SwissNYF: Tool Grounded LLM Agents for Black Box Setting

Somnath Sendhil Kumar[1] **Dhruv Jain**[1] **Eshaan Agarwal**[1] **Raunak Pandey**[1]

[1] Intelligence Group, IIT (BHU), Varanasi

Abstract

While Large Language Models (LLMs) have demonstrated enhanced capabilities in function-calling, these advancements primarily rely on accessing the functions' responses. This methodology is practical for simpler APIs but faces scalability issues with irreversible APIs that significantly impact the system, such as a database deletion API. Similarly, processes requiring extensive time for each API call and those necessitating forward planning, like automated action pipelines, present complex challenges. Furthermore, scenarios often arise where a generalized approach is needed because algorithms lack direct access to the specific implementations of these functions or secrets to use them. Traditional tool planning methods are inadequate in these cases, compelling the need to operate within black-box environments. Unlike their performance in tool manipulation, LLMs excel in black-box tasks, such as program synthesis. Therefore, we harness the program synthesis capabilities of LLMs to strategize tool usage in black-box settings, ensuring solutions are verified prior to implementation. We introduce **TOPGUN**, an ingeniously crafted approach leveraging program synthesis for black box tool planning. Accompanied by **SwissNYF**, a comprehensive suite that integrates black-box algorithms for planning and verification tasks, addressing the aforementioned challenges and enhancing the versatility and effectiveness of LLMs in complex API interactions. The public code for SwissNYF is available at https://github.com/iclr-dummy-user/SwissNYF

1 Introduction

Significant advancements in Large Language Models (LLMs) like GPT (Radford et al. (2018); Radford et al. (2019); Brown et al. (2020); Achiam et al. (2023)) and PaLM (Chowdhery et al. (2023);Anil et al. (2023);) have demonstrated profound abilities in reasoning and following instructions over an extensive array of tasks Huang & Chang (2023). The recent shift towards leveraging LLMs to interact with external tools for addressing complex real-world challenges marks a significant area of interest (Hao et al. (2023); Zhang et al. (2023a); Zhuang et al. (2023b); Yang et al. (2023); Schick et al. (2023);Lu et al. (2023a);). In addressing intricate problems, autonomous agents powered by LLMs employ an amalgamation of LLMs and various external tools (APIs), crafting solutions that

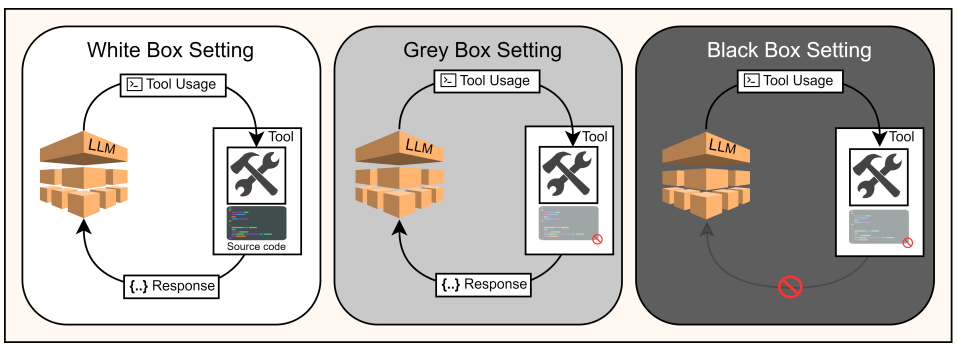

Figure 1: Illustration of different settings that an LLMs may require to manipulate tools.

necessitate a sequence of intermediate reasoning steps (Schick et al. (2023);Lu et al. (2023a);Lu et al. (2023a);Patil et al. (2023);Qin et al. (2023)). When presented with a problem, These agents' primary objective is to identify and execute a series of API function calls sequentially, leading to a coherent solution. These approaches are ineffective when queries lack transparency or when the APIs are irreversible.

We coin the term "black-box" settings in the context of tool planning as scenarios where the outcomes of an API or tool are not observable. This framework is especially pertinent in systems where using certain APIs poses risks, such as those causing inconsistencies by deleting or updating database entries, canceling jobs, or performing similar operations. It's also relevant where API experimentation incurs high costs or when APIs require considerable time to execute, ensuring clarity and comprehensive coverage without redundancy, making it challenging to interpret their outcomes. We present a taxonomy of such systems Fig. 1 into three branches:

1. **White Box Systems**: In these settings, planners can invoke the API, receive responses, access the source code and understand its complex logic. This access enables the system to navigate complex inputs, intricacies and use cases efficiently.
2. **Gray Box Systems**: Planners in these environments have descriptions of the tools at their disposal and the capability to call the API and receive responses. The system's planning relies solely on the limited descriptions provided and the responses for each tool.
3. **Black Box Systems**: In the most challenging scenarios, planners are confined to tool descriptions without access to actual tool outputs. Here, the planner must decipher the dynamics of each tool based solely on its description, making it a particularly demanding task to formulate responses to queries.

The Zhuang et al. (2023a) and Qin et al. (2023) methods excel in straightforward scenarios where an agent can iterate over tools to identify the optimal path, yet they lack efficiency and necessitate extensive exploration. Approaches like Yao et al. (2022) and Parisi et al. (2022), subsets of this exploratory paradigm, offer enhanced efficiency yet frequently falter due to their constrained directionality in tool search, making them suitable predominantly for straightforward API challenges. In contrast, the Zhang et al. (2023b) approach is efficient regarding API execution costs by constraining the number of calls. However, it omits any form of verification for its proposed trajectory, diminishing its precision in practical applications.

These methodologies in tool application present a dichotomy between accuracy and computational overhead. While generally unsuitable for black-box settings, the Reverse chain approach exhibits potential for adaptation within such frameworks. On the other hand, program synthesis-based algorithms have been instrumental in exalting reasoning and decision-making capabilities within LLMs, offering a more naturally associative decision-making process than that afforded by mere text. Works like The Chain of Code Li et al. (2023) and Program-of-thoughts Chen et al. (2022) are great examples of using code generation to improve decision-making for answering general open-domain questions. To this end, few works also upheld the reasoning capability of LLMs using code like "TORA: A Tool-Integrated Reasoning Agent for Mathematical Problem Solving" Gou et al. (2023), "Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification" Zhou et al. (2023b) and "PAL: Program-aided Language Models" Gao et al. (2023) have exploited code interpreters for zero-shot verified solving, substantially surpassing few-shot learning benchmarks by enabling semi-verification of proposed solutions.

However, works like Paranjape et al. (2023), which employs code synthesis for tool usage, are restricted by their limited toolset and the scalability challenge posed by the need for extensive human feedback and interventions and the need for the human expert to be familiar with the whole toolset. Similarly, works such as Xu et al. (2023), which deploys language models for real-time code generation and command execution within controlled environments, are limited by their narrow tool range and a deficit in generalizability. The state-of-the-art approaches on HumanEval Chen et al. (2021) and HumanEval-X Zheng et al. (2023) datasets for code generation, like Reflexion Shinn et al. (2023) and LATS Zhou et al. (2023a), which iterate upon code based on interpreter outputs and reflect over them, these approaches have yet to be experimented with in other domains associated with LLMs.

