Exploring Python Metaprogramming

Final Year Project CA Report

Submitted by

Foo Shi Yu

Under the Supervision of

Prof. Richard Ma

School of Computing

National University of Singapore

November 2024

Contents

1 Introduction

1.1 Motivation

In recent years, the fields of data analytics and machine learning have experienced exponential growth, largely influenced by the increasing availability of data and computational resources. Python has emerged as a dominant programming language in these domains due to its simplicity, readability, and an extensive ecosystem of libraries and frameworks.

Leveraging Python's strengths in these areas, there is a brewing plan to design and implement a next-generation distributed streaming framework. This envisioned framework aims to support applications like data analysis and machine learning by enabling efficient, scalable, and stateful processing of streaming data in a distributed environment. However, while the framework itself is significant, it serves more as a backdrop for this project rather than its main focus.

The true motivation behind this project lies in exploring and documenting one of Python's most powerful features: metaprogramming. Metaprogramming is the ability of a program to manipulate itself or other programs as data. By delving into metaprogramming techniques and patterns, we aim to leverage these capabilities in the design and implementation of the framework, thereby showcasing how metaprogramming can enhance complex systems.

1.2 Constraints and Assumptions

A notable challenge lies in the undefined architecture of the framework, complicating efforts to align metaprogramming exploration with its specific needs. This ambiguity adds risk, as the exploratory nature of the project may lead to areas that don't directly support the final design. Although a broad approach can foster innovation, it's essential to focus resources on elements that will enhance the framework's functionality. Additionally, while metaprogramming is powerful, it introduces a level of complexity that can make code harder to read and maintain if not used appropriately. Striking a balance between harnessing metaprogramming's capabilities and maintaining code clarity is therefore essential.

Several key assumptions guide the project. Python was chosen for its extensive use in data analytics and machine learning, despite possible performance issues in distributed systems. This decision leverages Python's versatility and its ecosystem's support for rapid development. Moreover, it is assumed that the metaprogramming patterns and techniques observed in existing projects can be successfully transferred to and applied within the new framework's design.

2 Analysis of Open Source Projects

This section examines two open-source projects, stateflow and SGLang, which effectively utilize Python metaprogramming to achieve advanced functionality. The primary goal of analyzing these projects is not to understand their inner workings but rather to identify metaprogramming techniques and patterns used.

> Note
> - This section assumes basic understanding of metaprogramming. Refer to this page[1] for a short introduction.
> - Code snippets are heavily truncated and simplified from the source for illustrative purposes.

2.1 Analysis of stateflow

2.1.1 Introduction

Stateflow[2] is a framework that transforms object-oriented Python classes into distributed dataflows, enabling stateful operations in cloud applications. Developers simply annotate their Python classes with `@stateflow`, after which the framework processes these classes through multiple stages of transformation. The pipeline begins with Abstract Syntax Tree (AST) static analysis to extract class variables and methods, followed by identifying inter-class interactions to create function call graphs. Areas requireing remote function calls are identified to generate appropriate splits in user written code. For example, in Figure 1, the `buy_item` method in the `User` entity references another entity `Item`, resulting in remote calls at runtime. The framework then splits the method into five parts, allowing execution in an event-based manner without the need to block.

