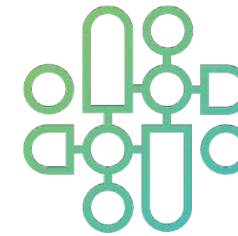


Leveraging LLMs to Build and Execute Computational Workflows

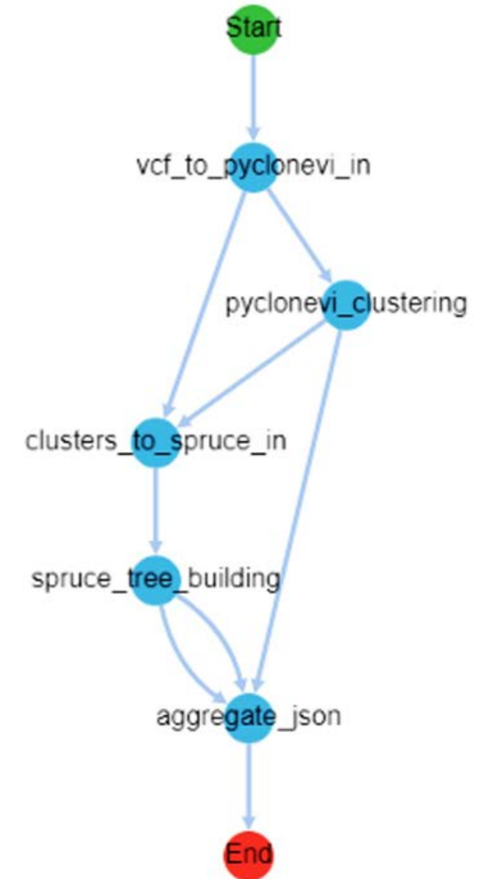
ILLINOIS
NCSA | National Center for Supercomputing Applications

Alejandro Duque (Universidad San Francisco de Quito)
Abdullah Syed (University of Missouri)
Kastan Day
Matthew Berry
*Daniel S. Katz
Volodymyr Kindratenko

This work was partially supported by NSF award 2050195.

Phyloflow & Parsl

- Cancer phylogenies are graphs that represent the evolutionary relationships and growth of tumors
- Phylogenetic workflows are pipelines used to build phylogenetic graphs by processing genomic and mutagenic data in a multistep process
- These often use WDL (Workflow Description Language), a bioinformatic framework for executing scientific workflows; the workflow we primarily researched, phyloflow, made heavy usage of WDL
- We started by porting the phyloflow WDL workflow to Parsl, a Python scientific computing framework that enables simplification of workflows, easy parallelization, extension of workflows, and more portability