To bridge these gaps, we introduce the **TOPGUN** (**T**ool **O**rchestration and **P**rogram synthesis for **G**eneralizing over **UN**known systems) framework, which unifies code generation, reasoning, and

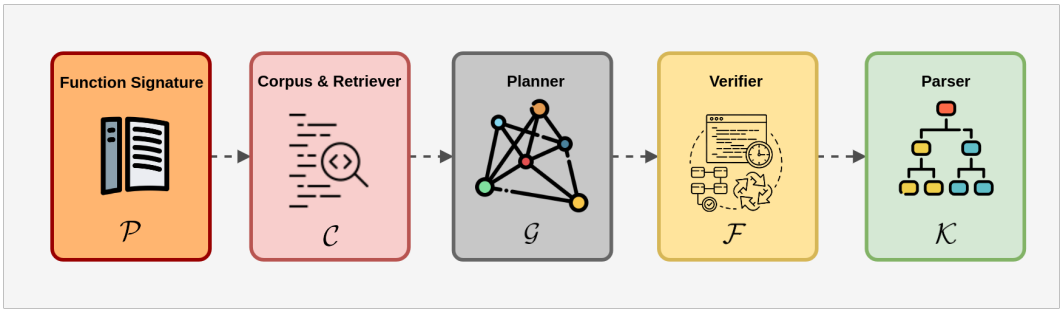

Figure 2: Illustration of SwissNYF pipeline for tool usage in Black Box setting.

strategic tool planning designed for complex tasks. TOPGUN also verifies the execution plans and does so with exceptional efficiency in API cost, effectively addressing the limitations of preceding models.

Key contributions of our work are summarized as follows:

1. To the best of our knowledge, we are the First to coin the term Black Box setting for API usage and developed a suite to encourage the development of algorithms for such scenarios.
2. We leverage the program synthesis capabilities of Large Language Models (LLMs) to augment their efficacy in tool usage substantially, showcasing a notable enhancement in performance.
3. We present a robust and cost-efficient framework for scalable solutions across a wide array of open-domain queries, even when faced with limited knowledge of user data/tools. It is also publically hosted to demonstrate the same.[1]

This paper details our methodology and its evaluation by first elucidating the background on Tool planning 2.1 and Code generation using LLM 2.2 followed by detailing individual components of the pipeline 3. Our evaluation is bifurcated into two segments: initially, we undertake a gray box 4.1 across principal datasets, and subsequently, we delve into a black box setting 4.2. For the latter, we have curated a bespoke dataset employing Toolbench prompts, intentionally adjusting the dataset to include only limited documentation of widely used libraries. This adjustment aims to validate the generalizability of our approach. Additionally, we juxtapose our methodology with a tailored variant of the Reverse Chain method to scrutinize performance disparities.

2 Preliminaries

2.1 Problem Formulation

Tool planning within the context of a Large Language Model (LLM), denoted as ρ, involves leveraging a selection of tools from a pool of n candidate tools in the corpus $\mathcal{C}$, represented as $\mathcal{C} = \{t_0, t_1, \ldots, t_n\}$, to effectively address a user's query q. The primary goal is to formulate a meticulous plan, known as the Solution Trajectory St, for the orchestration of these tools. The Solution Trajectory St, which outlines the sequential execution of tools, is crafted to directly address the query q. The LLM agent, or planner $\mathcal{G}$, is responsible for planning or generating St from $\mathcal{C}$, formalized as $St \leftarrow \mathcal{G}(q, \rho, \mathcal{C})$. This process ensures a structured and coherent response strategy, aligning the tools' capabilities with the query's specific requirements for an effective solution.

2.2 Code Generation

The integration of Reflexion Shinn et al. (2023) with Large Language Models (LLM) ρ and Python Interpreter $\mathcal{I}$ has significantly advanced coding tasks by enabling iterative code refinement. This approach leverages feedback $\mathcal{F}$ to iteratively address exceptions and enhance initial code output c, guided by test cases dynamically generated by ρ itself. This ensures comprehensive verification and

[1] https://swiss-nyf.azurewebsites.net/

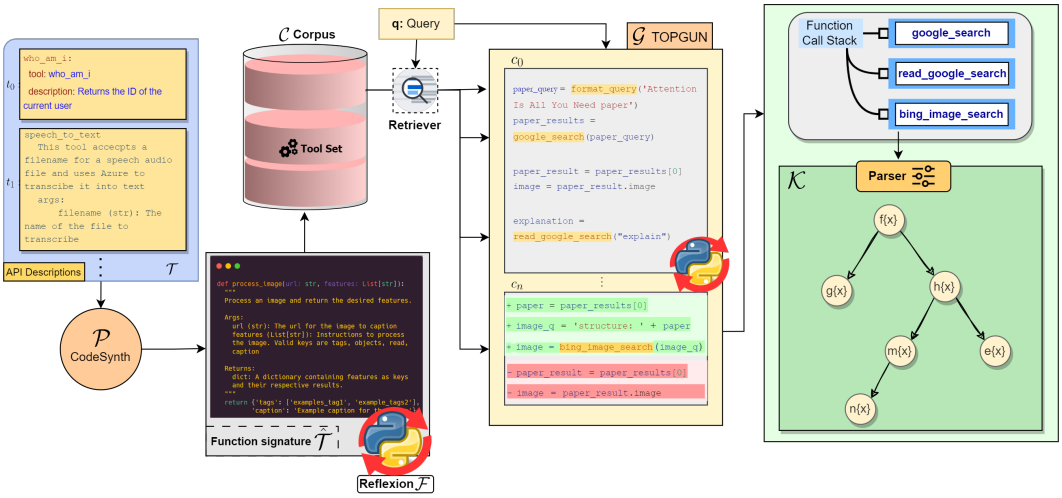

Figure 3: Detailed pipeline of our proposed approach with TOPGUN in SwissNYF

refinement within a Function Call module, leading to a finalized code c_n. This methodology enhances code quality and aligns with contemporary standards, marking a leap in automated code development and verification. This process of iterative code generation can be mathematically denoted as Eq. 1

$$\begin{aligned} c_i &\leftarrow \rho(q, feedback_{i-1}, c_{i-1}) \\ output &\leftarrow \mathcal{I}(c_i) \\ feedback_i, verified &\leftarrow \mathcal{F}(output) \end{aligned} \tag{1}$$

3 SwissNYF

3.1 Overview

In this section, we introduce SwissNYF, a suite that enables LLM-based agents to efficiently navigate the action space to identify a valid solution for problem-solving in a black box scenario. SwissNYF is composed of five major components i.e., Function Signature Generation $\mathcal{P}$, Corpus & Retriever $\mathcal{C}$, Planner $\mathcal{G}$, Verifier $\mathcal{F}$ and Parser $\mathcal{K}$ as in Fig. 2. We explain individual components of the pipeline in the subsequent subsections.

3.2 Function Signature Generation

Function signatures, conceptualized as pseudo APIs, serve to emulate the behaviour of real API functions based on given tool descriptions. This emulation is crucial for two primary reasons in our tool planning methodology: firstly, they act as stand-ins for actual API calls, thereby enabling LLMs to plan and execute tasks with higher efficiency; secondly, they are treated as pre-defined functions, facilitating the transformation of tool augmentation into a task akin to code generation, using these pseudo functions. These function signatures are distinguished by their docstrings and an example return object that aligns with the tool description, equipping the planner with the necessary means to effectively address user queries. In the context of our SwissNYF implementation, we have adopted a straightforward yet effective method for generating these function signatures, termed **CodeSynth**. The efficacy of this approach is further analyzed in 4.3.

