toolkit. The overall process can be illustrated in the form of the Algorithm 1.

We utilize the lowest common ancestor on the decision tree to find the branching node that represents the common prefix toolkit of the two schemes, thereby achieving search complexity similar to DFS. In the specific implementation, we rely on the prompting for path reselection and search to expand the solution space.

## D Comparison with Toolchain*

ToolChain* rely on A* algorithms and require analysis through multiple tool invocations during path exploration. They are primarily applied to relatively small categories with a limited number of tools (only 10–100). Therefore, ToolChain* performs well in such scenarios. However, when the number of tools increases, such as in real-world scenarios or comprehensive benchmarks like APIBench and ToolBench, the performance of ToolChain* significantly degrades and may even become ineffective. Thus, our primary focus has been on detailed performance comparisons with strategy learning or planning-based methods. For a fair comparison, we have still reported ToolChain*'s performance on 1) the Home Search subtask and 2) the overall dataset. The comparison results are shown in the Table 6. As the number of tools increases, the pass rate of heuristic-based methods declines significantly due to the massive reasoning required, whereas the Tool-Planner approach maintains competitive performance in terms of both win rate and pass rate in the final results.

## E Tool-Planner on Smaller LLMs

To understand the performance of Tool-Planner across different models, we explored the performance and effectiveness of Llama-2-13B within our framework. Since tool learning requires strong reasoning capabilities to understand the functionality of tool APIs and their documentation, DFSDT performs poorly with Llama-2-13B due to its inadequate comprehension of functionalities, rendering it incapable of effectively completing recent actions and generating effective plans. Additionally, during the generation process, Llama-2-13B is prone to hallucination issues Guerreiro et al. (2023); Ahmad et al. (2023) due to deficiencies in parametric knowledge. Our method integrates multiple APIs into a toolset, where the APIs within the toolset have the same or similar functions. This means we can use the functional description of the toolset to aid in reasoning and planning throughout the process. Notably, in Chain-of-Thought Wei et al. (2022); Liu et al. (2024a); Fu et al. (2023) scenarios, even small models exhibited excellent reasoning and planning capabilities. Therefore, it is beneficial to separately explore the impact and role of small models on reasoning and execution in this context. The experimental results are shown in the Table 7.

As we can see, DFSDT method has a success rate and win rate of zero due to its inadequate understanding of API documentation, resulting in generated content that cannot solve the problem. In contrast, within our framework using the Llama-2-13B model, it can generate some complete reasoning results, but it still fails in most cases with low generation quality. However, when we use the Llama-2-13B model as a planning model, the overall performance is not significantly different from using LLMs for reasoning. This indicates that the bottleneck for smaller LLMs in tool learning is

Table 7. Evaluation of Pass Rate and Win Rate for planning models and behavior models of different sizes.

| Method | Planning Model | Behavior Model | G1-Inst. | | G1-Tool. | | G1-Cat. | | G2-Inst. | | G2-Cat. | | G3-Inst. | | Avg. | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Pass. | Win. | Pass. | Win. | Pass. | Win. | Pass. | Win. | Pass. | Win. | Pass. | Win. | Pass. | Win. |
| DFSDT | Llama-2-13B | Llama-2-13B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Tool-Planner | Llama-2-13B | Llama-2-13B | 12.5 | 13.3 | 8.5 | 12.0 | 9.5 | 5.8 | 11.5 | 14.5 | 9.5 | 13.3 | 14.0 | 3.0 | 10.9 | 10.3 |
| Tool-Planner | GPT-4 | Llama-2-13B | 14.5 | 15.8 | 7.0 | 9.5 | 10.5 | 6.8 | 9.5 | 16.3 | 7.0 | 15.3 | 16.0 | 10.5 | 10.8 | 12.4 |
| Tool-Planner | Llama-2-13B | GPT-4 | 62.5 | 71.8 | 74.5 | 73.3 | 71.5 | 69.8 | 80.0 | 81.3 | 71.5 | 65.8 | 79.0 | 87.0 | 73.1 | 74.8 |
| Tool-Planner | GPT-4 | GPT-4 | 66.0 | 75.5 | 78.5 | 75.8 | 75.0 | 71.8 | 83.5 | 79.8 | 77.5 | 70.3 | 83.0 | 92.0 | 77.3 | 77.5 |

Table 8. Performance and Efficiency Analysis of Clustering Methods in Tool-Planner Reasoning.

| Clustering Option | $\epsilon$ | G1-Inst | G2-Inst | G3-Inst | TorchHub | HuggingFace | TensorFlow |
|---|---|---|---|---|---|---|---|
| Tool-Planner(k-means best) | - | 63.3 | 72.3 | 67.5 | 77.6 | 75.4 | 72.4 |
| Tool-Planner(DBSCAN) | 0.001 | 57.5 | 68.3 | 66.3 | 75.1 | 72.8 | 65.2 |
| Tool-Planner(DBSCAN) | 0.01 | 61.8 | 71.8 | 66.5 | 77.3 | 76.5 | 70.3 |
| Tool-Planner(DBSCAN) | 0.02 | **64.3** | 70.3 | **68.0** | 78.5 | 77.2 | 71.9 |
| Tool-Planner(DBSCAN) | 0.04 | 63.5 | 69.8 | 68.3 | 77.0 | 73.1 | 69.7 |
| Tool-Planner(DBSCAN) | 0.08 | 59.5 | 59.3 | 66.0 | 73.2 | 65.8 | 61.2 |
| Tool-Planner(DBSCAN) | 0.16 | 50.5 | 45.5 | 57.5 | 65.1 | 56.9 | 55.9 |

mainly their ability to understand tools and their documentation, whereas they can achieve reasonably good planning with coarse-grained information, aiding in task planning for overall tool learning. Future work can delve deeper into this aspect.

## F Variant of Tool-Planner on DBSCAN Clustering

Our approach replaces the original tool reasoning components with ReACT and AdaPlanner. First, we perform a preprocessing step involving tool clustering. Then, we allocate tools from the clustered toolkits to assist the model in completing sub-tasks within the planning process. This method is highly adaptable to scenarios that rely on policy-based learning and planning approaches.

The adaptive adjustment of the number of clusters allows us to fine-tune the clustering granularity within a specific range, such as by using the DBSCAN method. Our primary objective is to ensure that the overall reasoning process remains controllable, while predefining the number of clusters helps manage the time required for re-exploring the solution space. The advantage of DBSCAN lies in its ability to more accurately group tools with similar functionalities. Given that the encoded features contain more unsupervised semantic information in the vector directions, we use cosine similarity as the distance metric for DBSCAN and set the default $min_{pts}$ to 1.

To conduct a comprehensive comparison, as referenced in the Table 8, we evaluated the clustering performance of DBSCAN. The results indicate that DBSCAN achieved a slight performance improvement over methods like k-means++. Therefore, in scenarios where performance is the primary focus, DBSCAN performs better when the number of clusters is appropriate. Conversely, in scenarios that emphasize overall efficiency and consistency, the k-means++ method ensures stable performance and time consumption throughout the process.

