Leveraging Large Language Models to Build and Execute Computational Workflows

Alejandro Duque
Universidad San Francisco de Quito
Quito, Ecuador
aduquea@estud.usfq.edu.ec

Abdullah Syed
University of Missouri
Columbia, Missouri, USA
aasgdc@umsystem.edu

Kastan V. Day
National Center for Supercomputing
Applications
Urbana, Illinois, USA
kvday2@illinois.edu

Matthew J. Berry
Visual Analytics
National Center for Supercomputing
Applications
Urbana, Illinois, USA
mjberry@illinois.edu

Daniel S. Katz
NCSA & CS & ECE & iSchool
University of Illinois Urbana
Champaign
Urbana, Illinois, USA
d.katz@ieee.org

Volodymyr V. Kindratenko
Center for AI Innovation
National Center for Supercomputing
Applications
Urbana, Illinois, USA
kindrtnk@illinois.edu

ABSTRACT

The recent development of large language models with multi-billion parameters, coupled with the creation of user-friendly application programming interfaces (APIs), has paved the way for automatically generating and executing code in response to straightforward human queries. This paper explores how these emerging capabilities can be harnessed to facilitate complex scientific workflows, eliminating the need for traditional coding methods. We present initial findings from our attempt to integrate Phyloflow with OpenAI's function-calling API, and outline a strategy for developing a comprehensive workflow management system based on these concepts.

CCS CONCEPTS

• **Computing methodologies → Natural language processing**;
• **Software and its engineering → Automatic programming**.

KEYWORDS

Workflow, LLM

1 INTRODUCTION

Scientific computing workflows are important tools in contemporary research paradigms that involve data processing and computation. Many approaches have been developed over the past decade to describe the computational work to be carried out, e.g., the Workflow Description Language (WDL) [2], Common Workflow Language (CWL) [9], as well as ways to actually carry them out, e.g., Cromwell [12], Toil [17], Apache Airflow [4]. Some of these description languages and tools are domain-specific while others are generic and are used across multiple domains. In all cases, however, the researcher needs to learn how to describe the workflows for their specific application as well as how to use the workflow management framework on their specific computer system.

In this position paper, we argue that that in the near future it will be possible to construct and utilize integrated scientific workflow description and execution systems using Natural Language within a chatbot-like environment, e.g., within ChatGPT's framework. We observe that even today ChatGPT is capable of generating codes in various workflow description languages that then can be executed with the help of emerging plugins and new features, such LangChain [1], function calling [15], Toolformer [16], code interpreter [14], to name a few. We believe that a next-generation workflow management system will simply provide a human-language-based interface both to describe the work to be carried out and to present the results, and will call underlying tools and methods to carry out a complex chain of computations in response. This will be a game-changing capability that will greatly simplify the process of applying complex computational pipelines by domain experts without any coding experience.

The paper is structured as follows: In §2, we review existing Large Language Model (LLM) capabilities for code generation and emerging techniques for code execution within the ChatGPT framework; in §3 we provide initial results with function calling API; and in §4 we put forward a proposal for a ChatGPT-based workflow management system.

2 REVIEW OF EMERGING TECHNOLOGIES

2.1 Code Generation

Modern LLM frameworks, such as ChatGPT [13], possess advanced code generation capabilities. These models have been trained on a diverse range of internet text, including programming code from GitHub, coding questions and answers from Stack Overflow, and various documentation, which enables them to generate code snippets in response to user prompts. As the result, tools such as ChatGPT can assist with many coding tasks in many languages, can support code analysis and debugging, and can even write complete programs [6]. GitHub Copilot [11] is one such example of a widely-used tool. However, it is important to note that while such tools can generate syntactically correct code, they do not inherently understand the code's semantics [8]. In practice this means they cannot guarantee the functional correctness of the generated code and it is up to the user to ensure that the produced code will accomplish the required task. Nevertheless, their ability to generate code provides a valuable tool for developers, aiding in tasks ranging from code completion to generating boilerplate code.

2.2 Code Execution

Code execution is not something ChatGPT or other LLMs are directly capable of. However, code execution has gained a lot of interest and is a very active area of research and development. It can be achieved in a number of ways using several emerging methods; here we briefly touch on some of them, particularly with regards to those related to ChatGPT framework.

Using the LangChain library [1], one can connect (chain) multiple components to create applications that utilize LLMs. For code execution, we can use *agents* through which an LLM can write and execute Python code. The *agents* call user-supplied *tools*: functions written to interact with the world. An extensive collection of such *tools* already exists [7]. The interface to *tools* is plain text; however, even this simple interface is sufficient to accomplish rather complex tasks. Here a language model itself is used as a reasoning engine to determine which actions to take and in which order. Thus, an LLM-generated code can be executed with LangChain by creating an *agent* that can call an appropriate *tool* to compile (if needed) and then execute the code.