3.2.1 CodeSynth

For a given set of tool descriptions $t \in \mathcal{T}$, we direct the Large Language Model (LLM) ρ to generate pseudo-function implementations, denoted as $\hat{t}$. Our primary objective is to ensure that the arguments and return types of these pseudo-functions remain consistent with their descriptions. Additionally, we craft detailed docstrings for each pseudo-function to facilitate subsequent processes. A critical aspect of CodeSynth is the inclusion of an example return value, which is designed to mimic all potential operations the returned object might undergo during the verification process. The output generated by

```
N: bing_video_search
D: Make a query to bing video search. Useful for finding
   a video related to a query.
   query (str): The query to be passed to bing.
```

```python
def bing_video_search(query: str) -> str:
    """
    Make a query to bing video search. Useful for
    finding a video related to a query.
    Args:
    query (str): The query to be passed to bing.
    Returns:
        str: A dummy named example that matches with the
             text description.
    """
    return ["https://www.example.com/video1",
            "https://www.example.com/video2"]
```

(a) Example output of CodeSynth $\mathcal{P}$ Algorithm

```
Q: Extract the picture of the menu of the famous Paradise restaurant
in JP Nagar, Bangalore, and tell me how many dishes have a price of
more than Rs.300 among the non-vegetarian starters.
```

```python
# First, we need to find the image of the menu
image_url = bing_image_search('menu of Paradise restaurant in
                               JP Nagar, Bangalore')
# Next, we need to process the image to read the text
menu_text = process_image(image_url, ['read'])
# Now, we need to find the non-vegetarian starters and the prices
import re
non_veg_starters = re.findall(r'Non-Vegetarian Starters\n(.*?)\n',
   menu_text, re.DOTALL)
# Check if non_veg_starters is not empty
if non_veg_starters:
    prices = re.findall(r'\d+', non_veg_starters[0])
    # Now, We need to count the dishes with price more than 300
    count = 0
    for price in prices:
        if int(price) > 300:
            count += 1
            print(f'There are {count} dishes with a price more than
                  Rs.300 among the non-vegetarian starters.')
else:
    print("No non-vegetarian starters found in the menu.")
```

(b) Example output of TOPGUN $\mathcal{G}$ Algorithm

Figure 4: Illustration of pseudo function and tool planning generated by CodeSynth and TOPGUN, respectively.

CodeSynth is illustrated in Fig. 4a. Moreover, the code generation facilitated by this block benefits from validation through Reflexion, as outlined in Eq. 1. Ultimately, the methodologies applied within CodeSynth can be encapsulated in Algo. 1.

Utilizing the Function Calling module alongside the Interpreter, we rigorously test the pseudo-functions against a wide range of real-world scenarios. This approach guarantees that the test cases are comprehensive and reflective of actual function usage, allowing us to gather detailed feedback on the pseudo-functions' performance. Such feedback is vital for the iterative improvement of the pseudo-functions, significantly enhancing their reliability and applicability in practical settings. Prompts for CodeSynth can be documented in A.1.

Algorithm 1: $\mathcal{P}$: CodeSynth

Input: ρ: large language model; T: tool descriptions; $\mathcal{I}$: python interpreter; $\mathcal{F}(\mathcal{I})$: reflexion feedback of $\mathcal{I}$; $\mathcal{C}$: empty corpus of pseudo tools

for $t = 1, 2, \cdots, T$ **do**

- $\hat{t}_0 \leftarrow \rho(t)$ *//Pseudo code*
- $verified \leftarrow \mathcal{I}(\hat{t}_0)$
- **while** *not verified* **do**
  - $\hat{t}_i \leftarrow \rho(t, feedback_{i-1}, \hat{t}_{i-1})$
  - $feedback_i,\ verified \leftarrow \mathcal{F}(\mathcal{I}(\hat{t}_i))$
- Update $\mathcal{C} \leftarrow \hat{t}_n$ *// Update Corpus*

Output: A corpus of verified psuedo functions $\mathcal{C}$

3.3 Corpus and Retriever

The function signatures, crucial components of our methodology, are systematically stored within a corpus for future utilization by any planning system. This corpus facilitates the indexing of tool descriptions, enabling the precise retrieval of the most appropriate tool based on the index. Notably, the literature documents several advanced retrieval systems designed for this purpose, demonstrating exceptional accuracy. These include ToolBench IR Qin et al. (2023), APIRetriever Zan et al. (2022), Instructor-XL Su et al. (2022), and GEAR Lu et al. (2023b). Our framework incorporates these retrievers, with Instructor-XL set as the default option, owing to its proven efficacy. Furthermore, we are actively exploring the integration of AnyTool's Hierarchical API Retriever Du et al. (2024), anticipating significant enhancements to our tool retrieval capabilities. This strategic inclusion of multiple retrievers ensures our system remains versatile and effective in identifying the most suitable tools for a given task, aligning with the latest advancements in retrieval technology.

3.4 Planner

We have implemented two planning approaches in our framework. The first leverages a modified Reverse Chain Zhang et al. (2023b) to support multiple end function calls by decomposing tasks into subtasks and creating sub-trees with the original reverse chain technique. The second, **TOPGUN**, is our proposed code-driven planning algorithm, designed for speed, efficiency, consistency, and

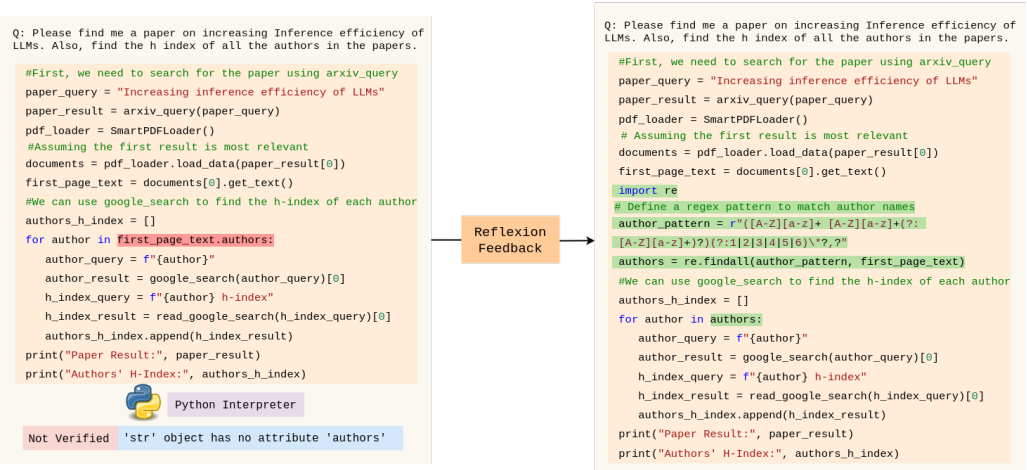

Figure 5: Illustartion of Self-Reflection Mechanism in TOPGUN

accuracy, especially in black box scenarios. **TOPGUN** offers a streamlined alternative to traditional planning methods, optimizing for complex system navigation and task execution with greater reliability and cost-effectiveness.

3.4.1 TOPGUN

TOPGUN, an acronym for **T**ool **O**rchestration and **P**rogram synthesis for **G**eneralizing over **UN**known systems, redefines the approach to addressing user queries q by framing the challenge as a task of code generation. Utilizing pseudo-functions $\hat{\mathcal{T}}$ as functions available to TOPGUN enables the agent to construct an accurate sequence of function calls $c_0 \leftarrow \rho(q, \hat{\mathcal{T}}, \mathcal{C})$, effectively depicted in Fig. 4b. Leveraging Reflexion detailed in Eq.1, the framework iteratively refines responses to the query. The synthesis of these components into the comprehensive algorithm is presented in Algo. 2 showcases TOPGUN's capability to navigate through various solution paths. Unlike traditional traversal-based techniques, TOPGUN capitalizes on the inherent code-generation capabilities of LLMs, facilitating a more direct and efficient solution process. This distinction not only enhances efficacy by pinpointing issues with precision but also ensures adaptability in black box scenarios, simultaneously optimizing performance in gray box settings. A detailed pipeline overview with TOPGUN in place is given in Fig.4b. With prompts documented in A.1.