## G Future Exploration on Tools Effect with Causal Inference

Considering that the success rate of tool invocation may vary under different tools, in multi-tool task planning, we aim to select the optimal combination of tools based on the current state and historical data to accomplish the task. We also expect to use causal inference models to predict the impact of different tool combinations on the task outcome, and in future work, evaluate whether these models contribute to optimizing tool selection.

Let the probability of task success be $P(Y = 1 \mid T_1, T_2, \ldots, T_n)$, representing the probability of task success given the tool choices. This probability can be computed using a causal inference model, such as a regression model:

$$P(Y=1 \mid T_1, T_2, \ldots, T_n) = \sigma\left(\sum_{i=1}^{n} \beta_i T_i + \sum_{i \neq j} \alpha_{ij} T_i T_j + \epsilon\right)$$

where $\beta_i$ and $\alpha_{ij}$ represent the effects of individual tools and tool combinations, respectively, and $\sigma(\cdot)$ is the Sigmoid function that outputs the probability of task success. If a tool combination significantly improves the task success probability, we can consider this tool combination as having greater potential for future tasks. Conversely, if certain tool combinations have little or even negative effects on task success, these combinations should be avoided in subsequent tasks. In the clustering distance evaluation, this design helps in selecting the correct tools and optimizing spatial classification. We look forward to further research in future work to improve clustering performance under k-means or DBSCAN, thereby reducing redundancy and misclassification in the tool selection process.

## H Examples in Error Analysis

In this section, we give the examples for the issues directly leading to failure or having potential failure risks in each step to further conduct a more specific analysis.

### H.1 Invalid Input Prameters

**Error Example**

Invalid Input Parameters: Example 1

- **Example:** The user specifies a non-existent dietary preference when making a request to the RecipeAPI.
- **API Call:**

```
{
    "taste_preferences": ["sweet", "spicy"],
    "dietary_preference": "paleo"
}
```

- **Response:**

```
{
    "error": "Invalid input parameters. Please provide a
        valid dietary preference."
}
```

- **Analysis:** In this scenario, the user has specified ”paleo” as their dietary preference. However, the system does not recognize ”paleo” as a valid dietary option in its predefined list of dietary preferences. As a result, the API call returns an error indicating that the input parameters are invalid.

**Error Example**

Invalid Input Parameters: Example 2

- **Example:** The user provides an invalid value for the ‘diet‘ parameter when making a recipe search request.
- **API Call:**

```
GET /recipes/search?cuisine=Italian&diet=glutenfull
```

- **Response:**

```
{
        "error": "Invalid input parameters. The value '
            glutenfull' for 'diet' is not recognized. Please
            use a valid diet option such as 'glutenfree', '
            vegetarian', or 'vegan'."
    }
```

- **Analysis:** In this scenario, the user provided ‘glutenfull‘ as the value for the ‘diet‘ parameter. However, ‘glutenfull‘ is not a recognized or valid option in the predefined list of dietary preferences. Valid options might include ‘glutenfree‘, ‘vegetarian‘, or ‘vegan‘. As a result, the API returns an error indicating that the input parameters are invalid.

### H.2 API Hallucinated

**Error Example**

API Hallucinated: Example 1

- **Example:** The user incorrectly calls a deprecated API endpoint.
- **API Call:**

```
POST /v1/recipes/search
```

- **Response:**

```
{
    "error": "API endpoint not found. Please use the
    updated API endpoint /v2/recipes/search."
}
```

- **Analysis:** In this example, the user attempts to call a deprecated API endpoint ‘/v1/recipes/search‘. However, the system has updated the API and uses ‘/v2/recipes/search‘ as a replacement. Therefore, the user receives an error response indicating they have used a non-existent API endpoint.

**Error Example**

API Hallucinated: Example 2

- **Example:** The model misinterprets information from the API documentation, leading to a call to a non-existent API.
- **API Call:**

```
GET /recipes/retrieve?cuisine=Italian
```

- **Response:**

```
{
    "error": "API endpoint not found. Please refer to
    the correct documentation for available endpoints."
}
```

- **Analysis:** In this example, the user attempts to use a non-existent API endpoint ‘/recipes/retrieve‘ mentioned in the documentation to retrieve recipes for Italian cuisine. However, this endpoint does not exist. This could be because the model misinterprets information from the API documentation, leading the user to mistakenly call a non-existent API.

### H.3 False API Call Format

**Error Example**

False API Call Format: Example 1

- **Example:** The user sends parameters in the request body instead of as query parameters for a GET request.
- **API Call:**

```
GET /recipes/search
{
    "cuisine": "Italian",
    "diet": "vegetarian"
}
```

- **Response:**

```
{
    "error": "Invalid request format. Please provide
        parameters as query strings."
}
```

- **Analysis:** In this case, the user included parameters in the request body for a GET request. The API expects parameters to be sent as query strings, leading to an invalid request format error.

**Error Example**

False API Call Format: Example 2

- **Example:** The user incorrectly formats the date parameter when making a request for recipe suggestions.
- **API Call:**

```
{
    "user_id": "12345",
    "preferred_date": "12-31-2023"
}
```

- **Response:**

```
{
    "error": "Invalid input parameters. Please use the
        format YYYY-MM-DD for the date."
}
```

- **Analysis:** In this case, the user provided the date in the format MM-DD-YYYY instead of the expected format YYYY-MM-DD. This mismatch in expected format led to an invalid input parameters error.

### H.4 Cluster Incomplete

*Cluster Incomplete* error occurs when the model overlooks some APIs due to incomplete clustering, resulting in the failure to identify APIs that could solve the problem.

### H.5 Miss Input Parameters

**Error Example**

Miss Input Parameters: Example 1

- **Example:** The user omits the required ‘cuisine‘ parameter when making a recipe search request.
- **API Call:**

```
GET /recipes/search?diet=vegetarian
```

- **Response:**

```
{
    "error": "Missing required parameter: cuisine. Please
        provide a cuisine type."
}
```

- **Analysis:** The error occurred because the user did not include the required ‘cuisine‘ parameter in the query string. This parameter is essential for the API to filter and return the appropriate recipes, and its absence leads to an error response.

**Error Example**

Miss Input Parameters: Example 2

- **Example:** The user fails to provide the ‘user_id‘ when requesting personalized recipe recommendations.
- **API Call:**

```
POST /recipes/recommendations
{
    "preferences": ["spicy", "low-carb"]
}
```

- **Response:**

```
{
    "error": "Missing required parameter: user_id. Please
        provide your user ID."
}
```

- **Analysis:** In this case, the user did not include the ‘user_id‘ in the request body. The ‘user_id‘ is necessary for the API to retrieve personalized recommendations based on the user’s history and preferences. Without it, the API cannot process the request and returns an error.

### H.6 Decision Failure

The ”Decision Failure” error refers to situations where, even if the aforementioned errors don’t happen, failure still occurs. We attribute these errors to flaws in the model’s planning or decision-making, which prevent the completion of the intended task. This could be because the model doesn’t fully understand the task or the user’s context, leading to a lack of necessary steps in the planned process or the occurrence of illogical errors. This error requires strengthening the model’s understanding to avoid happening.