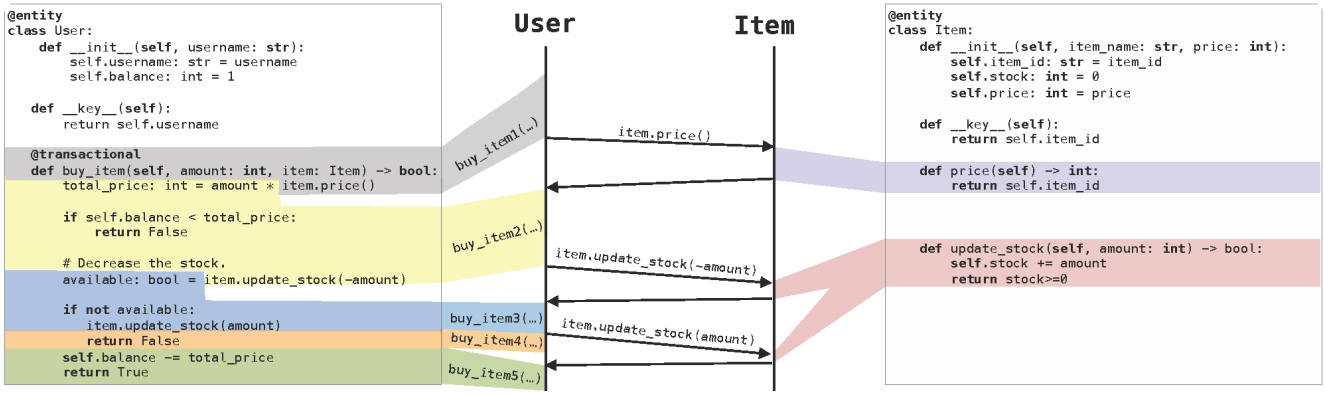

Figure 1: Two stateful entities: User and Item

This process ultimately results in an intermediate representation which can then be deployed and executed on various dataflow systems. The system's approach is particularly notable as it allows developers to write standard object-oriented Python code while automatically handling the complex transformation into deployable dataflow graphs. (Psarakis et al. 2023)

[1] https://striped-honeydew-d1f.notion.site/Python-Metaprogramming-Notes-0008aaaf3aaf4d5992297ed5387984c8
[2] https://github.com/delftdata/stateflow

2.1.2 Metaprogramming Techniques

2.1.2.1 Decorators for Class Transformation

The framework utilizes the `@stateflow` decorator to register classes and initialise the dataflow graph. This decorator transforms regular Python classes into distributed stateful operators by parsing and analyzing their source code.

```
registered_classes: List[ClassWrapper] = []
meta_classes: List = []

def stateflow(cls, parse_file=True):
    # Parse source with libcst...
    # Extract class description...
    # Create ClassDescriptor...

    # Register the class
    registered_classes.append(ClassWrapper(cls, class_desc))

    # Create a meta class
    meta_class = MetaWrapper(
        str(cls.__name__),
        tuple(cls.__bases__),
        dict(cls.__dict__),
        descriptor=class_desc,
    )
    meta_classes.append(meta_class)

    return meta_class
```

The decorator leverages libcst[3] to parse the class's source code, generating an abstract syntax tree (AST) that it then analyzes to extract a ClassDescriptor with metadata like methods and state variables. Subsequently, the class is registered, and the MetaWrapper metaclass transforms it, replacing the original class with a newly created and transformed class.

2.1.2.2 Metaclasses for Instance Control and Method Interception

The MetaWrapper metaclass intercepts class instantiation and method calls, allowing the framework to manage state and behavior in a distributed context.

```
class MetaWrapper(type):
    def __new__(msc, name, bases, dct, descriptor: ClassDescriptor = None):
        msc.client: StateflowClient = None
        msc.asynchronous: bool = False
        dct["descriptor"]: ClassDescriptor = descriptor
        return super(MetaWrapper, msc).__new__(msc, name, bases, dct)
```

[3] https://libcst.readthedocs.io/en/latest/index.html

```
    # continued...
    def __call__(msc, *args, **kwargs) -> Union[ClassRef, StateflowFuture]:
        if "__call__" in vars(msc):
            return vars(msc)["__async_call__"](args, kwargs)

        if "__key" in kwargs:
            return ClassRef(
                FunctionAddress(FunctionType.create(msc.descriptor), kwargs["__key"]),
                msc.descriptor,
                msc.client,
            )
        ...
        # Creates a class event.
        create_class_event = Event(
            event_id, fun_address, EventType.Request.InitClass, payload
        )
        return msc.client.send(create_class_event, msc)
```

The metaclass overrides the `__call__` method to intercept instance creation. The `__key` in kwargs is an internal marker denoting whether an instance has been created on the server. If it already exists on the server, a `ClassRef` is returned to the client. `ClassRef` a reference that the client-side can interact with (i.e. call methods, get and update attributes). Otherwise, a call is sent to the server to create the class instance.