Algorithm 2: $\mathcal{G}$: TOPGUN

Input: q: query; ρ: large language model; T: tool descriptions; $\mathcal{I}$: python interpreter; $\mathcal{F}(\mathcal{I})$: reflexion feedback of $\mathcal{I}$; $\mathcal{C}$: empty corpus of pseudo tools; $\mathcal{P}$: Codesynth, $\mathcal{K}$: parser

Initialize $\hat{\mathcal{T}} \leftarrow \mathcal{P}(\rho, T, \mathcal{I}, \mathcal{F}, \mathcal{C})$ *// Pseudo tools*

$c_0 \leftarrow \rho(q, \hat{\mathcal{T}}, \mathcal{C})$ *// Code for query*

$verified \leftarrow \mathcal{I}(c_0, \hat{\mathcal{T}})$ *// Verify with pseudo tools*

while *not verified* **do**

  $c_i \leftarrow \rho(q, \hat{\mathcal{T}}, \hat{\mathcal{C}}, feedback_{i-1}, c_{i-1})$

  $feedback_i,\ verified \leftarrow \mathcal{F}(\mathcal{I}(c_i, \hat{\mathcal{T}}))$

$\mathcal{S}t \leftarrow \mathcal{K}(c_n)$ *// Solution Trajectory*

Output: A solution trajectory $\mathcal{S}t$ and c_n code for execution and evaluation

3.5 VERIFIER

Verification is closely linked to the functionality of the Planner $\mathcal{G}$, relying on both the nature of $\mathcal{G}$'s output and its ability to incorporate feedback. Although verification initially serves as a preparatory step prior to parsing, it also plays a crucial role in refining outputs by providing feedback that $\mathcal{G}$ can use for subsequent iterations.

In our framework, we leverage Reflexion Shinn et al. (2023), detailed in Eq. 1 and depicted in Algo. 2, to seamlessly integrate verification and feedback within the TOPGUN methodology. This eliminates the requirement for an additional function call module, concentrating instead on directly executing code pertinent to the user query. This approach is illustrated in Fig. 5, providing a visual representation of the concept.

3.6 PARSER

The Parser $\mathcal{K}$, akin to the Verifier $\mathcal{F}$, is intrinsically dependent on the Planner $\mathcal{G}$ for its functionality. Its pivotal output is a well-defined Solution Trajectory St, mapping out the sequence of tool applications devised to address the query. In employing the Reverse Chain technique, our methodology involves synthesizing individual sub-trees into a singular, comprehensive tree through the capabilities of LLM ρ. The process's efficacy is markedly improved by the judicious reuse of elements from the individual trees during their amalgamation.

Conversely, for the **TOPGUN** methodology, we adopt the established Abstract Syntax Tree (AST) paradigm Fischer et al. (2007) to segment the program into fundamental function calls, alongside specifying their arguments and return values. This segmentation is instrumental in constructing a systematic series of tool invocations. This meticulously arranged series, denoted as St, is succinctly formalized as $St \leftarrow \mathcal{K}(c_n)$.

The entire pipeline, as depicted in Figure 3, emerges from the integration of various components designed to effectively address user queries through the strategic orchestration of tools within the SwissNYF framework.

Table 1: Win Rate of different Candidate and Reference model over G1 set

Candidate	Reference	G1-Instruction	G1-Tool	G1-Category
T.LLaMA ReACT	ChatGPT ReACT	45.0	42.0	47.5
T.LLaMA DFSDT	ChatGPT ReACT	55.0	55.3	54.5
T.LLaMA DFSDT+Ret	ChatGPT ReACT	62.3	59.0	55.0
ChatGPT DFSDT	ChatGPT ReACT	60.5	62.0	57.3
GPT4 ReACT	ChatGPT ReACT	60.0	58.8	63.5
GPT4 DFSDT	ChatGPT ReACT	67.5	67.8	66.5
GPT4 TOPGUN	ChatGPT ReACT	**88.192**	**87.46**	**87.15**
GPT4 TOPGUN	ChatGPT DFSDT	78.49	77.55	76.24
GPT4 TOPGUN	T.LLaMA ReACT	86.72	82.94	80.80
GPT4 TOPGUN	T.LLaMA DFSDT	81.75	75.51	73.81
GPT4 TOPGUN	T.LLaMA DFSDT+Ret	80.35	77.11	75.39
GPT4 TOPGUN	GPT4 ReACT	82.996	79.956	77.633
GPT4 TOPGUN	GPT4 DFSDT	**82.065**	**73.69**	**71.14**

4 EXPERIMENTS

Tool planning datasets, while diverse, often fall short in supporting multi-turn and multi-call dialogues, as seen in works by Schick et al. (2023) and Tang et al. (2023), and lack precise evaluation metrics, complicating thorough assessments. Even comprehensive datasets like ToolBench by Qin et al. (2023) struggle with aligning to black-box settings, presenting significant challenges for evaluating tool planning in such scenarios.

Our evaluation employs the ToolBench benchmark Qin et al. (2023) and a specially curated dataset for unchar codebases, assessed in both gray (4.1) and black box (4.2) settings. We benchmark our TOPGUN approach against existing methods using win rate, token count, and success rate. Additionally, we scrutinize CodeSynth's ($\mathcal{P}$) impact on the Planner's ($\mathcal{G}$) performance and independently evaluate its ability to generate effective function signatures, acting as pseudo functions, detailed in Section 4.3.

4.1 GRAY BOX EVALUATION

To assess the performance of **TOPGUN** and compare it with other gray box methodologies such as ReACT and DFSDT, we maintain the integrity of our pipeline while adapting the evaluation process to incorporate actual functions in place of pseudo functions within the output solution trajectory. This approach effectively leaves our black box pipeline intact while converting it into a gray box

Table 2: Win Rate of different Candidate and Reference model over G2, G3 set and Average over all sets

Candidate	Reference	G2-Instruction	G2-Category	G3-Instruction	Average
T.LLaMA ReACT	ChatGPT ReACT	50.8	41.8	55.0	47.0
T.LLaMA DFSDT	ChatGPT ReACT	68.5	58.0	69.0	60.0
T.LLaMA DFSDT+Ret	ChatGPT ReACT	68.5	60.8	73.0	63.1
ChatGPT DFSDT	ChatGPT ReACT	72.0	64.8	69.0	64.3
GPT4 ReACT	ChatGPT ReACT	65.8	60.3	78.0	64.0
GPT4 DFSDT	ChatGPT ReACT	73.3	63.3	84.0	70.4
GPT4 TOPGUN	ChatGPT ReACT	**87.59**	**78.78**	**90.05**	**86.54**
GPT4 TOPGUN	ChatGPT DFSDT	81.63	73.07	85.26	78.71
GPT4 TOPGUN	T.LLaMA ReACT	86.24	77.71	93.23	84.61
GPT4 TOPGUN	T.LLaMA DFSDT	78.31	71.80	89.47	78.44
GPT4 TOPGUN	T.LLaMA DFSDT+Ret	83.07	72.92	87.82	79.44
GPT4 TOPGUN	GPT4 ReACT	78.61	73.75	93.68	**80.27**
GPT4 TOPGUN	GPT4 DFSDT	**73.92**	**71.35**	**79.25**	**78.59**

evaluation framework. The necessity of responses and Final answers for evaluation purposes has led us to adopt this hybrid strategy. In practical scenarios, this mirrors the process where a generalist planner delivers a strategy to the client, who then substitutes pseudo-function implementations with their real functions for execution. For this evaluation, we employ ToolBench, as detailed by Qin et al. (2023), and conduct our analysis across all problem categories provided in the dataset. Further elaboration on the precise evaluation methodology and the application of ToolBench is documented in A.2.

Results : Win Rate comparisons for ToolLLaMa-ReACT, ToolLLaMA-DFSDT, ChatGPT-DFSDT, GPT4-DFSDT, and GPT4-TOPGUN against ChatGPT-ReACT and GPT4-TOPGUN are summarized, with averages taken from 7 runs per model pair, detailed in Tables 1 and 2. TOPGUN significantly surpassed ReAct and DFSDT in all categories, achieving win rates of **80.27%** versus GPT4-ReACT, **78.59%** against GPT4-DFSDT, and **86.54%** against ChatGPT-ReACT, showing improvements of **22.54%** and **16.14%** respectively. These results highlight TOPGUN's superior ability to create tool plans that align with preference evaluation criteria across various conditions.