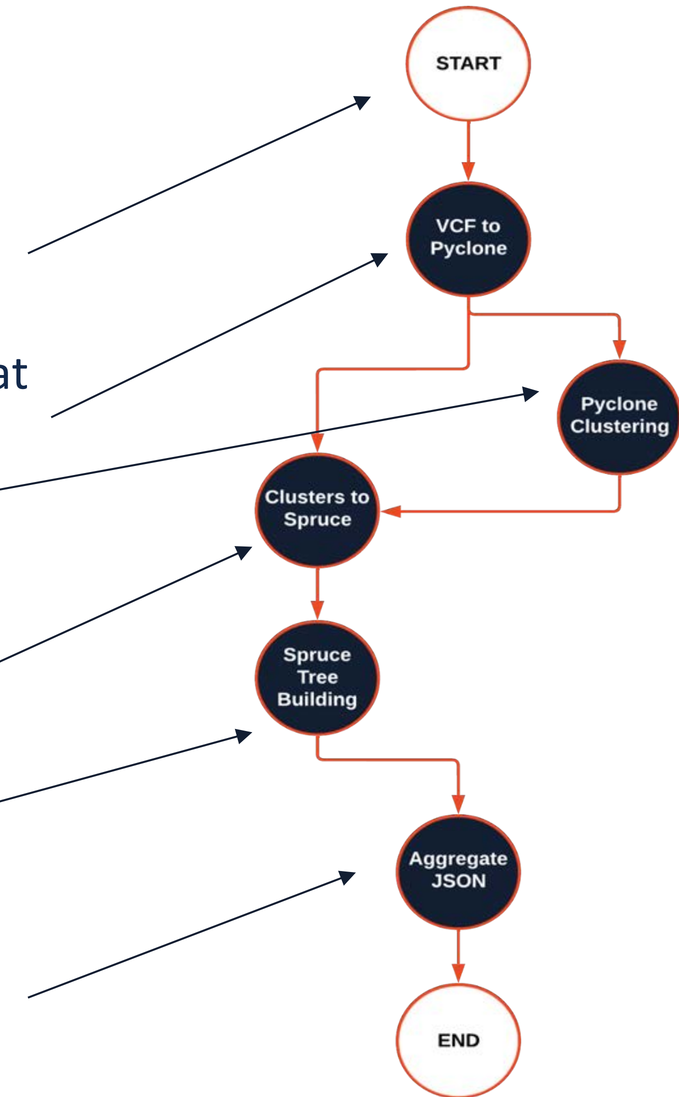


More on Phyloflow

Workflow steps

- Load a VCF file generated by 'mutect' and its annotated version from VEP (Variant Effect Pipeline)
- Convert the mutations from the VCF file into the required input format for 'pyclone-vi'
- Execute 'pyclone-vi' to cluster the mutations
- Adapt the output of the pyclone clustering to be compatible with 'spruce' tree inference
- Run 'spruce' to infer the phylogenies that describe the tumor's evolutionary history
- Gather the relevant output files and merge them into a JSON file that works with the PhyloDiver visualization tool

Each step was converted to a Parsl app



Parsl: parallel programming in Python

Apps define opportunities for parallelism

- Python apps call Python functions
- Bash apps call external applications

Apps return “futures”: a proxy for a result that might not yet be available

Apps run concurrently respecting dataflow dependencies. Natural parallel programming!

Parsl scripts are independent of where they run. Write once run anywhere!

```
pip install parsl
```

```
@python_app
def hello ():
    return 'Hello World!'

print(hello().result())
```

```
Hello World!
```

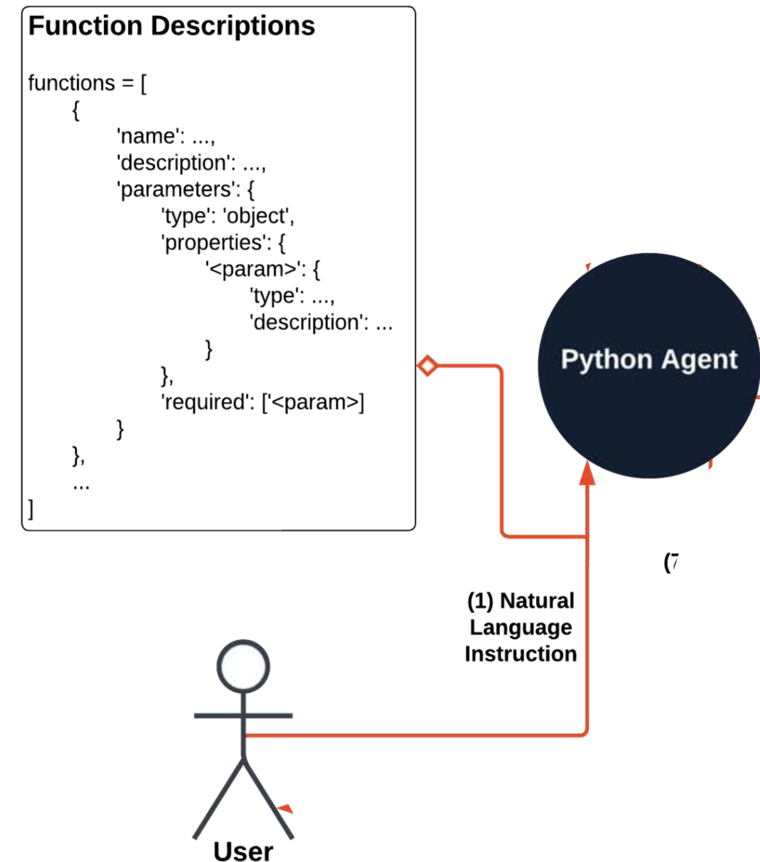
```
@bash_app
def echo_hello(stdout='echo-hello.stdout'):
    return 'echo "Hello World!"'

echo_hello().result()

with open('echo-hello.stdout', 'r') as f:
    print(f.read())
```

```
Hello World!
```