## I Ethics and Safeguard

Our work is based on open-source datasets and code for experimentation. All data and information comply with relevant code standards and data regulations, ensuring that there is no risk of privacy breaches or information leaks. The use of large language models in our paper is primarily applied to handling and solving task planning processes in most scenarios, as well as text processing during Chinese-to-English translation in token level. This fully complies with the conference's requirements and privacy security guidelines.

When using tools and interacting with large language models, we may utilize relevant information from instruction. It is important to note that hallucinations from large language models may lead to incorrect answers. Our approach can be further integrated into other frameworks.

## J Tool-Planner Prompting Template

Prompt of Plan Making

You will be provided with the toolkits, the clustered names of toolkits, and the descriptions of the function of the toolkits.Your task is to interact with API toolkits to construct user queries and use the functionalities of the toolkits to answer the queries. You need to identify the most suitable toolkits based on the user's requirements, and then outline your solution plan based on the toolkits you've selected.Remember, your goal is not to directly answer the query but to identify the toolkits and provide a solution plan. Here is the user's question:**[user query]**

Prompt of Plan Exploration

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

Prompt of the In-Toolkit Error Occurs

This is not your first attempt at this task. The previously called APIs have all failed, and you are now in the intermediate state of an In-Toolkit plan exploration. Before you decide on new actions, I will first show you the actions you have taken previously for this state. Then, you must develop an action that is different from all these previous actions. Here are some previous candidate actions: [previous API]. Now, please analyze the current state and then call another API within the same toolkit where the previously failed APIs are located.

Prompt of the Cross-Toolkit Error Occurs

This is not your first attempt at this task. All the APIs planned within the previous toolkits have failed, and you are now in the intermediate state of a Cross-Toolkit plan exploration. Before you decide on new actions, I will first show you the actions you have taken previously for this state. Then, you must develop an action that is different from all these previous actions. Here are some previous candidate actions: [previous API, previous toolkit]. Now, please revert to the previous node, revise the plan for this step, and use a different toolkit.

Prompt of the Ouputs

If you believe you have obtained the result capable of answering the task, please invoke this function to provide the final answer. Remember: the only part displayed to the user is the final answer, so it should contain sufficient information.

# K Case Study

## K.1 An example in G1 datasets

### K.1.1 Plan making

**Instruction:**

You will be provided with the toolkits, the clustered names of toolkits, and the descriptions of the function of the toolkits.Your task is to interact with API toolkits to construct user queries and use the functionalities of the toolkits to answer the queries. You need to identify the most suitable toolkits based on the user's requirements, and then outline your solution plan based on the toolkits you've selected.Remember, your goal is not to directly answer the query but to identify the toolkits and provide a solution plan. Here is the user's question: I'm planning to move to a new city. Can you suggest some locations to consider based on my preferences? I prefer a quiet neighborhood with at least 3 bedrooms and 2 bathrooms. It would be great if the location is close to parks and has a low crime rate. Also, provide me with the zip codes for the suggested locations using the getZipCodeByCity API.

**Answer:**

1. **Neighborhood Characteristics and Listings**:
   - **Proposed Toolkit Name:** Real Estate Listings
     - **Functionality:** The APIs in the Toolkit help us find listings with at least 3 bedrooms and 2 bathrooms in quiet neighborhoods.
2. **Crime Rate Information**:
   - **Proposed Toolkit Name:** Crime Data
     - **Functionality:** The APIs in the Toolkit provide information on the crime rates of different neighborhoods.
3. **Proximity to Parks**:
   - **Proposed Toolkit Name:** Local Amenities
     - **Functionality:** The APIs in the Toolkit provide data on the location of parks and recreational facilities.
4. **Zip Code Information**:
   - **Proposed Toolkit Name:** Geolocation
     - **Functionality:** The APIs in the Toolkit return the zip codes for the suggested neighborhoods.

### K.1.2 Plan Exploration

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 1: Identify Suitable Neighborhoods**
To identify suitable neighborhoods, we will use the **Property Listings API** to search for listings with at least 3 bedrooms and 2 bathrooms in quiet neighborhoods. Based on our toolkit, here is the likely API call:
**API Call: Property Listings API**

```
{
  "endpoint": "/getProperties",
  "method": "GET",
  "parameters": {
    "bedrooms": 3,
    "bathrooms": 2,
    "neighborhood": "quiet"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "data": [
    {
      "address": "123 Quiet St, Springfield",
      "city": "Springfield",
      "bedrooms": 3,
      "bathrooms": 2,
      "neighborhood": "Quiet Park",
      "price": 350000
    },
    {
      "address": "456 Peaceful Ave, Springfield",
      "city": "Springfield",
      "bedrooms": 4,
      "bathrooms": 3,
      "neighborhood": "Serenity Gardens",
      "price": 450000
    }
  ]
}
```

**Analysis:** The API call was successful and returned a list of properties matching the criteria. We now have data on potential neighborhoods and properties.
**Next Step:** Proceed to Step 2 of the plan: verifying the crime rates in the identified neighborhoods.

**Instruction:**

**Instruction**: Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 2: Verify Crime Rates**
We will use the **Crime Rate API** to check the crime rates for the neighborhoods ”Quiet Park” and ”Serenity Gardens”.
**API Call: Crime Rate API**

```
{
  "endpoint": "/getCrimeRate",
  "method": "GET",
  "parameters": {
    "neighborhood": "Quiet Park"
  }
}
```

**API Responses:**

```
{
  "status": "success",
  "data": {
    "Quiet Park": {
      "crime_rate": "low"
    },
    "Serenity Gardens": {
      "crime_rate": "low"
    }
  }
}
```

**Analysis:** The crime rate data indicates that both ”Quiet Park” and ”Serenity Gardens” have low crime rates, making them suitable neighborhoods based on your preferences.
**Next Step:** Proceed to Step 3 of the plan: finding the proximity to parks in these neighborhoods.

**Instruction:**

**Instruction**: Let’s begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step’s toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user’s requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 3: Find Proximity to Parks**
To find the proximity of parks in the neighborhoods ”Quiet Park” and ”Serenity Gardens”, we will use the **Parks and Recreation API**.
**API Call: Parks and Recreation API**

```
{
  "endpoint": "/getParksNearby",
  "method": "GET",
  "parameters": {
    "neighborhood": "Quiet Park"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "data": {
    "Quiet Park": [
      {
        "park_name": "Central Park",
        "distance": "0.5 miles"
      },
      {
        "park_name": "Riverside Park",
        "distance": "1.2 miles"
      }
    ],
    "Serenity Gardens": [
      {
        "park_name": "Greenfield Park",
        "distance": "0.3 miles"
      },
      {
        "park_name": "Lakeside Park",
        "distance": "1.0 miles"
      }
    ]
  }
}
```

**Analysis:** Both ”Quiet Park” and ”Serenity Gardens” neighborhoods have parks in close proximity, further meeting your criteria for a suitable location.
**Next Step:** Proceed to Step 4 of the plan: retrieve the zip codes for these neighborhoods using the **getZipCodeByCity API**.