4.2 Black Box Evaluation

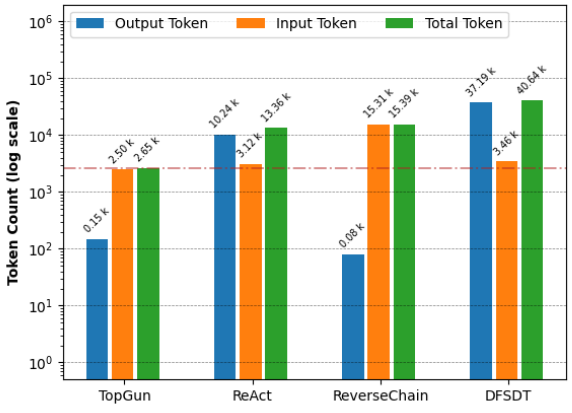

Figure 6: Average Token Consumption of individual methodologies in Black Box setting.

Utilizing the Data Generation pipeline from Qin et al. (2023), we constructed a black-box scenario dataset featuring 36 LLaMa-Hub LlamaIndex (2023) tools and unique functions from private libraries. Following Zan et al. (2022), we converted Pandas and Numpy into Monkey and BeatNum packages, renaming all internal functions and structures to test planner generalizability without LLM prior knowledge. This dataset, detailed at A.1, focuses on accuracy of the solution trajectory, with each query designed for a single correct path. After manual annotation, it comprises 100 queries and 162 tools, with samples and TOPGUN outcomes at A.3.2 and A.5.2.

Results : The black-box evaluation, featuring TOPGUN and a revised Reverse Chain, utilizes $\mathcal{P}$ function signatures for a comprehensive black-box methodology. TOPGUN surpasses Reverse Chain and undergoes comparison with GPT4-DFSDT and GPT4-ReACT within gray box

evaluations, emphasizing output trajectories. Success rates, derived from exact trajectory matches with the ground truth and averaged over ten iterations, are documented in Table 3. Figure 6 details the Average Token usage for each algorithm per query, underscoring TOPGUN's effectiveness and efficiency in generating precise and resourceful tool plans in black-box scenarios, demonstrating its adaptability across diverse datasets.

***Note**: A black-box evaluation using ToolBench is infeasible, as ToolEval's metrics, such as pass rate and win rate, rely on intermediate tool responses and the final answer.*

4.3 CODESYNTH EVALUATION

To assess the quality of function signatures produced by CodeSynth, we adopt neuro-symbolic representations, as proposed by Parisotto et al. (2017) and Nye et al. (2021). These representations aim to capture the abstract semantic essence of a given program, aligning well with our objectives. Our evaluation spans the Python subset of HumanEval-X Zheng et al. (2023) and MBPP Austin et al. (2021) dataset. Inspired by the semantic probing model introduced by Ma et al. (2023), we construct semantic representations of both synthesized pseudo functions and ground truth code. Utilizing the tree-sitter Brunsfeld et al. (2024) package, we form the Abstract Syntax Tree, focusing our computation of the F1 score exclusively on the Function Definition block while excluding the body block. Hence, the final metric is precisely representative of our objective with CodeSynth. The appendix A.4.1 can be referred to for function signature examples synthesized with the HumanEval-X dataset.

Results: We evaluate CodeSynth across multiple reflection cycles, tracking the F1 score for each cycle to illustrate consistent enhancements in function signature quality, as depicted in Table 4. CodeSynth significantly improved F1-scores on both HumanEval-X and MBPP datasets, achieving a perfect score of **1.0** by the fifth iteration from initial scores of **0.844** and **0.912**, respectively. These findings highlight CodeSynth's ability to produce function signatures closely resembling the semantics of the target function.

Method	Success Rate
GPT4-TOPGUN	**70.58**
GPT4-DFSDT	61.45
GPT4-ReAct	45.45
GPT4-ReverseChain	43.75

Table 3: Comparison of methodologies in Black Box Setting

Dataset	**F1 Score** for max Reflexion Iteration				
	@1	@2	@3	@4	@5
HumanEval-X	0.844	0.894	0.965	0.983	1.00
MBPP	0.912	0.963	0.994	1.00	1.00

Table 4: CodeSynth Evaluation for analyzing Reflexions improvement on Function Signature's AST

5 CONCLUSION

In this work, we address the challenge of tool planning in black-box settings, where direct access to API calls and their implementations is not feasible, raising concerns about cost efficiency and privacy in API interactions. We introduce SwissNYF, a comprehensive framework designed to equip Large Language Models (LLMs) with the ability to navigate these scenarios effectively. Central to SwissNYF is the ingenious function signature generation that allows the planner to rely on tool descriptions, circumventing the need for actual API executions. We further introduce TOPGUN, a code-driven planning approach leveraging LLMs' code generation capabilities to offer a robust solution for black-box environments. Our extensive evaluation across various toolsets and settings demonstrates the superior performance of our methodology against traditional tool planning strategies, validating its effectiveness and reliability. Through SwissNYF and TOPGUN, we establish an exciting and emerging paradigm in tool planning, We envision SwissNYF as a central hub for black-box tool usage, encouraging future advancements in developing strategies for black-box scenarios, thus making a significant leap towards efficient, privacy-conscious tool planning in the realm of LLM-enhanced applications.

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.

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

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.

Max Brunsfeld, Andrew Hlynskyi, Amaan Qureshi, Patrick Thomson, Josh Vera, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, dundargoc, Rob Rix, Daumantas Kavolis, Hendrik van Antwerpen, Michael Davis, Ika, Tuan-Anh Nguyen, Amin Yahyaabadi, Stafford Brunk, Matt Massicotte, and George Fraser. tree-sitter/tree-sitter: v0.21.0-pre-release-1, 2024. URL https://doi.org/10.5281/zenodo.10638807.

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

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

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.

Gregor Fischer, J Lusiardi, and J Wolff Von Gudenberg. Abstract syntax trees-and their role in model driven software development. In *International Conference on Software Engineering Advances (ICSEA 2007)*, pp. 38–38. IEEE, 2007.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pp. 10764–10799. PMLR, 2023.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Minlie Huang, Nan Duan, Weizhu Chen, et al. Tora: A tool-integrated reasoning agent for mathematical problem solving. *arXiv preprint arXiv:2309.17452*, 2023.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *arXiv preprint arXiv:2305.11554*, 2023.

Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 1049–1065, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.67. URL https://aclanthology.org/2023.findings-acl.67.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474*, 2023.

LlamaIndex. Llamahub, 2023. URL https://web.archive.org/web/20231229215448/https://llamahub.ai/.

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. *arXiv preprint arXiv:2304.09842*, 2023a.

Yining Lu, Haoping Yu, and Daniel Khashabi. Gear: Augmenting language models with generalizable and efficient tool resolution. *arXiv preprint arXiv:2307.08775*, 2023b.

Wei Ma, Mengjie Zhao, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jie Zhang, Wenhan Wang, and Yang Liu. Are code pre-trained models powerful to learn code syntax and semantics?, 2023.

Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B. Tenenbaum, and Armando Solar-Lezama. Representing partial programs with blended abstract semantics. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=mCtadqIxOJ.

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.

Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=rJ0JwFcex.

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.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

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

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A Smith, Luke Zettlemoyer, and Tao Yu. One embedder, any task: Instruction-finetuned text embeddings. *arXiv preprint arXiv:2212.09741*, 2022.

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

Yiheng Xu, Hongjin Su, Chen Xing, Boyu Mi, Qian Liu, Weijia Shi, Binyuan Hui, Fan Zhou, Yitao Liu, Tianbao Xie, et al. Lemur: Harmonizing natural language and code for language agents. *arXiv preprint arXiv:2310.06830*, 2023.

Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. Gpt4tools: Teaching large language model to use tools via self-instruction, 2023.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. When language model meets private library. *arXiv preprint arXiv:2210.17236*, 2022.