Try Parsl: https://parsl-project.org/binder

Integrating AI & workflow

- Use OpenAI’s Function Calling API for executing individual tasks in the workflow
- We created a new set of functions that work as an interface between Parsl apps and the OpenAI API



Interface functions

- Functions to serve as *adapters* for Parsl apps
- For each Parsl app, we created:
  - *function_call_from_file* - receives the paths to the physical files
  - *function_call_from_futures* - receives the identifiers of the AppFutures on which the Parsl app depends
- Following the OpenAI specifications, we wrote function descriptions in JSON format for all the *function_call_from_files* and *function_call_from_futures*

```
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']
        }
    }
]
```

Function-calling API

- The communication scheme with the OpenAI API consists of sending the 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 apps within the workflow

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>

User: Task scheduled with AppFuture id: future_5_run_vcf_transform '
Now what?

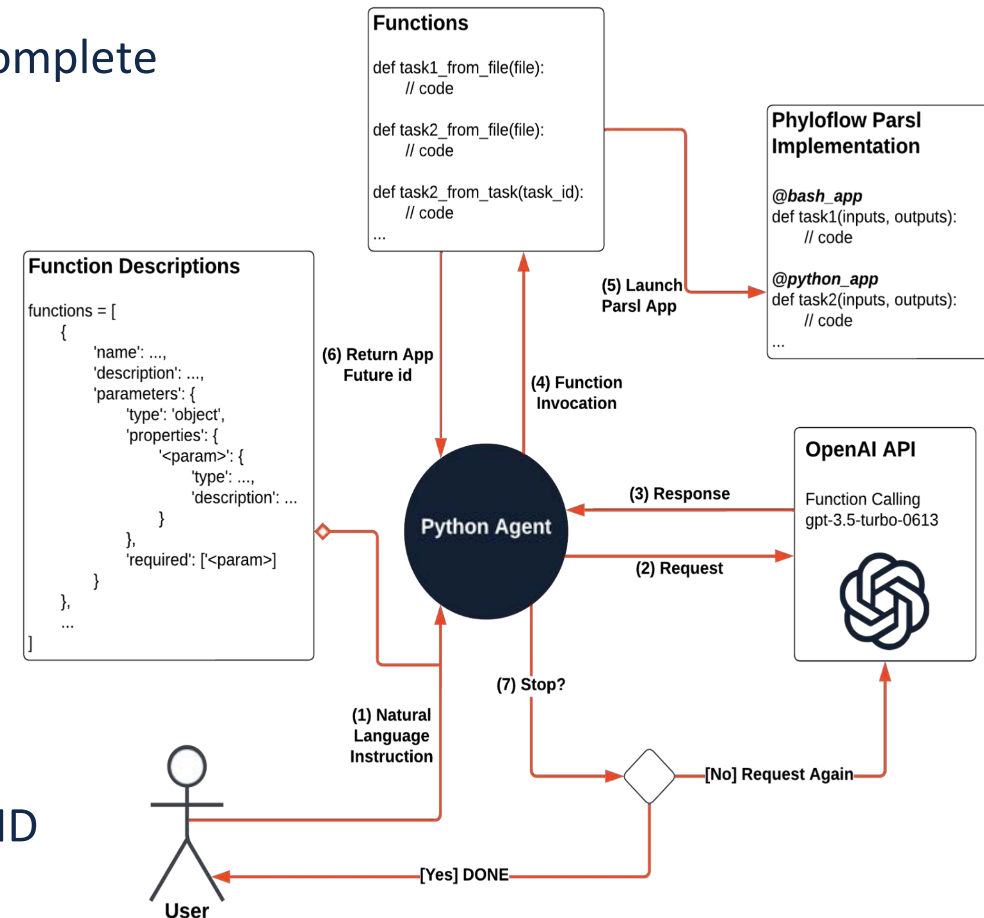
Function Calling
Function Name: fcall_pyclone_vi_from_futures
Function Args: {'vcf_future_id': 'future_5_run_vcf_transform'}
<AppFuture at 0x7f9072014490 state=pending>

User: Task scheduled with AppFuture id: future_6_run_pyclone_vi '
Now what?