**Instruction:**

**Instruction**: Let’s begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step’s toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user’s requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 4: Retrieve Zip Codes**
We will use the **getZipCodeByCity API** to get the zip codes for ”Quiet Park” and ”Serenity Gardens”.

**API Call: getZipCodeByCity API**

```
{
  "endpoint": "/getZipCode",
  "method": "GET",
  "parameters": {
    "city": "Springfield",
    "neighborhood": "Quiet Park"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "data": {
    "Quiet Park": "62704",
    "Serenity Gardens": "62711"
  }
}
```

**Analysis:** The zip codes for the neighborhoods ”Quiet Park” (62704) and ”Serenity Gardens” (62711) have been successfully retrieved.

### K.1.3 Outputs

**Instruction:**

If you believe you have obtained the result capable of answering the task, please invoke this function to provide the final answer. Remember: the only part displayed to the user is the final answer, so it should contain sufficient information.

**Outputs:**

Based on your preferences for a quiet neighborhood with at least 3 bedrooms and 2 bathrooms, proximity to parks, and a low crime rate, the neighborhoods ”Quiet Park” and ”Serenity Gardens” in Springfield are suitable options. The respective zip codes are 62704 and 62711.

## K.2 An example in G2 Datasets

### K.2.1 Plan Making

**Instruction:**

You will be provided with the toolkits, the clustered names of toolkits, and the descriptions of the function of the toolkits.Your task is to interact with API toolkits to construct user queries and use the functionalities of the toolkits to answer the queries. You need to identify the most suitable toolkits based on the user’s requirements, and then outline your solution plan based on the toolkits you’ve selected.Remember, your goal is not to directly answer the query but to identify the toolkits and provide a solution plan. Here is the user’s question:I want to surprise my sister with a trip to New York City. Can you find the average gas price in the city? Additionally, fetch the helicopter specifications for the Bell 206 model as my sister is fascinated by helicopters. Also, provide me with the gas price predictions for tomorrow in major Canadian cities.

**Answer:**

1. **Average Gas Price in New York City**:
   - **Proposed Toolkit Name:** Gas Prices
     - **Functionality:** The APIs in the Toolkit will help us find the current average gas price in New York City.
2. **Helicopter Specifications for Bell 206**:
   - **Proposed Toolkit Name:** Helicopter Specifications
     - **Functionality:** The APIs in the Toolkit will provide detailed specifications for the Bell 206 helicopter model.
3. **Gas Price Predictions in Major Canadian Cities**:
   - **Proposed Toolkit Name:** Gas Price Predictions
     - **Functionality:** The APIs in the Toolkit will provide predictions for gas prices for the next day in major Canadian cities.

### K.2.2 Plan Exploration

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 1: Get Average Gas Price in New York City**
To find the average gas price in New York City, we will use the **GasPriceAPI**. Based on our toolkit, here is the likely API call:
**API Call: GasPriceAPI**

```
{
  "endpoint": "/getGasPrice",
  "method": "GET",
  "parameters": {
    "cityName": "New York City"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "data": {
    "city": "New York City",
    "averageGasPrice": "$3.50 per gallon"
  }
}
```

**Analysis:** The API call was successful and returned the average gas price in New York City as $3.50 per gallon.
**Next Step:** Proceed to Step 2 of the plan: fetching the helicopter specifications for the Bell 206 model.

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 2: Fetch Helicopter Specifications for Bell 206**
To fetch the helicopter specifications for the Bell 206 model, we will use the **HelicopterSpecsAPI**. Based on our toolkit, here is the likely API call:
**API Call: HelicopterSpecsAPI**

```
{
  "endpoint": "/getHelicopterSpecs",
  "method": "GET",
  "parameters": {
    "model": "Bell 206"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "data": {
    "model": "Bell 206",
    "specifications": {
      "length": "39 ft 8 in",
      "height": "9 ft 4 in",
      "maxSpeed": "150 mph",
      "range": "430 miles"
    }
  }
}
```

**Analysis:** The API call was successful and returned the specifications for the Bell 206 helicopter.
**Next Step:** Proceed to Step 3 of the plan: getting gas price predictions for tomorrow in major Canadian cities.

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 3: Get Gas Price Predictions for Tomorrow in Major Canadian Cities**
To get gas price predictions for tomorrow in major Canadian cities, we will use the **GasPricePredictionAPI**. Based on our toolkit, here is the likely API call:
**API Call: GasPricePredictionAPI**

```
{
  "endpoint": "/getGasPricePredictions",
  "method": "GET",
  "parameters": {
    "country": "Canada"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "data": {
    "country": "Canada",
    "predictions": [
      {
        "city": "Toronto",
        "predictedPrice": "$1.40 per liter"
      },
      {
        "city": "Vancouver",
        "predictedPrice": "$1.45 per liter"
      }
    ]
  }
}
```

**Analysis:** The API call was successful and returned gas price predictions for major Canadian cities including Toronto and Vancouver.
**Next Step:** Present the retrieved data to the user.

### K.2.3 OUTPUTS

**Instruction:**

If you believe you have obtained the result capable of answering the task, please invoke this function to provide the final answer. Remember: the only part displayed to the user is the final answer, so it should contain sufficient information.

**Outputs:**

- **Average Gas Price in New York City**:
  - **City:** New York City
  - **Average Gas Price:** $3.50 per gallon
- **Helicopter Specifications for Bell 206**:
  - **Model:** Bell 206
  - **Specifications:**
    - **Length:** 39 ft 8 in
    - **Height:** 9 ft 4 in
    - **Max Speed:** 150 mph
    - **Range:** 430 miles

- **Gas Price Predictions in Major Canadian Cities**:
  - **Country:** Canada
  - **Predictions:**
    - **Toronto:** $1.40 per liter
    - **Vancouver:** $1.45 per liter

### K.3 An example in G3 Datasets

#### K.3.1 Plan Making

**Instruction:**

You will be provided with the toolkits, the clustered names of toolkits, and the descriptions of the function of the toolkits.Your task is to interact with API toolkits to construct user queries and use the functionalities of the toolkits to answer the queries. You need to identify the most suitable toolkits based on the user's requirements, and then outline your solution plan based on the toolkits you've selected.Remember, your goal is not to directly answer the query but to identify the toolkits and provide a solution plan. Here is the user's question: I want to surprise my sister with a trip to New York City. Can you find the average gas price in the city? Additionally, fetch the helicopter specifications for the Bell 206 model as my sister is fascinated by helicopters. Also, provide me with the gas price predictions for tomorrow in major Canadian cities.