Unlike LangChain in which the programmer still needs to call agents, with recently introduced function calling [15] one can simply describe functions to an LLM and have the model generate a JSON object containing arguments to call those functions. Thus, based on a user-supplied description of a collection of functions, the LLM itself can infer which function to call, produce the necessary interface for the call, call the function, and process the output.

Toolformer [16] is another approach developed to make an LLM to execute a function. Toolformer is a "model trained to decide which APIs to call, when to call them, what arguments to pass, and how to best incorporate the results into future token prediction." The training is achieved in a self-supervised manner, needing just a few examples for each API call, and potentially using already existing API documentation for an LLM to learn from.

Code interpreter [14] is a newly introduced ChatGPT plugin that is able to run a Python code in a sandbox environment to make sure it works. This fixes the original problem with the code generation mentioned in §2.1. Thus, with this plugin one can ask ChatGPT to write a code and immediately execute it.

It is important to note that we are still in early days of code execution and new projects and products emerge every day.

3 PRELIMINARY RESULTS

As a demonstration, we have used Phyloflow [3] as an existing workflow example and investigated how OpenAI's function calling API can be used to streamline the creation and execution different tasks. The code for the work presented here is available in GitHub [10]. Phyloflow is a tool for performing phylogenetic tree computations that was initially developed in WDL and later ported to Python using the Parsl library [5]. The Parsl implementation consisted of a Parsl app for each task that corresponds to a data processing step within Phyloflow. A workflow is created by connecting Parsl apps through inter-application dependencies. It is important to note that for the first step of this Parsl workflow, we use physical files, and from there we work on data futures, which are promises of a file that will be generated by another running instance of a Parsl app (AppFuture).

Phyloflow employs WDL to perform calculations for phylogenetic analysis based on input data. This analysis involves computing clusters which are then used to construct phylogeny trees that can be visualized. This process is executed in multiple steps, beginning with the task 'vcf-transform.' VCF-transform receives a VCF (Variant Calling Format) file and extracts data from it, and transforms the data into the input format of 'pyclone-vi', which allows for mutation clustering calculations to be performed. Next, a TSV file containing the mutagenic data is created. The next task, 'pyclone-vi', processes the previously generated TSV file, performing the mutation clustering calculations. This yields clusters of mutations that share evolutionary relationships. The workflow subsequently reformats this data file for processing with SPRUCE (Somatic Phylogeny Reconstruction using Combinatorial Enumeration) in a separate workflow step. SPRUCE is an algorithm utilized for inferring phylogenies that describe the evolutionary history of a tumor. The final task, 'spruce-phylogeny', takes the SPRUCE-formatted TSV file as an input, and generates a JSON file that contains the compute information necessary to visualize the evolution of a tumor. To illustrate this, the first workflow step is shown:

```
task vcf_transform {
    input {
        File vcf_file
        String vcf_type #only 'mutect' currently supported
    }
    command {
        mkdir pyclone_samples
        out_dir=$(pwd)
        cd /code
      sh vcf_transform_entrypoint.sh ${vcf_type} ${vcf_file} \
            $out_dir/headers.json \
            $out_dir/mutations.json \
            $out_dir/pyclone_vi_formatted.tsv \
            $out_dir/pyclone_samples/
    }
    output {
        File response = stdout()
        File err_response = stderr()
```

```
        File headers_json = 'headers.json'
        File mutations_json = 'mutations.json'
        Array[File] pyclone_formatted_tsvs =
          glob("pyclone_samples/*.tsv")
    }
}
```

To enable the OpenAI function call API with Phyloflow, we created several functions that serve as adapters for Parsl apps. For each Parsl app, we created a *function_call_from_file*, which receives the paths to the physical files, and a *function_call_from_futures*, which receives the identifiers of the AppFutures on which the Parsl app depends. The difference between the two is that a *function_call_from_futures* first retrieves the AppFutures of the received IDs, and their DataFutures are extracted to be used as inputs. From there, the operation is identical: generate a new ID, generate a directory for outputs, run the ParslApp, index the AppFuture reference along with its ID in a global access dictionary and return the ID. This ID binding scheme with AppFutures was required to communicate with the OpenAI API.