2.1.2.3 Abstract Syntax Tree (AST) Manipulation for Metadata Extraction

AST manipulation is used to extract detailed information about classes and methods, which is essential for building the dataflow graph.

```
class ExtractClassDescriptor(cst.CSTVisitor):
    def __init__(
        self, module_node: cst.CSTNode, decorated_class_name: str, expression_provider
    ):
      ...

    def visit_FunctionDef(self, node: cst.FunctionDef) -> Optional[bool]:
        """Visits a function definition and analyze it.

        Extracts the following properties of a function:
        1. The declared self variables (i.e. state).
        2. The input variables of a function.
        3. The output variables of a function.
        4. If a function is read-only.
        """
```

```python
    # continued...
    def visit_ClassDef(self, node: cst.ClassDef) -> Optional[bool]:
        """Extracts class name and prevents nested classes."""

    @staticmethod
    def create_class_descriptor(
        analyzed_visitor: "ExtractClassDescriptor",
    ) -> ClassDescriptor:
        state_desc: StateDescriptor = StateDescriptor(
            analyzed_visitor.merge_self_attributes()
        )
        return ClassDescriptor(
            class_name=analyzed_visitor.class_name,
            module_node=analyzed_visitor.module_node,
            class_node=analyzed_visitor.class_node,
            state_desc=state_desc,
            methods_dec=analyzed_visitor.method_descriptors,
            expression_provider=analyzed_visitor.expression_provider,
        )
```

The ExtractClassDescriptor class uses `CSTVisitor`[4] from libcst to analyze a Python class's source code and extract metadata needed for creating a stateful function. It visits class and function definitions within the AST of the code to gather key information, including class names, method details, and state attributes (e.g., variables associated with self). After parsing all relevant elements, it merges state attributes and generates a ClassDescriptor, encapsulating the class metadata and enabling the creation of a stateful function object.

2.2 Analysis of SGLang

2.2.1 Introduction

SGLang[5] introduces a domain-specific language (DSL) that integrates with Python to facilitate advanced prompting workflows, particularly for natural language processing tasks. It uses metaprogramming to implement its DSL, enabling the construction of computational graphs from high-level code and allowing for features like parallel execution and state management. A usage example is shown below. (Zheng et al. 2024)

```python
@function
def multi_dimensional_judge(s, path, essay):
  s += system("Evaluate an essay about an image.")
  s += user(image(path) + "Essay:" + essay)
  s += assistant("Sure!")
```

[4] https://libcst.readthedocs.io/en/latest/visitors.html#libcst.CSTVisitor
[5] https://github.com/sgl-project/sglang

```
    # continued...
  # Return directly if it is not related
  s += user("Is the essay related to the image?")
  s += assistant(select("related", choices=["yes", "no"]))
  if s["related"] == "no": return
  # ...

state = multi_dimensional_judge.run(...)
print(state["output"])
```

2.2.2 Metaprogramming Techniques

2.2.2.1 Custom Expression Classes and Operator Overloading

SGLang defines custom expression classes that represent DSL constructs, enabling natural syntax through operator overloading.

```
class SglExpr:
    def __add__(self, other):
        if isinstance(other, str):
            other = SglConstantText(other)
        return SglExprList([self, other])

class SglConstantText(SglExpr):
    def __init__(self, text):
        super().__init__()
        self.text = text
```

The `SglExpr` class serves as the base for all expressions in the DSL, providing essential structure for expression representation. By implementing operator overloading, specifically through the `__add__` method, `SglExpr` enables expressions to be combined using the + operator. This feature supports a natural syntax, allowing developers to write code that closely resembles natural language or mathematical expressions, enhancing readability and intuitiveness.

```
s += system("Evaluate an essay about an image.")
s += user(image(path) + "Essay:" + essay)
```

The expressions within `user(...)` are combined using the + operator. Each component in `(image(path), "Essay:", essay)` is an SglExpr or converted into one.

2.2.2.2 Tracing and Computational Graph Construction

SGLang traces the execution of functions decorated with `@function` to build a computational graph.

```
def function(func=None):
    if func:
        return SglFunction(func)
    def decorator(func):
        return SglFunction(func)
    return decorator
```

When a function is decorated with `@function`, the decorator transforms the regular function into an `SglFunction` object.

```
class SglFunction:
    def __call__(self, *args, **kwargs):
        from sglang.lang.tracer import TracingScope

        tracing_scope = TracingScope.get_current_scope()
        if tracing_scope is None:
            return self.run(*args, **kwargs)
        else:
            kwargs["backend"]
                = tracing_scope.tracer_state.backend
            return self.trace(*args, **kwargs)
```

`__call__`[6] is overridden allowing the class to intercept execution. Depending on whether the tracing process is complete or not, the class either runs the function or continues the trace.