**Answer:**

1. **Beach Destination Information**:
   - **Proposed Toolkit Name:** Travel Information
     - **Functionality:** The APIs in the Toolkit provide information on popular beach destinations, including weather forecasts and local attractions.
2. **Cocktail Recommendations**:
   - **Proposed Toolkit Name:** Recipes
     - **Functionality:** The APIs in the Toolkit offer a database of cocktail recipes, including refreshing options suitable for beach enjoyment.
3. **Stock Price Data**:
   - **Proposed Toolkit Name:** Financial Markets
     - **Functionality:** The APIs in the Toolkit provide historical open and close prices for specific stocks, including Qualcomm.
4. **Beach Activities Suggestions**:
   - **Proposed Toolkit Name:** Recreational Activities
     - **Functionality:** The APIs in the Toolkit offer information on various sports and recreational activities suitable for beach settings.

#### K.3.2 Plan Exploration

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 1: Gather Beach Destination Information**
To gather information on popular beach destinations, including weather forecasts and local attractions, we will use the **Travel Advisory API**. Here is the likely API call:
**API Call: Travel Advisory API**

```
{
  "endpoint": "/getBeachDestinations",
  "method": "GET",
  "parameters": {
    "location": "beach"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "data": [
    {
      "destination": "Maldives",
      "weather_forecast": "Sunny with occasional showers, temperature: 30
           degree",
      "attractions": ["Snorkeling", "Scuba Diving", "Relaxing on Beaches
          "]
    },
    {
      "destination": "Maui, Hawaii",
      "weather_forecast": "Partly cloudy, temperature: 28 degree",
      "attractions": ["Hiking", "Whale Watching", "Luaus"]
    },
    // More beach destinations...
  ]
}
```

**Analysis:** The API call successfully retrieved information on popular beach destinations, including weather forecasts and local attractions, such as the Maldives and Maui, Hawaii.
**Next Step:** Proceed to Step 2 of the plan: recommending refreshing cocktails suitable for beach enjoyment.

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 2: Recommend Refreshing Cocktails**
To suggest refreshing cocktails suitable for beach enjoyment, we will use the **Cocktail Recipe API**. Here is the likely API call:
**API Call: Cocktail Recipe API**

```
{
"endpoint": "/getRefreshingCocktails",
"method": "GET",
"parameters": {
"location": "beach"
}
}
```

**API Response:**

```
{
"status": "error",
"message": "Internal server error. Please try again later."
}
```

**Analysis:** The API call to the original Cocktail Recipe API encountered an internal server error, resulting in an unsuccessful response. This could be due to temporary server issues or an invalid request format.
**Next Step:** To address this issue, we'll switch to another API within our toolkit. Let's replace the current API with the CocktailDB API, which provides a broader selection of cocktail recipes.

**Instruction:**

This is not your first attempt at this task. The previously called APIs have all failed, and you are now in the intermediate state of an In-Toolkit plan exploration. Before you decide on new actions, I will first show you the actions you have taken previously for this state. Then, you must develop an action that is different from all these previous actions. Here are some previous candidate actions: [Cocktail Recipe API]. Now, please analyze the current state and then call another API within the same toolkit where the previously failed APIs are located.

**Answer:**

**Step 2 (Revised): Recommend Refreshing Cocktails**
We'll use the **CocktailDB API** instead of the problematic API. Here is the likely API call:
**API Call: CocktailDB API**

```
{
"endpoint": "/getRandomCocktails",
"method": "GET",
"parameters": {
"number": 5,
"tag": "beach"
}
}
```

**API Response:**

```
{
"status": "success",
"data": [
{
"cocktail_name": "Sex on the Beach",
"ingredients": ["Vodka", "Peach Schnapps", "Orange Juice", "Cranberry
    Juice"],
"instructions": "Pour all ingredients into a shaker with ice. Shake well
    and strain into a glass."
},
{
"cocktail_name": "Tequila Sunrise",
"ingredients": ["Tequila", "Orange Juice", "Grenadine"],
"instructions": "Pour tequila and orange juice into a glass over ice. Add
     grenadine and stir."
},
// More refreshing cocktails...
]
}
```

**Analysis:** The API call successfully retrieved a wider variety of refreshing cocktail recipes suitable for beach enjoyment, including classics like Sex on the Beach and Tequila Sunrise.
**Next Step:** Proceed to Step 3 of the plan: fetching historical stock prices for Qualcomm.

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 3: Fetch Historical Stock Prices for Qualcomm**
To retrieve historical stock prices for Qualcomm, we'll utilize the **Alpha Vantage API**, which offers comprehensive financial market data. Here is the likely API call:
**API Call: Alpha Vantage API**

```
{
  "endpoint": "/time_series_daily",
  "method": "GET",
  "parameters": {
    "symbol": "QCOM",
    "outputsize": "compact"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "data": [
    {
      "date": "2024-05-20",
      "open": 155.00,
      "high": 156.20,
      "low": 153.50,
      "close": 154.75,
      "volume": 2500000
    },
    // More historical data...
  ]
}
```

**Analysis:** The API call successfully retrieved historical stock prices for Qualcomm, including open, high, low, and close prices for each trading day.
**Next Step:** Proceed to Step 4 of the plan: suggest sports activities that can be enjoyed on the beach.

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 4: Suggest Sports Activities for Beach Enjoyment**
For suggesting sports activities that can be enjoyed on the beach, we can use the **Beach Sports API**, which provides information on various beach sports and recreational activities. Here is the likely API call:
**API Call: Beach Sports API**

```
{
  "endpoint": "/getBeachSports",
  "method": "GET",
  "parameters": {
    "location": "beach"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "data": [
    "Beach Volleyball",
    "Surfing",
    "Snorkeling",
    "Jet Skiing",
    "Beach Soccer"
  ]
}
```

**Analysis:** The API call successfully retrieved a list of sports activities that can be enjoyed on the beach, including beach volleyball, surfing, snorkeling, jet skiing, and beach soccer.

### K.3.3 OUTPUTS

**Instruction:**

If you believe you have obtained the result capable of answering the task, please invoke this function to provide the final answer. Remember: the only part displayed to the user is the final answer, so it should contain sufficient information.

**Outputs:**

For your family beach vacation, we've gathered information on popular destinations and local attractions. The Maldives offers sunny weather with occasional showers and activities like snorkeling, scuba diving, and relaxing on beaches. Similarly, Maui, Hawaii provides partly cloudy weather with attractions such as hiking, whale watching, and luaus.
To keep you refreshed, we recommend trying cocktails like Sex on the Beach and Tequila Sunrise, which can be enjoyed while soaking up the sun.
Regarding financial data, we've retrieved historical stock prices for Qualcomm. On May 20th, 2024, the stock opened at $155.00, reached a high of $156.20, a low of $153.50, and closed at $154.75, with a trading volume of 2,500,000 shares.
For beach activities, you can indulge in beach volleyball, surfing, snorkeling, jet skiing, and beach soccer, ensuring there's something enjoyable for everyone.
With this comprehensive guide, your family vacation planning is complete, offering a perfect blend of relaxation, entertainment, and adventure!