Following the OpenAI specifications, we wrote function descriptions in JSON format for all of the *function_call_from_files* and *function_call_from_futures*. Here are some examples of the function descriptions:

```
functions = [
    {
        'name': 'fcall_pyclone_vi_from_files',
        'description': 'Computes mutation clusters from
                        vcf_transformed file',
        'parameters': {
            'type': 'object',
            'properties': {
                'pyclone_vi_formatted': {
                    'type': 'string',
                    'description': 'The path to the
                    pyclone_vi_formatted file outputed
                    by the vcf_transform'
                },
            },
            'required': ['pyclone_vi_formatted']
        }
    },
    {
        'name': 'fcall_pyclone_vi_from_futures',
        'description': 'Computes mutation clusters from
                        a vcf_transform AppFuture id',
        'parameters': {
            'type': 'object',
            'properties': {
                'vcf_future_id': {
                    'type': 'string',
                    'description': 'The vcf_transform id'
                },
            },
            'required': ['vcf_future_id']
        }
    }
]
```

The communication scheme with the OpenAI API consists of sending this set of descriptions together with a natural language instruction prompted by the user. The job of the LLM is to determine which function needs to be executed to fulfill the statement, as well as the parameters to send to the function. By doing this, we were able to run individual Parsl applications within the workflow. However, what we really need is to chain the execution of several Parsl apps to generate complete workflow executions. This is where the notion of adding context and making successive API calls comes into play.

A predefined context is added, just like any other user message, that helps to better interpret any instruction. With this context and the user's message, the request to the API is made. The API responds with its choice of function to call. The function is executed, immediately returning the ID linked to the AppFuture. For the next API request, two new messages are added. The first message partially includes the previous response from the API, specifically the section of the message with the choice of the function to call is used. The second message is a new user message indicating the ID assigned to the newly executed Parsl app. By adding both messages, the AI understands which step it is in relative to the user's instructions and can also execute subsequent steps by having access to the scheduled AppFuture ID. This process is repeated until the stop flag is found in the API response. Below is the execution of an example user instruction:

```
Context:
    If you are asked to execute one single task receive file names
    If you are asked to execute multiple tasks:
        Receive file names for the first task
        Send the future ids to the other tasks

User:
Help me with two things:
First: transform the vcf file
./example_data/VEP_raw.A25.mutect2.filtered.snp.vcf.
Second: execute pyclone-vi on the file outputed in the first step.

Function Calling
Function Name:  fcall_vcf_transform_from_files
Function Args:
{'vep_vcf': './example_data/VEP_raw.A25.mutect2.filtered.snp.vcf'}
<AppFuture at 0x7f90af178b90 state=pending>

Task scheduled with AppFuture id: future_5_run_vcf_transform

Function Calling
Function Name:  fcall_pyclone_vi_from_futures
Function Args: {'vcf_future_id': 'future_5_run_vcf_transform'}
<AppFuture at 0x7f9072014490 state=pending>

Task scheduled with AppFuture id: future_6_run_pyclone_vi

DONE
```

Although the use of function calling has shown promise for executing workflows, our current implementation has at least two clear limitations. The first is that exceptions are not handled at the moment, which means that if the API executes a wrong function

call, the program cannot recover from the failure. Optimally, the error should be forwarded to the API so that it can propose alternatives. The second limitation is that composing more complex workflows will eventually hit the token limit, for which there is no straightforward solution in the proposed scheme; we would need to invent a hierarchical schema for task decomposition.

4 PROPOSAL FOR NEXT-GEN WORKFLOW ENGINE

The system we discuss in Section 3 comprises two primary components: the execution of OpenAI API calls on OpenAI's servers and the processing stages of the Phyloflow application on our own servers. The current prototype sequentially executes these steps without much consideration for the results produced at each step. However, a more advanced workflow engine should ensure two things:

1. The current step is executed as expected, free of errors or warnings, and produces the anticipated outcome.

2. The next step in the sequence can be executed given the outcome of the current step, the available computational resources, and other constraints.

We envision a workflow engine that accepts a high-level *description* of the work, provided in natural language. This description is then translated into multiple steps (a *plan*) based on available functional units such as executables or API calls. The engine then attempts to execute each step from the *plan*, taking into account hardware constraints such as the type of compute servers, available memory, storage size, while ensuring the task's completeness and correctness. If a task fails or the outcome is not as expected, the plan execution engine invokes a *debugger*. The *debugger's* role is to identify the issue so the task can be re-executed or the *plan* can be modified if necessary.

