B.Comp. Dissertation

Exploring Python Metaprogramming

By

Foo Shi Yu

Department of Computer Science
School of Computing

National University of Singapore

2024/2025

B.Comp. Dissertation

Exploring Python Metaprogramming

By

Foo Shi Yu

Department of Computer Science
School of Computing

National University of Singapore

2024/2025

Project No: H159600

Advisor: Associate Prof Richard Ma

Deliverables:

- Report: 1 PDF Document
- Code: 2 GitHub Repositories
  - Repository 1: https://github.com/CFSY/meta_task_runner
  - Repository 2: https://github.com/CFSY/meta-reactive
- Documentation: 1 GitHub Pages Site
  - URL: https://cfsy.github.io/fyp-docs/

Abstract

Python's widespread adoption in data analytics and machine learning necessitates robust frameworks, yet designing intuitive and powerful APIs for these frameworks remains challenging. Python's metaprogramming capabilities, such as decorators, metaclasses, and Abstract Syntax Tree (AST) manipulation, offer potential solutions for enhancing API design by reducing boilerplate and enabling more declarative interfaces. However, these techniques can also introduce complexity, impacting code readability, maintainability, and debuggability if not applied appropriately. This project explores the practical application of Python metaprogramming to API design.

Subject Descriptors:

D.1.5 Object-oriented Programming

D.2.4 Software/Program Verification

D.2.10 Design

D.2.13 Reusable Software

D.3.2 Language Classifications

D.3.3 Language Constructs and Features

Keywords:

Python, Metaprogramming, API Design, Frameworks, Decorators, Metaclasses, Abstract Syntax Trees (AST), Code Generation, Reactive Programming, Developer Experience, AST Analysis

Implementation Software and Hardware:

macOS Ventura 13.x, Python 3.9+, libcst, pydantic, Quart, Hypercorn, aiohttp, 2021 14-inch MacBook Pro (M1 Pro).

Contents

1 Introduction

> **Note**
> Sections 1 - 3 are brought over from the CA report. Jump stright to section 4 for the second-half of the project.
>
> For a refresher of metaprogramming basics, refer to the documentation site[1].

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.

[1] https://cfsy.github.io/fyp-docs/docs/category/metaprogramming

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[2] 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[3] 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.

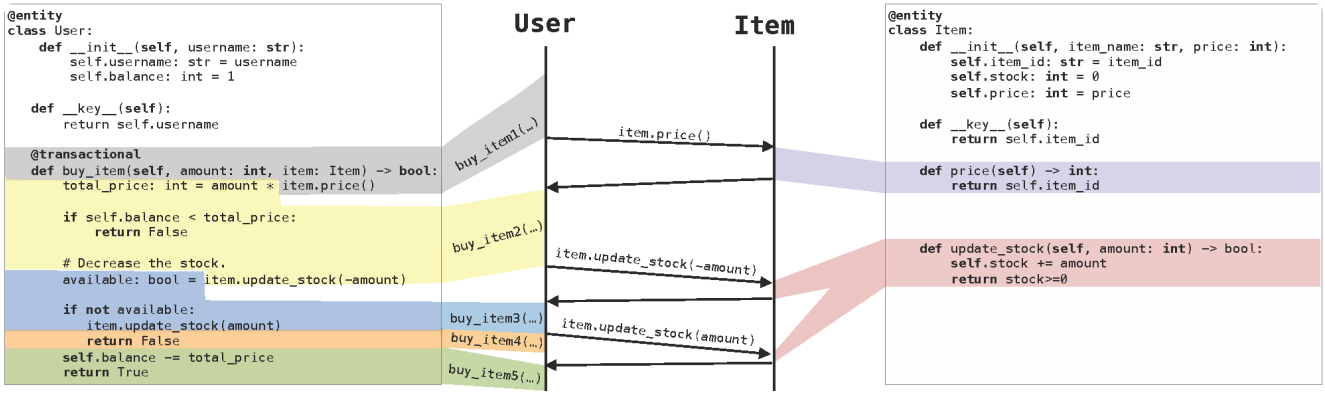

Figure 1: Two stateful entities: User and Item

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)

[2] https://cfsy.github.io/fyp-docs/docs/category/metaprogramming
[3] 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[4] 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)
```

[4] 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`[5] 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[6] 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!")
```

[5] https://libcst.readthedocs.io/en/latest/visitors.html#libcst.CSTVisitor
[6] 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__`[7] 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__`[8] 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.