## K.4 An example of No Error Occurrence.

### K.4.1 Plan Making

**Instruction:**

You will be provided with the toolkits, the clustered names of toolkits, and the descriptions of the function of the toolkits.Your task is to interact with API toolkits to construct user queries and use the functionalities of the toolkits to answer the queries. You need to identify the most suitable toolkits based on the user's requirements, and then outline your solution plan based on the toolkits you've selected.Remember, your goal is not to directly answer the query but to identify the toolkits and provide a solution plan. Here is the user's question: My friend sent a package with the tracking number YT2003521266065328. I need to track this package and get the tracking information. Additionally, detect the carrier for this tracking number.

**Answer:**

1. **Carrier Detection**:
   - **Proposed Toolkit Name:** Carrier Detection
     - **Functionality:** The APIs in the Toolkit detect the carrier associated with a given tracking number.
2. **Package Tracking**:
   - **Proposed Toolkit Name:** Package Tracking
     - **Functionality:** The APIs in the Toolkit provide detailed tracking information for a package using the tracking number.

### K.4.2 Plan Exploration

**Instruction:**

Let’s begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step’s toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user’s requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 1: Carrier Detection**
To identify the carrier for the tracking number YT2003521266065328, we will use the **AfterShip API**. Here is the likely API call:
**API Call: AfterShip API**

```
{
  "endpoint": "/couriers/detect",
  "method": "POST",
  "parameters": {
    "tracking_number": "YT2003521266065328"
  }
}
```

**API Response:**

```
{
  "meta": {
    "code": 200,
    "type": "Success",
    "message": "Successfully detected the courier."
  },
  "data": {
    "couriers": [
      {
        "slug": "yuntu",
        "name": "Yun Express",
        "phone": "+86 400 068 6006",
        "other_name": "",
        "web_url": "http://www.yuntrack.com/"
      }
    ]
  }
}
```

**Analysis:** The API call successfully detected the carrier for the tracking number YT2003521266065328 as Yun Express.
**Next Step:** Proceed to Step 2 of the plan: tracking the package using the detected carrier, Yun Express.

**Instruction:**

Let’s begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step’s toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user’s requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 2: Package Tracking**
To get the tracking information for the package with the tracking number YT2003521266065328, we will use the **AfterShip API** and the detected carrier Yun Express. Here is the likely API call:
**API Call: AfterShip API**

```
{
  "endpoint": "/trackings",
  "method": "POST",
  "parameters": {
    "tracking": {
      "slug": "yuntu",
      "tracking_number": "YT2003521266065328"
    }
  }
}
```

**API Response:**

```
{
  "meta": {
    "code": 200,
    "type": "Success",
    "message": "Successfully retrieved tracking information."
  },
  "data": {
    "tracking": {
      "id": "xxxxxxxxxxxxxxxxxxxxxxxx",
      "created_at": "2023-05-20T00:00:00+00:00",
      "updated_at": "2023-05-21T00:00:00+00:00",
      "last_update_time": "2023-05-21T00:00:00+00:00",
      "tracking_number": "YT2003521266065328",
      "slug": "yuntu",
      "active": true,
      "delivery_status": "In Transit",
      "shipment_package_count": 1,
      "origin_country_iso3": "CHN",
      "destination_country_iso3": "USA",
      "expected_delivery": "2023-06-01",
      "checkpoint": [
        {
          "city": "Shenzhen",
          "country_name": "China",
          "message": "Shipment picked up",
          "checkpoint_time": "2023-05-20T00:00:00+00:00"
        },
        {
          "city": "Los Angeles",
          "country_name": "USA",
          "message": "Arrived at sorting center",
          "checkpoint_time": "2023-05-21T00:00:00+00:00"
        }
      ]
    }
  }
}
```

**Analysis:** The API call successfully retrieved the tracking information for the package, indicating that it is currently in transit with the most recent checkpoint being its arrival at a sorting center in Los Angeles, USA.

**Conclusion:** The package with tracking number YT2003521266065328, handled by Yun Express, is currently in transit and has arrived at a sorting center in Los Angeles. The expected delivery date is June 1, 2023.

### K.4.3 OUTPUTS

**Instruction:**

If you believe you have obtained the result capable of answering the task, please invoke this function to provide the final answer. Remember: the only part displayed to the user is the final answer, so it should contain sufficient information.

**Outputs:**

Based on the provided tracking number YT2003521266065328, the carrier associated with the package is Yun Express. Upon tracking the package, it was found to be currently in transit and has arrived at a sorting center in Los Angeles, USA. The expected delivery date is June 1, 2023.

## K.5 AN EXAMPLE OF IN-TOOLKIT ERROR OCCURRENCE.

### K.5.1 PLAN MAKING

**Instruction:**

You will be provided with the toolkits, the clustered names of toolkits, and the descriptions of the function of the toolkits.Your task is to interact with API toolkits to construct user queries and use the functionalities of the toolkits to answer the queries. You need to identify the most suitable toolkits based on the user's requirements, and then outline your solution plan based on the toolkits you've selected.Remember, your goal is not to directly answer the query but to identify the toolkits and provide a solution plan. Here is the user's question: I need to ship a package internationally and require the services of a reliable transitaire. Can you provide me with a list of transitaires available? Additionally, I would like to know the health status of the SQUAKE authentication system.

**Answer:**

1. **Transitaire Identification**:
    - **Proposed Toolkit Name:** Shipping and Logistics
        - **Functionality:** The APIs in the Toolkit provide information about shipping services, including international shipping, and can help identify reliable transitaires.
2. **SQUAKE Authentication System Health Check**:
    - **Proposed Toolkit Name:** System Health Check

    - **Functionality:** The APIs in the Toolkit monitor the health status of various systems and services, including authentication systems like SQUAKE.

### K.5.2 Plan Exploration

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 1: Transitaire Identification**
To identify reliable transitaires for international shipping, we will use the **EasyPost API**. Here is the likely API call:
**API Call: EasyPost API**

```
{
  "endpoint": "/v2/carriers",
  "method": "GET",
  "parameters": {
    "filter": {
      "services": {
        "international": true
      }
    }
  }
}
```

**API Response:**

```
{
  "status": "success",
  "data": [
    {
      "name": "DHL Express",
      "code": "dhl_express",
      "services": {
        "international": true,
        // Other services...
      }
    },
    {
      "name": "FedEx",
      "code": "fedex",
      "services": {
        "international": true,
        // Other services...
      }
    },
    // More transitaires...
  ]
}
```

**Analysis:** The API call successfully retrieved a list of transitaires that offer international shipping services. Transitaires like DHL Express and FedEx are included in the response.
**Next Step:** Filter the list of transitaires based on reliability and international shipping capabilities, then provide the user with the refined list.

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 2: SQUAKE Authentication System Health Check**
To check the health status of the SQUAKE authentication system, we will use the **StatusPage API**.
**API Response:**

```
{
  "status": "error",
  "message": "Failed to fetch status information for the SQUAKE
      authentication system. Please try again later."
}
```

**Analysis:** In case of an unsuccessful API call, the error message suggests a failure to fetch status information for the SQUAKE authentication system. This could be due to various reasons such as network issues, server downtime, or invalid API credentials.
**Next Step:** Attempt to resolve the issue by checking for any network or server-related issues, ensuring the correctness of API credentials, and retrying the API call. If the problem persists, consider reaching out to the API provider for assistance or using alternative methods to verify the health status of the SQUAKE authentication system.