2.2.2.3 Building a Computational Graph:

The tracing process traces the full execution flow of a function, building a computational graph for execution when the function is run. From `SglFunction`, the `self.trace` call leads to the execution of `trace_program`

```
def trace_program(program, arguments, backend):
    #...
    # Trace
    tracer = TracerProgramState(backend, arguments, only_trace_prefix=False)
    with TracingScope(tracer):
        tracer.ret_value = program.func(tracer, **arguments)
    return tracer

class TracerProgramState(ProgramState):
    def __iadd__(self, other):
        self._execute(other)
        return self

    def _execute(self, other):
        if isinstance(other, SglExpr):
            self.nodes.append(other)
```

The tracer's operation interception works by overriding the `__iadd__`[7] method to capture += operations, preventing their immediate execution, and instead recording these expressions in `self.nodes`. This recording process effectively builds a computational graph from the collected expressions.

[6] https://docs.python.org/3/reference/datamodel.html#object.__call__
[7] https://docs.python.org/3/library/operator.html#operator.__iadd__

2.3 Common Themes and Patterns

2.3.1 Use of Intermediate Representations

Both projects utilize an intermediate representation (IR) to abstract the behavior of the system. Stateflow constructs a dataflow graph representing classes and methods as distributed operators, while SGLang builds a computational graph from traced expressions to represent execution flow.

2.3.2 Metaprogramming Techniques

The frameworks employ several key metaprogramming techniques. Decorators are used to transform and enhance classes and functions, while metaclasses and operator overloading control instance creation and enable natural syntax. AST manipulation and tracing are utilized to analyze and record code structures and execution for dynamic behavior, complemented by dynamic method and attribute handling for intercepting and managing method calls and state.

2.3.3 Implications for Framework Design

The implementation of these patterns has significant implications for framework design. Metaprogramming effectively abstracts the complexity of distributed systems, making them more accessible to developers. The use of operator overloading and decorators leads to more intuitive and expressive interfaces, while AST manipulation and execution tracing enable dynamic adaptation of code, particularly beneficial in distributed environments.

3 Practical Look at Metaprogramming

To gain an appreciation of Python's metaprogramming capabilities, a simple distributed task execution framework was implemented to help distill and showcase some techiniques identified in the previous analysis. While this demo framework itself provides limited practical utility, its primary purpose is to showcase how Python's metaprogramming features can be leveraged to create intuitive APIs, domain-specific languages (DSLs), and flexible runtime behaviors. The source code for the demo is available here[8].

3.1 Overview

At its core, the framework allows developers to define computational tasks using simple Python functions and execute them in a distributed manner. The system leverages metaprogramming to transform these simple function definitions into distributed tasks with minimal boilerplate code.

Consider this simple task definition:

```python
@task
def add_numbers(a: int, b: int) -> int:
    """Add two numbers together"""
    return a + b
```

This seemingly simple code triggers a chain of metaprogramming operations:

1. A new task class is created in place of the function
2. The metadata of the function is preserved and stored in a task registry
3. Type information is extracted for runtime validation
4. Custom worker code is generated for distributed execution of tasks

Figure 2 provides a good visual representation of the whole flow.