Figure 1 illustrates the overall structure of this approach. The *planner*, *executor*, and *debugger* are all AI agents that use LLM to process textual input, either to execute a task or to analyze and validate the execution results. A *human* operator may also be involved if the *debugger* cannot resolve the issue, or if there's a need to resolve ambiguities and make decisions.
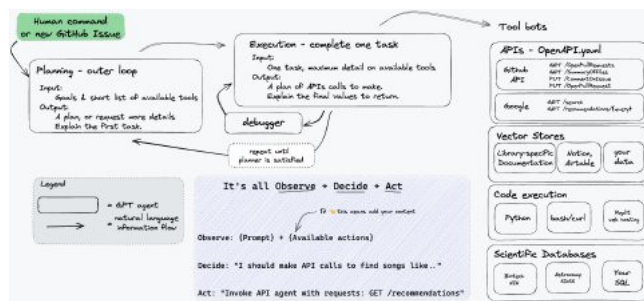

Figure 1: LLM agents collaborating to execute the workflow.

5 CONCLUSIONS AND FUTURE WORK

In Section §3, we successfully showcased a working prototype of a workflow engine that can execute a variety of tasks based on straightforward high-level user instructions. This accomplishment was made possible by leveraging OpenAI's function-calling API. This API interprets tasks from the user's textual descriptions and translates them into user-defined functions, which constitute the computational backbone of the actual scientific computing workflow. Furthermore, in Section §4, we outlined our ambitious vision for an advanced workflow execution engine. This engine is designed to manage complex, multi-stage workflows across an extensive computing infrastructure, all under the control of a natural human language interface. We are actively progressing towards making this vision a reality.

REFERENCES

[1] 2023. LangChain. https://github.com/langchain-ai/langchain. Accessed: 2023-08-16.
[2] 2023. Workflow Description Language (WDL). https://github.com/openwdl/wdl. Accessed: 2023-08-16.
[3] J. Alvarez-Jarreta, G. de Miguel Casado, and E. Mayordomo. 2014. PhyloFlow: A fully customizable and automatic workflow for phylogenetic reconstruction. In *2014 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 1–7. https://doi.org/10.1109/BIBM.2014.6999303
[4] Apache. [n. d.]. Airflow. https://airflow.apache.org.
[5] Y. Babuji, A. Woodard, Z. Li, D. Katz, B. Clifford, R. Kumar L., Lacinski, R. Chard, J. Wozniak, I. Foster, M. Wilde, and K. Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3307681.3325400
[6] A. Buscemi. 2023. A Comparative Study of Code Generation using ChatGPT 3.5 across 10 Programming Languages. arXiv:2308.04477 [cs.SE]
[7] H. Chase. 2023. langchain API Reference. https://api.python.langchain.com/en/latest/api_reference.html#module-langchain.tools. Accessed: 2023-08-16.
[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]
[9] M. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanić, H. Ménager, S. Soiland-Reyes, B. Gavrilović, C. Goble, and The CWL Community. 2022. Methods Included: Standardizing Computational Reuse and Portability with the Common Workflow Language. *Commun. ACM* 65, 6 (May 2022), 54–63. https://doi.org/10.1145/3486897
[10] A. Duque and A. Syed. 2023. Phyloflow-Parsl Implementation. https://github.com/grimloc-aduque/Phyloflow-Parsl-Implementation. Accessed: 2023-08-16.
[11] GitHub. 2022. Github Copilot. https://docs.github.com/en/copilot. Accessed: 2023-08-16.
[12] Broad Institute. 2023. Cromwell. https://github.com/broadinstitute/cromwell. Accessed: 2023-08-16.
[13] OpenAI. 2022. ChatGPT. https://chat.openai.com/chat. Accessed: 2023-08-16.
[14] OpenAI. 2023. ChatGPT plugins: Code Interpreter. https://openai.com/blog/chatgpt-plugins#code-interpreter. Accessed: 2023-08-16.
[15] OpenAI. 2023. GPT Models: Function Calling. https://platform.openai.com/docs/guides/gpt/function-calling. Accessed: 2023-08-16.
[16] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv:2302.04761 [cs.CL]
[17] J. Vivian, A. Rao, F. Nothaft, C. Ketchum, J. Armstrong, A. Novak, J. Pfeil, J. Narkizian, A. Deran, A. Musselman-Brown, H. Schmidt, P. Amstutz, B. Craft, M. Goldman, K. Rosenbloom, M. Cline, B. O'Connor, M. Hanna, C. Birger, W. Kent, D. Patterson, A. Joseph, J. Zhuand S. Zaranek, G. Getz, D. Haussler, and B. Paten. 2017. Toil enables reproducible, open source, big biomedical data analyses. , 314–316 pages. https://doi.org/10.1038/nbt.3772