**Instruction:**

This is not your first attempt at this task; you are currently in the middle of the planning and exploration phase, and the APIs you previously called all failed. Now, please call different APIs within the same toolbox where the previous API calls. Remember to perform actions different from all previous APIs.

**Answer:**

**Step 2: SQUAKE Authentication System Health Check**
For this step, let's use the Pingdom API instead of the StatusPage API to check the health status of the SQUAKE authentication system.
**API Call: Pingdom API**

```
{
  "endpoint": "/checks",
  "method": "GET",
  "parameters": {
    "target": "https://squakesystem.com",
    "type": "http"
  }
}
```

**API Response:**

```
{
  "status": "error",
  "message": "Failed to perform health check for the SQUAKE
      authentication system. Please try again later."
}
```

**Analysis:** The API call to Pingdom for the health check of the SQUAKE authentication system encountered an error, indicating a failure to perform the health check.
**Next Step:** Since the Pingdom API call also failed, we may need to consider alternative methods or tools to check the health status of the SQUAKE authentication system. This could involve manual verification or exploring other system monitoring services within the same toolbox.

**Instruction:**

This is not your first attempt at this task. The previously called APIs have all failed, and you are now in the intermediate state of an In-Toolkit plan exploration. Before you decide on new actions, I will first show you the actions you have taken previously for this state. Then, you must develop an action that is different from all these previous actions. Here are some previous candidate actions: [Pingdom API]. Now, please analyze the current state and then call another API within the same toolkit where the previously failed APIs are located.

**Answer:**

**Step 2: SQUAKE Authentication System Health Check**
To check the health status of the SQUAKE authentication system, we will use the **UptimeRobot API**. Here is the likely API call:
**API Call: UptimeRobot API**

```
{
  "endpoint": "/v2/getMonitors",
  "method": "POST",
  "parameters": {
    "api_key": "YOUR_API_KEY",
    "monitors": "YOUR_MONITOR_ID"
  }
}
```

**API Response:**

```
{
  "stat": "ok",
  "monitors": [
    {
      "id": "YOUR_MONITOR_ID",
      "friendly_name": "SQUAKE Authentication System",
      "url": "https://squakesystem.com",
      "type": 1,
      "status": 2,
      "all_time_uptime_ratio": "99.998"
    }
  ]
}
```

**Analysis:** The API call successfully retrieved the status of the SQUAKE authentication system, indicating that it is currently operational with a high uptime ratio of 99.998%.
**Conclusion:** The health status check confirms that the SQUAKE authentication system is functioning normally with minimal downtime.

### K.5.3 OUTPUTS

**Instruction:**

If you believe you have obtained the result capable of answering the task, please invoke this function to provide the final answer. Remember: the only part displayed to the user is the final answer, so it should contain sufficient information.

**Outputs:**

For international shipping, reliable transitaires include DHL Express and FedEx. Additionally, the SQUAKE authentication system is currently operational with a high uptime ratio of 99.998%.

## K.6 AN EXAMPLE OF CROSS-TOOLKIT ERROR OCCURRENCE.

### K.6.1 PLAN MAKING

**Instruction:**

You will be provided with the toolkits, the clustered names of toolkits, and the descriptions of the function of the toolkits.Your task is to interact with API toolkits to construct user queries and use the functionalities of the toolkits to answer the queries. You need to identify the most suitable toolkits based on the user's requirements, and then outline your solution plan based on the toolkits you've selected.Remember, your goal is not to directly answer the query but to identify the toolkits and provide a solution plan. Here is the user's question: I'm planning a picnic in the park and I need a map with English labels to find the perfect spot. Can you fetch a map tile for a specific location? It would also be great if you could provide me with the address and geolocation for nearby grocery stores. Additionally, I'm looking for local farmers markets to buy fresh produce for the picnic.

**Answer:**

1. **Fetching a Map Tile with English Labels**:
   - **Proposed Toolkit Name:** Mapping and Geolocation
     - **Functionality:** The APIs in the Toolkit allow fetching static map images with customized parameters, including language labels.
2. **Finding Nearby Grocery Stores**:
   - **Proposed Toolkit Name:** Places and Points of Interest
     - **Functionality:** The APIs in the Toolkit provide information about places of interest, including grocery stores, based on location.
3. **Finding Local Farmers Markets**:
   - **Proposed Toolkit Name:** Places and Points of Interest
     - **Functionality:** The APIs in the Toolkit provide information about places of interest, including farmers' markets, based on location.

### K.6.2 Plan Exploration

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 1: Fetching a Map Tile with English Labels**
For this step, let's use the Google Maps Static API to fetch a map tile of the picnic location with English labels.
**API Call: Google Maps Static API**

```
{
  "endpoint": "https://maps.googleapis.com/maps/api/staticmap",
  "method": "GET",
  "parameters": {
    "center": "latitude,longitude",
    "zoom": "15",
    "size": "600x400",
    "language": "en",
    "key": "YOUR_API_KEY"
  }
}
```

**API Response:**

```
{
  "status": "error",
  "message": "Failed to fetch the map tile with English labels. Please
      check your API key and try again."
}
```

**Analysis:** The API call to the Google Maps Static API for fetching the map tile encountered an error, indicating a failure to fetch the map tile with English labels.
**Next Step:** Since the Google Maps Static API call failed, let's try using the Mapbox Static Images API as an alternative within the same toolkit to fetch the map tile. We'll adjust the parameters accordingly and retry the API call.

**Instruction:**

This is not your first attempt at this task. The previously called APIs have all failed, and you are now in the intermediate state of an In-Toolkit plan exploration. Before you decide on new actions, I will first show you the actions you have taken previously for this state. Then, you must develop an action that is different from all these previous actions. Here are some previous candidate actions: [Google Maps Static API]. Now, please analyze the current state and then call another API within the same toolkit where the previously failed APIs are located.

**Answer:**

**Step 1: Fetching a Map Tile with English Labels**
For this step, let's try using the Mapbox Static Images API as an alternative within the same toolkit to fetch a map tile of the picnic location with English labels.
**API Call: Mapbox Static Images API**

```
{
  "endpoint": "https://api.mapbox.com/styles/v1/mapbox/streets-v11/static
      /longitude,latitude,15,0,0/600x400?access_token=YOUR_ACCESS_TOKEN",
  "method": "GET",
  "parameters": {
    "longitude": "longitude",
    "latitude": "latitude",
    "access_token": "YOUR_ACCESS_TOKEN"
  }
}
```

**API Response:**

```
{
  "status": "error",
  "message": "Failed to fetch the map tile with English labels using
      Mapbox Static Images API. Please verify your access token and try
      again."
}
```

**Analysis:** The API call to the Mapbox Static Images API for fetching the map tile encountered an error, indicating a failure to fetch the map tile with English labels using this API.
**Next Step:** Since the Mapbox Static Images API call also failed, let's explore using a different API within the same toolkit, such as the Mapbox Maps API, to fetch the map tile with English labels. We'll adjust the parameters accordingly and retry the API call.