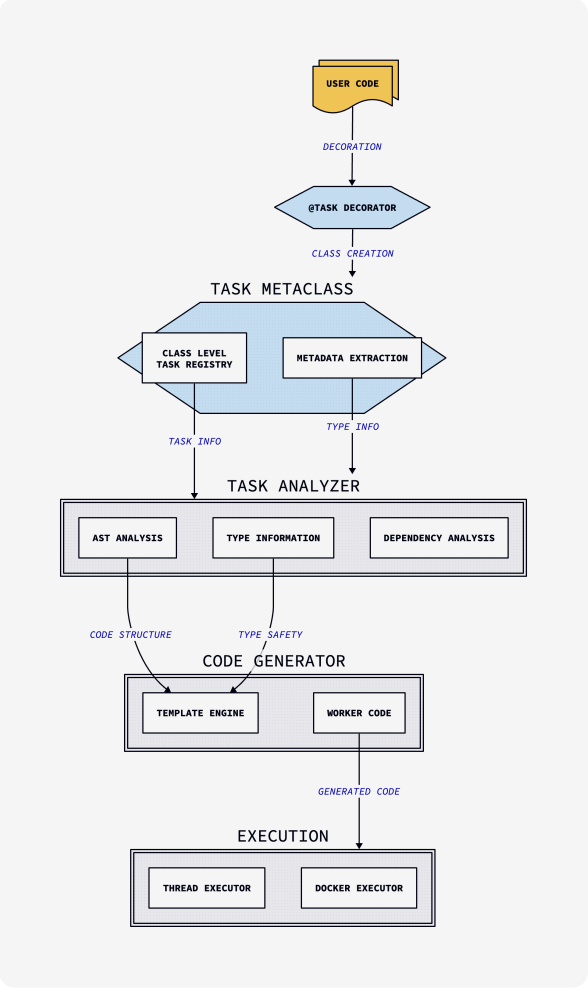

Figure 2: Code analysis and transformation flow

[8] https://github.com/CFSY/fyp-python-metaprogramming/tree/main/demos/demo1

Upon startup, a web interface is provided, exposing the user defined functions for execution.

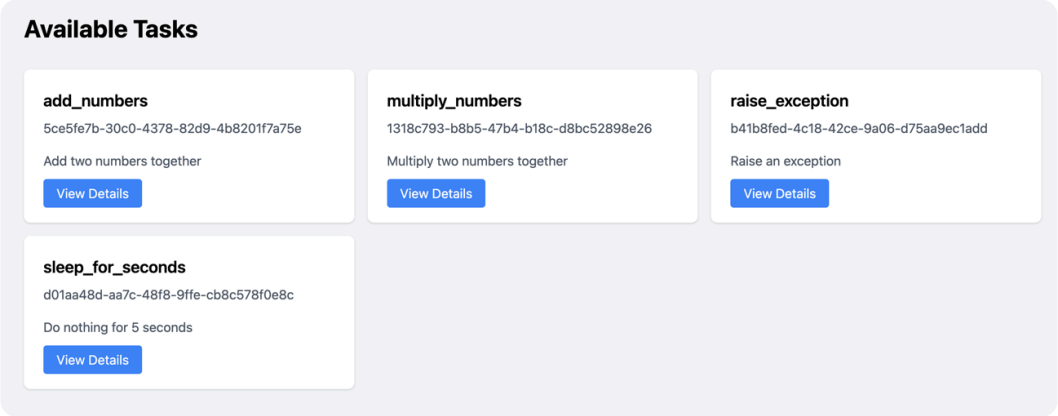

Figure 3: Framework generated web UI

The tasks can be executed through the web interface. Task input parameters are automatically detected and exposed for input.

There are currently two executors provided, the thread and docker executors.

- Thread: spins up a new thread for execution
- Docker: spins up a new docker container for execution