Beichen Zhang, Kun Zhou, Xilin Wei, Wayne Xin Zhao, Jing Sha, Shijin Wang, and Ji-Rong Wen. Evaluating and improving tool-augmented computation-intensive math reasoning, 2023a.

Yinger Zhang, Hui Cai, Yicheng Chen, Rui Sun, and Jing Zheng. Reverse chain: A generic-rule for llms to master multi-api planning. *arXiv preprint arXiv:2310.04474*, 2023b.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 5673–5684, 2023.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023a.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. *arXiv preprint arXiv:2308.07921*, 2023b.

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, 2023a.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. Toolqa: A dataset for llm question answering with external tools, 2023b.

A Appendix

A.1 Prompts

CodeSynth prompt for function signature generation

```
You are a Python code assistant that can generate a
pseudo-Python function given the name, description,
and arguments.

function name: {}
function description: {}

You have to generate a pseudo-Python function that only
contains docstring and a return example object for the
above-given information. Use dummy examples as return
objects.

Maintain the return datatype. Docsrting contains Args and
Returns. Maintain the arguments typing. The arguments are
optional and should be assigned relevant default values
according to their return type.

Only generate the def function itself as instructed above,
no typing imports or other code is needed.
```

TOPGUN prompt for code-based plan generation

```
You are a Python code assistant. Today, you are challenged
to generate a Python code for executing a query. You will
be given a list of pseudo functions that you will use in
your Python code to help you in solving the query correctly.

Understand the query properly and use the required
function to solve it.

We have the following pseudo functions:
=====
{}
=====

Let's start

If the query is {}
Return the python code to execute it with the help of given
functions. Do not use double quotes; only use single quotes.
Always have to the code within ```python\n<--Your Code-->\n```
Always remember if a function is to input or output an object
assume the object to be a string.
```

Function Call Prompt for verification

```
You are a Python code assistant. You are given a function.
For the given function, write an executable function call
using dummy argument values.

Provided Libraries: {}

Details of the provided library can only be fetched using
the query engine tool, feel free to use it.

-You can import the required classes from one of the provided
 libraries, according to the function arguments and documentation.
-If any library is not provided, ignore any imports.
-Do not import {} function for which you generate the
 function call.
-Do not generate any unnecessary import statements.
-No print statements are needed.
-Always have to code within ```python\n<--Your Code-->\n```

Example:

Given Function:
  def add(a: int, b: int) -> int:
      '''
      Given integers a and b,
      return the total value of a and b.
      '''
      return a + b

Function Call:
```

```
  a = 1
  b = 4
  add(a, b)

The function name is: {}
The function description is: {}
The Function is: {}
Function Call:
```

Self-Reflection Prompt

```
You are a Python code assistant. You will be given your last
Python code implementation, and an error in your last
implementation will be provided. Taking the error into
account, refactor your Python code.

Use the query engine to export the information needed
to resolve.

Always have to code within ```python\n<--Your Code-->\n```

Previous python code implementation: {}
Self-reflection: {}

Refactored Python code:
```

CodeSynth prompt for function signature generation on PrivateEval

```
You are a Python code assistant that can generate a pseudo
Python function given its name, description, and arguments.

function name: {}
function description: {}
Provided Libraries: {}

Always remember to import the required classes from one of the
provided library, according to the function arguments and the
provided documentation.

Documentation is to be fetched using the query engine tool.

If any library is not provided, ignore any imports.

The function arguments and returns are clearly defined in the
function description. Use as provided in the description.

You have to generate a pseudo-Python function that only contains
docstring and a dummy return object matching the actual return
datatype. No need to use the provided arguments. Just return a
dummy object that matches the actual return datatype of the
function.

Maintain the actual return datatype in the return object.
Docsrting contains Args and Returns. Maintain the arguments
typing.
```

```
Only generate the def function as instructed above; no typing
imports or other code is needed.

Always have to the code within ```python\n<--Your Code-->\n```

Pseudo Function:
```

TOPGUN prompt for code-based plan generation on ToolBench

```
You are a Python code assistant. Today, you are challenged
to generate a Python code for executing a query. You will
be given a list of pseudo functions that you will use in
your Python code to help you in solving the query correctly.
Understand the query properly and use the required function
to solve it.

We have the following pseudo functions:
=====
{}
=====

You have to make sure to follow the below guardrails:
 - Do not use double quotes; only use single quotes.
 - You are not allowed to define any functions; you must
   always use the given functions in the code.
 - If in case you end up creating a function, please
   rememeber to have a decorator named @update_traverse_dict
   on them.
 - Do not create a main function script and using
   'if __name__ == "__main__"' is strictly prohibited.
 - Always have to the code within ```python\n<--Your Code-->\n```
 - Always remember to use .get() to fetch values from a
   dictionary or a JSON.
 - Always remember to replace the values in .get() of the
   generated code with a value that matches the description of
   its key and dictionary whose argument it is. Use your
   world knowledge to replace the value with a
   good, real example.
   Example:
     contact = company_info.get('contact_number', '999991999')
     name = company_info.get('name', 'ryanair')

   Remember to Keep the values inside single quotes ' '.
 - This is also required when accessing the value of the list
   use try: except: and in except use a value that matches
   the description of the output.
 - Never use print statements. The user can use the variables
   in the code to infer the code.

You have to remember the following to solve the query:
 - Always remember if a function is to input or output an
   object assumes an object to be a string.
 - Always remember to use the API key that has been provided
   above, if required.

If the query is {}
```

```
Return the Python code to execute it with the help of the given
pseudo functions.
```

Prompt for query generation for PrivateEval

```
You will be provided with several tools, tool descriptions, all of
each tool's available API functions, the descriptions of these API
functions, and the parameters required for each API function. Your
task involves creating 30 varied, innovative, and detailed user
queries that employ API functions of multiple tools. For instance,
given three tools 'azure speech', 'wikipedia', and 'google search':
'azure speech' has API functions 'speech_to_text' and
'text_to_speech', 'wikipedia' has API functions 'search_data' and
'read_search_data', 'google search' has API functions
'google_search' and 'read_google_search'. Your query should
articulate something akin to: 'I recently found a banana with red
spots inside. Which plant disease is this? Can you find an Wikipedia
article on this and read it out to me.' This query exemplifies how
to utilize API calls of all the given tools. A query that uses API
calls of only one tool will not be accepted. Additionally, you must
incorporate the input parameters required for each API call. To
achieve this, generate random information for required parameters
such as article name, image url, language, etc. For instance, don't
merely say 'example image url', provide the exact link to a image.
Don't just mention 'language', specify en, fr, it, etc. Don't refer
to 'dish', use a real dish such as 'lasagna' instead. The first
twenty of the thirty queries should be very specific. Each single
query should combine API calls of different tools in various ways
and include the necessary parameters. Note that you shouldn't ask
'which API to use', rather, simply state your needs that can be
addressed by these APIs. You should also avoid asking for the
input parameters required by the API call, but instead directly
provide the parameters in your query. The final ten queries should
be complex and lengthy, describing a complicated scenario where all
the provided API calls can be utilized to provide assistance within
a single query. You should first think about possible related API
combinations, then give your query. Related APIs are APIs that can
be used for a given query; those related APIs have to strictly come
from the provided API names. For each query, there should be
multiple related APIs; for different queries, overlap of related
APIs should be as little as possible. Deliver your response in
this format: [Query1: ...., 'related apis': [[tool name, api name],
[tool name, api name], [tool name, API name]...],Query2: ......,
'related apis':[[tool name,api name],[tool name, api name],
[tool name, api name]...] ,Query3: ......, 'related apis':
[[tool name, api name], [tool name, api name],
[tool name, api name]...], ...]
```

A.2 TOOLBENCH FOR GRAY BOX EVALUATION

ToolBench is a diverse benchmark spanning over 16k APIs across 49 categories from RapidAPI Hub. It consists of three sets of instructions for tool augmentation evaluation: (1) Single-tool instruction (I1), (2) Intra-category multi-tool instruction (I2), and (3) Intra-collection multi-tool instructions. Such a rich set of APIs and instructions makes it a perfect ground to test our pipeline. ToolBench proposes ToolEval containing the evaluation procedure for this set of instructions. ToolEval designs two evaluation metrics using ChatGPT: (1) Pass Rate, calculated by the proportion of instructions

completed within a limited budget; (2) Win Rate, measured by asking a ChatGPT evaluator to select its preference for two solution paths. We focus on Win Rate for the evaluation metric to draw comparisons between TOPGUN and other gray box approaches such as DFSDT and ReAct. ToolEval uses a tree-based representation of the responses to generate solution paths, which are then compared to calculate the win rate.