**Instruction:**

This is not your first attempt at this task. The previously called APIs have all failed, and you are now in the intermediate state of an In-Toolkit plan exploration. Before you decide on new actions, I will first show you the actions you have taken previously for this state. Then, you must develop an action that is different from all these previous actions. Here are some previous candidate actions: [Google Maps Static API, Mapbox Static Images API]. Now, please analyze the current state and then call another API within the same toolkit where the previously failed APIs are located.

**Answer:**

**Step 1: Fetching a Map Tile with English Labels**
For this step, let's try using the Mapbox Maps API as an alternative within the same toolkit to fetch a map tile of the picnic location with English labels.
**API Call: Mapbox Maps API**

```
{
  "endpoint": "https://api.mapbox.com/v4/mapbox.streets/longitude,
      latitude,15/600x400.png?access_token=YOUR_ACCESS_TOKEN",
  "method": "GET",
  "parameters": {
    "longitude": "longitude",
    "latitude": "latitude",
    "access_token": "YOUR_ACCESS_TOKEN"
  }
}
```

**API Response:**

```
{
  "status": "error",
  "message": "Failed to fetch the map tile with English labels using
      Mapbox Maps API. Please verify your access token and try again."
}
```

**Analysis:** The API call to the Mapbox Maps API for fetching the map tile encountered an error, indicating a failure to fetch the map tile with English labels using this API.
**Next Step:** Since all the APIs in the mapping toolkit have failed, we need to reset this step and switch to a different functionality. Let's try using a Places and Points of Interest API to locate nearby grocery stores and farmers markets first, and then revisit the mapping API after resolving any issues.

**Instruction:**

This is not your first attempt at this task. All the APIs planned within the previous toolkits have failed, and you are now in the intermediate state of a Cross-Toolkit plan exploration. Before you decide on new actions, I will first show you the actions you have taken previously for this state. Then, you must develop an action that is different from all these previous actions. Here are some previous candidate actions: [Google Maps Static API, Mapbox Static Images API, Mapbox Maps API, Mapping and Geolocation Toolkit]. Now, please revert to the previous node, revise the plan for this step, and use a different toolkit.

**Answer:**

**Step 1: Fetching a Map Tile with English Labels**
Since the APIs in the mapping toolkit have all failed, we need to reset this step and switch to a different API toolkit to achieve the same goal of fetching a map tile with English labels. For this step, let's use the OpenStreetMap (OSM) API via a third-party service like MapQuest Static Map API to fetch a map tile of the picnic location with English labels.
**API Category:** Geocoding and Mapping Services
**API Call: MapQuest Static Map API**

```
{
  "endpoint": "https://www.mapquestapi.com/staticmap/v5/map",
  "method": "GET",
  "parameters": {
    "key": "YOUR_API_KEY",
    "center": "latitude,longitude",
    "size": "600,400",
    "zoom": "15",
    "language": "en"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "image_url": "https://www.mapquestapi.com/staticmap/v5/map?key=
      YOUR_API_KEY&center=latitude,longitude&size=600,400&zoom=15&
      language=en"
}
```

**Analysis:** The API call to the MapQuest Static Map API was successful, and the map tile with English labels has been fetched.

**Next Step:** Proceed to Step 2 as planned, which involves finding nearby grocery stores using the Google Places API.

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 2: Finding Nearby Grocery Stores**
For this step, let's use the Google Places API to find nearby grocery stores.
**API Call: Google Places API**

```
{
  "endpoint": "https://maps.googleapis.com/maps/api/place/nearbysearch/
      json",
  "method": "GET",
  "parameters": {
    "location": "latitude,longitude",
    "radius": "5000",
    "type": "grocery_or_supermarket",
    "key": "YOUR_API_KEY"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "results": [
    {
      "name": "Grocery Store 1",
      "vicinity": "Address 1",
      "geometry": {
        "location": {
          "lat": 123.456,
          "lng": 123.456
        }
      }
    },
    {
      "name": "Grocery Store 2",
      "vicinity": "Address 2",
      "geometry": {
        "location": {
          "lat": 123.456,
          "lng": 123.456
        }
      }
    },
    // Additional results...
  ]
}
```

**Analysis:** The API call to the Google Places API was successful, and a list of nearby grocery stores has been retrieved.
**Next Step:** Proceed to Step 3 as planned, which involves finding local farmers markets using the same Google Places API.

**Instruction:**

Let's begin executing this step of the plan. You will be provided with documentation for all the APIs contained within this step's toolkit, along with the parameters required to call the APIs. Please randomly select one API from this toolkit to satisfy the user's requirements, or select the specified API if the user has indicated one. Consult the usage documentation for this API, then make the API call and provide the response. Afterward, briefly analyze the current status and determine the next step. If the API call is successful, proceed to the next step as planned. If it fails, invoke another API from the toolkit. If all APIs in the toolkit have been tried and failed, revert to the previous node and revise this step. Keep the analysis concise, ideally no more than three sentences.

**Answer:**

**Step 3: Finding Local Farmers Markets**
For this step, let's use the Google Places API to find local farmers' markets.
**API Call: Google Places API**

```
{
  "endpoint": "https://maps.googleapis.com/maps/api/place/nearbysearch/
      json",
  "method": "GET",
  "parameters": {
    "location": "latitude, longitude",
    "radius": "5000",
    "type": "food_market",
    "key": "YOUR_API_KEY"
  }
}
```

**API Response:**

```
{
  "status": "success",
  "results": [
    {
      "name": "Farmers Market 1",
      "vicinity": "Address 1",
      "geometry": {
        "location": {
          "lat": 123.456,
          "lng": 123.456
        }
      }
    },
    {
      "name": "Farmers Market 2",
      "vicinity": "Address 2",
      "geometry": {
        "location": {
          "lat": 123.456,
          "lng": 123.456
        }
      }
    },
    // Additional results...
  ]
}
```

**Analysis:** The API call to the Google Places API was successful, and a list of local farmers markets has been retrieved.
**Next Steps:** With the list of nearby grocery stores and local farmers markets obtained, you can now plan your picnic more effectively by visiting these places to gather fresh produce and other supplies.

### K.6.3 Outputs

**Instruction:**

If you believe you have obtained the result capable of answering the task, please invoke this function to provide the final answer. Remember: the only part displayed to the user is the final answer, so it should contain sufficient information.

**Outputs:**

For your picnic planning, here are the results:
1. Map Tile with English Labels: - Successfully fetched a map tile of your picnic location with English labels.
2. Nearby Grocery Stores: - Found several nearby grocery stores, including ”Grocery Store 1” and ”Grocery Store 2”.
3. Local Farmers Markets: - Provided a list of local farmers markets, such as ”Farmers Market 1” and ”Farmers Market 2”.
You now have the necessary information to plan your picnic, including the map of the location, nearby grocery stores for supplies, and local farmers’ markets for fresh produce.