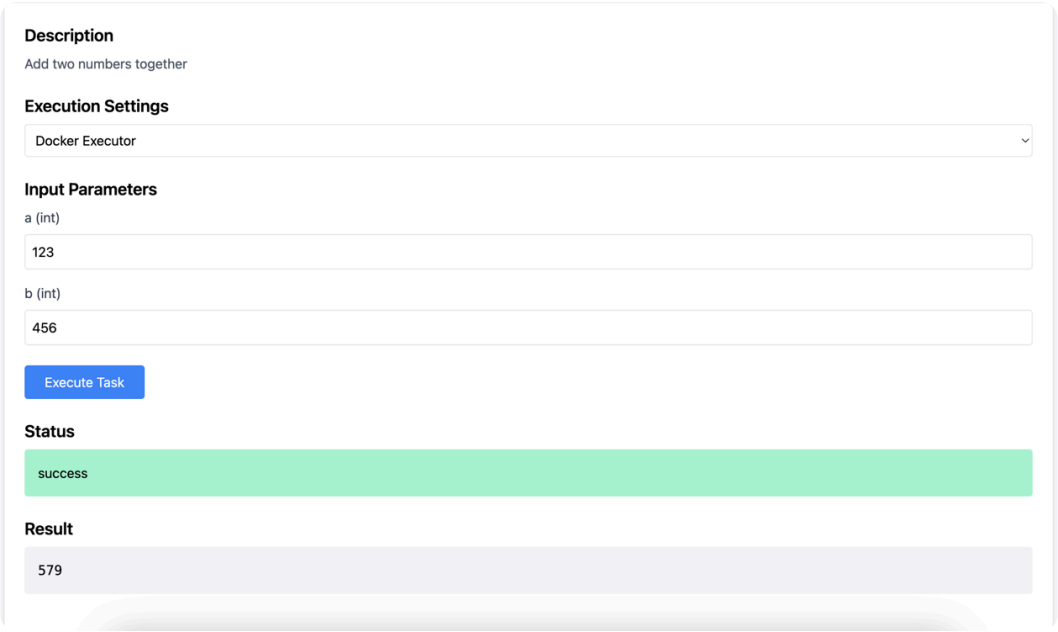

Figure 4: Framework generated task execution UI

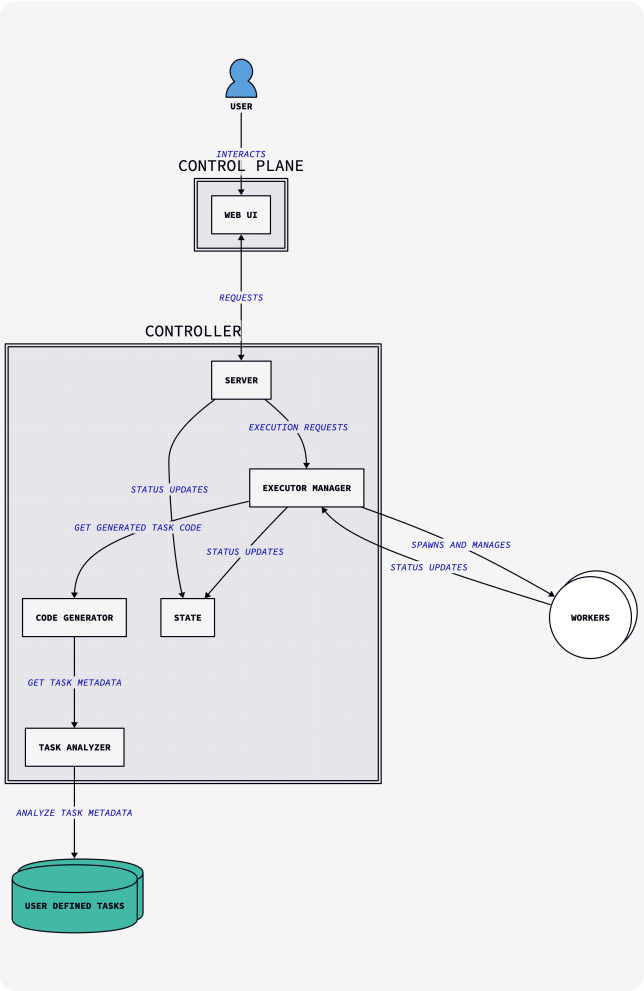

Figure 5: System architecture

In the case where the task is executed with the Docker executor, the following occurs:

1. An execution request is sent from the web UI to the server
2. The execution manager receives the request and code generation
3. Custom worker code is generated, containing only code relevant to the task being executed
4. Docker container orchestration configurations are generated and executed
5. The task is executed on the worker container
6. Meanwhile, the web UI automatically polls for the execution results and status
7. Execution is completed and everything is cleaned up

Figure 4 provides a high level system architecture of all the moving parts during task execution.

3.2 A Closer Look

3.2.1 Metaclasses and Class Generation

The `@task` decorator and `Task` metaclass form the foundation of the framework's metaprogramming capabilities. When a function is decorated with `@task`, the following code will run:

```python
def task(func):
    """Decorator to create a task class from a function"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    # Create a new task class dynamically
    task_name = f"{func.__name__.title()}Task"
    class DynamicTask(metaclass=Task):
        execute = staticmethod(func)
        __module__ = func.__module__

    DynamicTask.__name__ = task_name
    return DynamicTask
```

This creates a new class that inherits the Task metaclass that the framework defined.