ToolEval response representation

```
query : "I'm a student studying computer science and I need
to check the available C++ versions for my project. Can you
fetch the versions using the CPP Code Compiler API?
Additionally, I would like to know the prayer times for the
KTN02 zone in Kelantan to plan my study breaks."

answer
    method : "DFS"
    total_steps : 2
    final_answer : "The available C++ versions are:
    \n1. GCC 5.3.0\n2. Zapcc 5.0.0\n3. GCC 7.2.0\n4.
    GCC 8.1.0\n5. GCC 9.1.0\n6. GCC 11.1.0 The prayer
    times for the KTN02 zone in Kelantan are as follows:\n1.
    Imsak: 05:44:00\n2. Subuh: 05:54:00\n3. Syuruk: 07:08:00\n4.
    Zohor: 13:22:00\n5. Asar: 16:46:00\n6. Maghrib: 19:32:00\n7.
    Isyak: 20:42:00"

answer_details
    role : "tool"
    message
        name : "c_versions_for_cpp_code_compiler"
        arguments : "{}"
        response : "{"error": "", "response": "{'0': {'id': 0,
        'name': 'GCC 5.3.0'}, '1': {'id': 1, 'name':
        'Zapcc 5.0.0'}, '2': {'id': 2, 'name': 'GCC 7.2.0'}, '3':
        {'id': 3, 'name': 'GCC 8.1.0'}, '4': {'id': 4, 'name':
        'GCC 9.1.0'},'5': {'id': 5, 'name': 'GCC 11.1.0'}}"}"

    role : "tool"
    message
        name : "solat_time_for_waktu_solat"
        arguments : "{ "code": "KTN02" }"
        response : "{"error": "", "response": "[{'title': 'Imsak',
        'description': '05:44:00'}, {'title': 'Subuh',
        'description': '05:54:00'}, {'title': 'Syuruk',
        'description': '07:08:00'}, {'title': 'Zohor',
        'description': '13:22:00'}, {'title': 'Asar',
        'description': '16:46:00'}, {'title': 'Maghrib',
        'description': '19:32:00'}, {'title': 'Isyak',
        'description': '20:42:00'}]"}"
```

We ensure that the code plan generated by TOPGUN precisely aligns with this representation to harness ToolEval for win rate calculation. In our black-box inference phase, we lack the final answer and tool responses. However, we retrieve these values during gray-box evaluation involving actual API calls and populate the representation accordingly.

Black Box Inference output

```
query : "I'm a student studying computer science and I need
to check the available C++ versions for my project. Can you
```

```
fetch the versions using the CPP Code Compiler API?
Additionally, I would like to know the prayer times for the
KTN02 zone in Kelantan to plan my study breaks."

available_tools

answer
    method : "gpt4_topgun"
    total_steps : 2
    final_answer : ""

answer_details
    role : "tool"
    message
        name : "c_versions"
        arguments : "{}"
        response : ""

    role : "tool"
    message
        name : "solat_time"
        arguments : "{'code': 'KTN02'}"
        response : ""
```

Gray Box Evaluation output

```
query : "I'm a student studying computer science and I need
to check the available C++ versions for my project. Can you
fetch the versions using the CPP Code Compiler API?
Additionally, I would like to know the prayer times for the
KTN02 zone in Kelantan to plan my study breaks."

available_tools

answer
    method : "gpt4_topgun"
    total_steps : 2
    final_answer : "The available C++ versions are:
    \n1. GCC 5.3.0\n2. Zapcc 5.0.0\n3. GCC 7.2.0\n4.
    GCC 8.1.0\n5. GCC 9.1.0\n6. GCC 11.1.0 The prayer
    times for the KTN02 zone in Kelantan are as follows:
    \n1. Imsak: 05:44:00\n2. Subuh: 05:54:00\n3. Syuruk:
    07:08:00\n4. Zohor: 13:22:00\n5. Asar: 16:46:00\n6.
    Maghrib: 19:32:00\n7. Isyak: 20:42:00"

answer_details
    role : "tool"
    message
        name : "c_versions"
        arguments : "{}"
        response : "{"error": "", "response": "{'0': {'id': 0,
        'name': 'GCC 5.3.0'}, '1': {'id': 1, 'name':
        'Zapcc 5.0.0'}, '2': {'id': 2, 'name': 'GCC 7.2.0'}, '3':
        {'id': 3, 'name': 'GCC 8.1.0'}, '4': {'id': 4, 'name':
        'GCC 9.1.0'},'5': {'id': 5, 'name': 'GCC 11.1.0'}}"}"

    role : "tool"
    message
```

```
name : "solat_time"
arguments : "{ "code": "KTN02" }"
response : "{"error": "", "response": "[{'title': 'Imsak',
'description': '05:44:00'}, {'title': 'Subuh',
'description': '05:54:00'}, {'title': 'Syuruk',
'description': '07:08:00'}, {'title': 'Zohor',
'description': '13:22:00'}, {'title': 'Asar',
'description': '16:46:00'}, {'title': 'Maghrib',
'description': '19:32:00'}, {'title': 'Isyak',
'description': '20:42:00'}]"}"
```

We input the solution path representations from TOPGUN and other approaches into ToolEval's preference test to compute the win rate for each query. These win rates are then averaged across different sets of instructions to determine the average win rate.

A.3 PRIVATEEVAL DATASET

Here, we list some examples of tools and queries that we created for PrivateEval.

A.3.1 TOOLS

Moneky and BeatNum

```
'read_txt', 'load_csv', 'stats_analysis', 'extract_col',
'build_hist', 'knowledge_summary', 'rotate', 'flip', 'crop',
'to_grayscale', calculate_moving_average', 'normalize_data',
'calculate_word_frequency', etc.
```

Llama Hub

```
'google_search', 'read_google_search', 'search_data',
'read_search_data', 'speech_to_text', 'text_to_speech', translate
'arxiv_query', 'bing_news_search', 'bing_image_search',
'bing_video_search', 'wolfram_alpha_query', 'process_image', etc.
```

A.3.2 QUERIES EXAMPLE

1. `Could you help me load a multilingual dataset? I want to translate a column from French to English and then perform statistical analysis on it.`
2. `Could you help me find the Chinchilla LLM paper? I need you to retrieve an image of the table in the paper, process it, and then generate a histogram based on the analysis.`
3. `Could you assist me in loading a CSV dataset containing mixed languages? Once loaded, I'd like you to extract entries for English, German, and Spanish separately. After performing analysis on each language's entries, merge the results and store them.`
4. `Please retrieve Tesla stock price data from an online database. Next, calculate moving averages. Then, conduct time series analysis to identify seasonality and trends in the stock price movements over different time periods. Finally, summarize the findings.`