DONE

Chaining apps

- We need to chain the execution of several Parsl apps to generate complete workflow executions
- To do this, we add *context* and make successive API calls
- API responds to each call with its choice of function to call
- Function is executed, immediately returns ID linked to AppFuture
- Add two new messages to next API request
    - First partially includes section of API's previous response message with the choice of the function to call
    - Second is a new user message with ID assigned to newly executed Parsl app
- Lets AI understand which step it is in, relative to user's instructions; can execute subsequent steps with access to scheduled AppFuture ID
- Repeated until API response include 'stop' flag



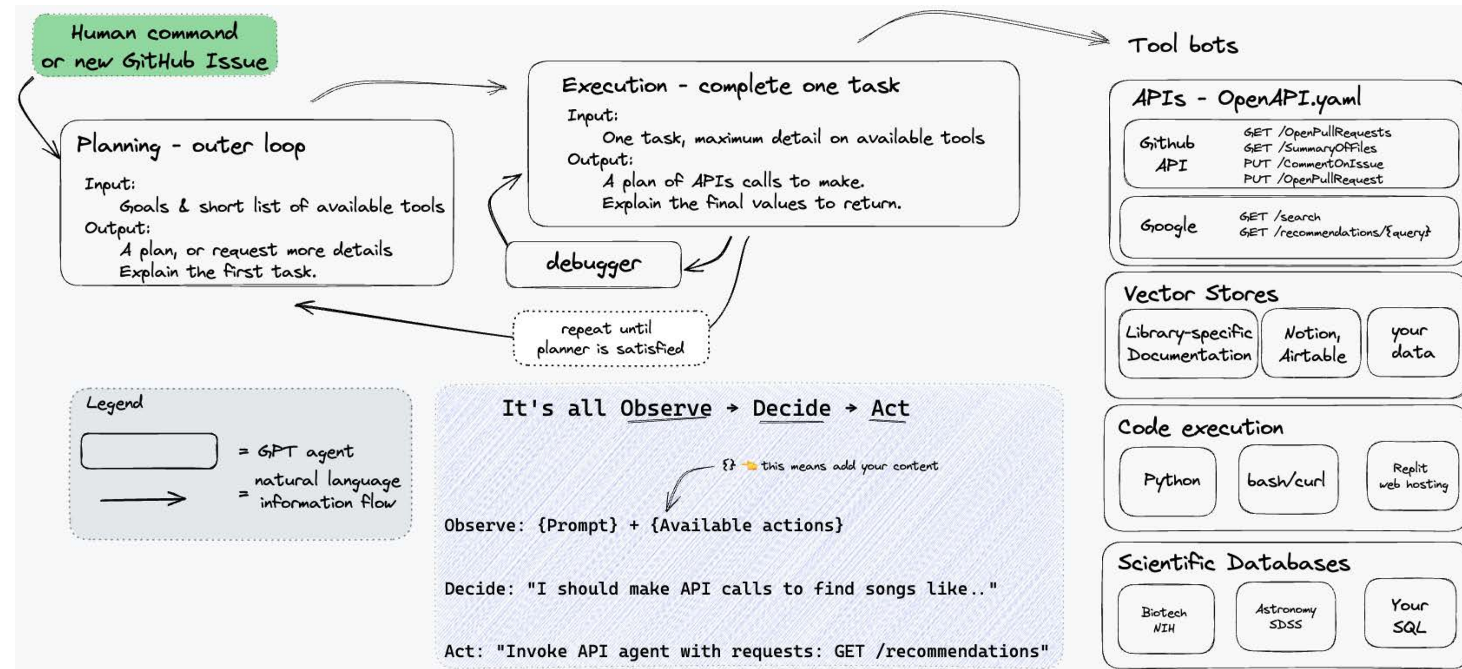
Next-gen workflow engine

Limitations of current implementation

- Exceptions are not handled: if the API selects an incorrect function, the program can’t recover from the failure
- Composing more complex workflows may hit the token limit, e.g., 128K tokens for GPT-4

Proposal for next-gen workflow engine

- 3 AI agents — *planner*, *executor*, *debugger* use LLM to process textual input, either to execute a task or to analyze & validate 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



References

- Phyloflow: https://github.com/ncsa/phyloflow
- Parsl: https://parsl-project.org/
- Langchain: https://python.langchain.com/docs/
- OpenAI API: https://platform.openai.com/docs/api-reference
- Function calling: https://openai.com/blog/function-calling-and-other-api-updates

- Our implementation: https://github.com/grimloc-aduque/Phyloflow-Parsl-Implementation