The original function is also stored in the execute attribute of the class for later analysis and use.

```python
class Task:
    """Task metaclass for creating task definitions"""
    _registry = {}
    def __new__(cls, name, bases, attrs):
        new_cls = super().__new__(cls, name, bases, attrs)
        if 'execute' in attrs:
            metadata = TaskMetadata(attrs['execute'])
            new_cls._metadata = metadata
            Task._registry[metadata.id] = new_cls
        return new_cls

class TaskMetadata:
    def __init__(self, func):
        self.func = func
        self.id = str(uuid.uuid4())
        self.name = func.__name__
        self.signature = inspect.signature(func)
        self.doc = func.__doc__
            or "No description available"

    @property
    def input_schema(self) -> Dict[str, Type]:
      return {
        name: param.annotation
          if param.annotation != inspect.Parameter.empty
          else Any
        for name, param in
            self.signature.parameters.items()
      }
```

The metaclass serves multiple purposes:

- Maintains a global registry of all tasks
- Provides introspection capabilities through the TaskMetadata class
- Allows enforcement of task interface requirements with `input_schema`
- Enables runtime modification of task behavior

This approach demonstrates how metaclasses can create powerful abstractions that feel natural to Python developers while providing sophisticated functionality under the hood.

3.2.2 Type Annotations and Runtime Reflection

Python's type annotation system is leveraged extensively for both documentation and runtime behavior:

```python
class TaskAnalyzer:
    @staticmethod
    def analyze_task(task_cls: Type) -> Dict:
        if not hasattr(task_cls, '_metadata'):
            raise ValueError(f"Invalid task class: {task_cls}")
        metadata = task_cls._metadata
        source = inspect.getsource(metadata.func)

        # Extract type information and validate
        input_schema = metadata.input_schema
        for param_name, param_type in input_schema.items():
            if param_type == inspect.Parameter.empty:
                raise ValueError(f"Missing type annotation for parameter: {param_name}")
        return {...}
```

The framework uses `inspect.signature()` to extract type information at runtime, enabling:

- Automatic input validation
- Dynamic UI generation
- Type-safe serialization for execution
- Documentation generation

This showcases how Python's type system can be used beyond static type checking, becoming a powerful tool for runtime behavior modification.

3.2.3 3. Dynamic Code Generation

One of the most powerful metaprogramming techniques demonstrated is dynamic code generation. The framework generates worker code on-the-fly using:

- Jinja2[9] templates for code structure
- AST manipulation for code analysis
- Dynamic module loading for execution

This approach allows for:

- Optimal worker code with minimal dependencies
- Environment-specific optimizations
- Runtime code modification based on execution context

3.3 Project Insights

I believe this demo project has shown how metaprogramming can bridge the gap between simplicity and sophistication. The techniques explored here, from metaclasses to AST manipulation, represent just the beginning of what's possible. As we look toward implementing more complex features like workflow orchestration, dependency management, and distributed computing patterns, these metaprogramming patterns will prove invaluable in maintaining clean APIs while handling increasing complexity under the hood.

[9] https://jinja.palletsprojects.com/en/stable/

4 Future Directions

The analysis and demo project has laid the groundwork for several exciting metaprogramming explorations. In the following months, more focus will be placed on these potential areas:

1. Advanced DSL Development
   - Create a declarative DSL for defining task workflows, dependencies, and execution patterns.
   - Explore alternatives for implementing a DSL like chainable APIs or context managers. For example:

```
@workflow
def data_processing():
    with parallel():
        task1 = process_data(input="data1.csv")
        task2 = process_data(input="data2.csv")

    with sequential():
        combined = combine_results(task1.output, task2.output)
        validated = validate_data(combined.output)

    with conditional(validated.success):
        save_results(validated.output, path="results.csv")

    with error_handler():
        notify_admin(error=current_error)
```

2. AST-Based Optimization
   - Automatic parallelization through code analysis
   - Dead code elimination
   - Resource usage optimization
3. Dynamic Protocol Generation
   - Automatic API generation, maybe even for different communication protocols
   - Runtime protocol negotiation and adaptation
4. Documentation and Knowledge Base
   - Comprehensive documentation of metaprogramming patterns
   - Best practices and anti-patterns
   - Performance implications and trade-offs
   - Case studies and practical examples

Bibliography

Psarakis, K., Zorgdrager, W., Fragkoulis, M., Salvaneschi, G., & Katsifodimos, A. (2023,). Stateful Entities: Object-oriented Cloud Applications as Distributed Dataflows. https://arxiv.org/abs/2112.00710

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., & Sheng, Y. (2024,). SGLang: Efficient Execution of Structured Language Model Programs. https://arxiv.org/abs/2312.07104