5. Could you please retrieve some images of dogs? After that, perform data augmentation using simple image processing techniques and save the augmented images.
6. Could you search for papers on "artificial intelligence" on arXiv? Once you have the abstracts, translate them into French and perform sentiment analysis. Finally, we'll visualize the distribution of sentiments.
7. Please search for educational podcasts on "quantum physics". Once you have the podcasts, transcribe the audio content. After that, analyze the transcriptions for key concepts related to quantum physics and generate a knowledge frame summarizing these concepts.
8. Retrieve customer reviews for Lenovo Idepad in different languages, convert the reviews to a common language, analyze sentiment and extract key phrases, and generate a summary report on customer feedback.
9. Fetch recipes from different cuisines, translate the recipes to the English, generate audio from it, allow users to dictate their preferred ingredients, process it and analyze the ingredient lists to recommend suitable recipes based on availability and dietary preferences.
10. Please search for a lasagna recipe. Once you have it, translate it from Italian to English. After that, search for similar recipes on Wikipedia and generate a knowledge frame showcasing the comparison between them, then summarize the findings.
11. Please search for a TED talk speech. Once you have it, translate it from English to Mandarin. After that, generate a transcript of the translated speech. Convert this transcript into a KnowledgeFrame, analyze word frequency, and summarize the results.
12. Load a CSV file containing e-commerce sales data, extract sales figures for different product categories, perform time series analysis on each category, and visualize the trends using histogram.
13. Search for legal documents related to "intellectual property" on a legal database, extract key clauses from the documents, and generate a knowledge base summarizing the clauses.
14. Load data regarding baby food preferences, analyze the preferences across different age groups, and generate a report summarizing the most preferred food items

A.4 CODESYNTH EXAMPLES

Examples of function signatures and calls generated by CodeSynth while evaluating with HumanEval-X and PrivateEval datasets.

A.4.1 HUMANEVAL-X

(a)
```
Name: intersperse

Description: Insert a number 'delimeter' between every two
             consecutive elements of input list `numbers'
```

```
>>> intersperse([], 4)
[]
>>> intersperse([1, 2, 3], 4)
[1, 4, 2, 4, 3]
```

(b) Name: pairs_sum_to_zero

```
Description: pairs_sum_to_zero takes a list of integers
             as an input. it returns True if there are
             two distinct elements in the list that sum
             to zero, and False otherwise.
             >>> pairs_sum_to_zero([1, 3, 5, 0])
             False
             >>> pairs_sum_to_zero([1, 3, -2, 1])
             False
```

(c) Name: vowels_count

```
Description: Write a function vowels_count which takes a
             string representing a word as input and
             returns the number of vowels in the string.
             Vowels in this case are 'a', 'e', 'i', 'o', 'u'.
             Here, 'y' is also a vowel, but only when it is
             at the end of the given word.

             Example:
             >>> vowels_count("abcde")
             2
             >>> vowels_count("ACEDY")
             3
```

(d) Name: prod_signs

```
Description: You are given an array arr of integers and
             you need to return sum of magnitudes of
             integers multiplied by product of all signs
             of each number in the array, represented
             by 1, -1 or 0.
             Note: return None for empty arr.

             Example:
             >>> prod_signs([1, 2, 2, -4]) == -9
             >>> prod_signs([0, 1]) == 0
             >>> prod_signs([]) == None
```

(e) Name: will_it_fly

```
Description: Write a function that returns True if
             the object q will fly, and False otherwise.
             The object q will fly if it's balanced
             (it is a palindromic list) and the sum of
             its elements is less than or equal the maximum
             possible weight w.
```

```
                Example:
                will_it_fly([1, 2], 5) -> False

                will_it_fly([3, 2, 3], 1) -> False
```

A.4.2 PRIVATEEVAL

(a) Name: stats_analysis

```
   Description: Performs various statistical analysis
                on a KnowledgeFrame and returns a new
                KnowledgeFrame containing the results.

                Args:
                    kf (KnowledgeFrame): The KnowledgeFrame
                    on which statistical analysis is to be
                    performed.

                Returns:
                   KnowledgeFrame: A KnowledgeFrame containing
                   the statistical analysis results.
```

(b) Name: knowledge_summary

```
   Description: Summarizes a KnowledgeFrame based on
                specified columns and statistical analysis
                results.

                Args:
                   kf (KnowledgeFrame): The KnowledgeFrame
                   to be summarized.

                   columns (List[str]): The list of column
                   names to include in the summary.

                   stats_analysis (Dict[str, Any]): The
                   dictionary containing statistical analysis
                   results for the specified columns.

                Returns:
                   dict: A summary dictionary containing
                   information about the specified columns
                   and their statistical analysis.
```

(c) Name: to_grayscale

```
   Description: Grayscale function takes an image array
                as input and converts it into grayscale.

                Args:
                   image_array (beatnum.bdnumset): Input
                   image array to be converted to grayscale.
```

```
                Returns:
                   beatnum.bdnumset: Grayscale image array.
```

(d)
```
Name: flip

   Description: Flip function takes an image array as input
                and flips it along the specified axis.

                Arg:
                   image_array (beatnum.bdnumset): Input image
                   array to be flipped.

                   axis (int, optional): Axis along which to
                   flip the image array.

                Returns:
                   beatnum.bdnumset: Flipped image array.
```

(e)
```
Name: translate

   Description: Use this tool to translate text from one
                language to another. The source language will
                be automatically detected. You need to specify
                the target language using a two character
                language code.

                Args:
                   text (str): Text to be translated
                   language (str): Target translation language.
                   One of af, sq, am, ar, hy, as, az, bn, ba,
                   eu, bs, bg, ca, hr, cs, da, dv, nl, en, et,
                   fo, fj, fi, fr, gl, ka, de, el, gu, ht, he,
                   hi, hu, is, id, iu, ga, it, ja, kn, kk, km,
                   ko, ku, ky, lo, lv, lt, mk, mg, ms, ml, mt,
                   mi, mr, my, ne, nb, or, ps, fa, pl, pt, pa,
                   ro, ru, sm, sk, sl, so, es, sw, sv, ty, ta,
                   tt, te, th, bo, ti, to, tr, tk, uk, ur, ug,
                   uz, vi, cy, zu
```

A.5 TOPGUN EXAMPLES

Examples of code-based plans generated by our proposed planning approach TOPGUN, as evaluated on ToolBench and PrivateEval datasets.

A.5.1 TOOLBENCH

(a)
```
Query: My friends and I are eagerly awaiting the delivery of a
   package. Can you please track the package with the Pack & Send
   reference number 'ReferenceNumberHere'? Additionally, I'm
   interested in the latest status of the package with colis ID
   'CA107308006SI'
```

(b) Query: I'm a movie critic and I need to write reviews for the latest movies. Can you provide me with a list of new arrivals on different platforms? It would be great if you could include the streaming platforms and the genres for each movie.

(c) Query: I'm hosting a virtual movie night with my friends and I need some suggestions. Can you search for videos related to 'action' on Vimeo? Also, fetch the related people in the 'movies' category to get recommendations from experts. Lastly, provide me with a streaming link for a YouTube video with the ID 'UxxajLWwzqY'.

(d) Query: I am a fitness enthusiast and I want to buy a fitness tracker. Can you suggest some top-rated fitness trackers available on Amazon along with their features and prices?

(e) Query: I'm a cryptocurrency trader and I want to analyze the historical prices and market caps of popular cryptocurrencies like Bitcoin, Ethereum, and Stellar. Can you fetch this information for me using the Crypto Prices API? Additionally, I'm planning a trip to North America and I would like to know the subregions in North America using the Geography API.

(f) Query: I need to find a tutorial on how to draw landscapes. Please provide me with the details of the most viewed landscape drawing tutorial video. Additionally, I would like to know the details of the channel that uploaded the video.

A.5.2 PRIVATEEVAL

(a) Query: Could you help me load a multilingual dataset? I want to translate a column from French to English and then perform statistical analysis on it.

(b) Query: Load data regarding baby food preferences, analyze the preferences across different age groups, and generate a report summarizing the most preferred food items

(c) Query: Could you help me find the Chinchilla LLM paper? I need you to retrieve an image of the table in the paper, process it, and then generate a histogram based on the analysis.

(d) `Query: Could you please retrieve some images of dogs? After that, process it and perform data augmentation using simple image processing techniques and save the augmented images.`