[7] https://docs.python.org/3/reference/datamodel.html#object.__call__
[8] 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[9].

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.

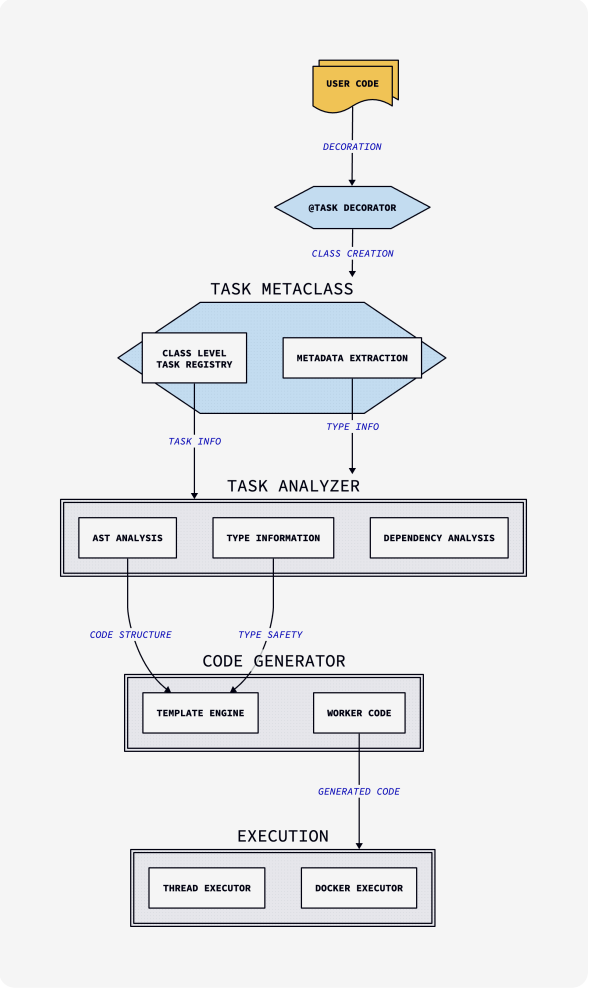

Figure 2: Code analysis and transformation flow

[9] https://github.com/CFSY/meta_task_runner

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

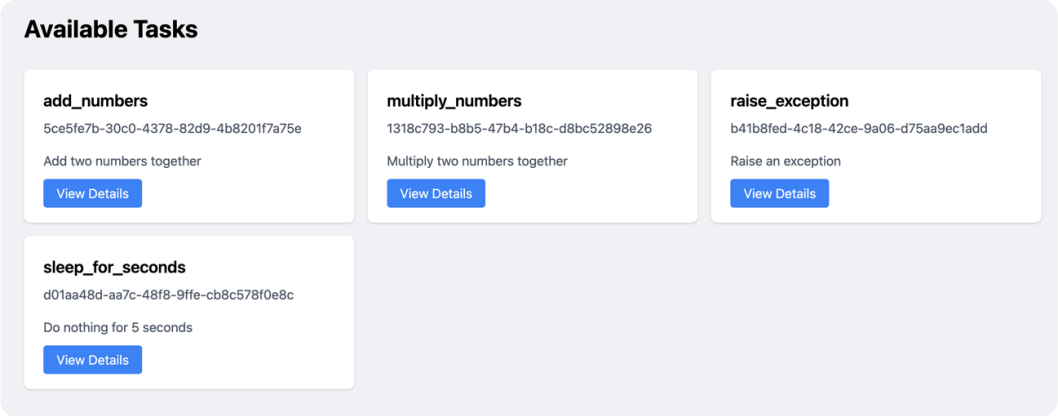

Figure 3: Framework generated web UI

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

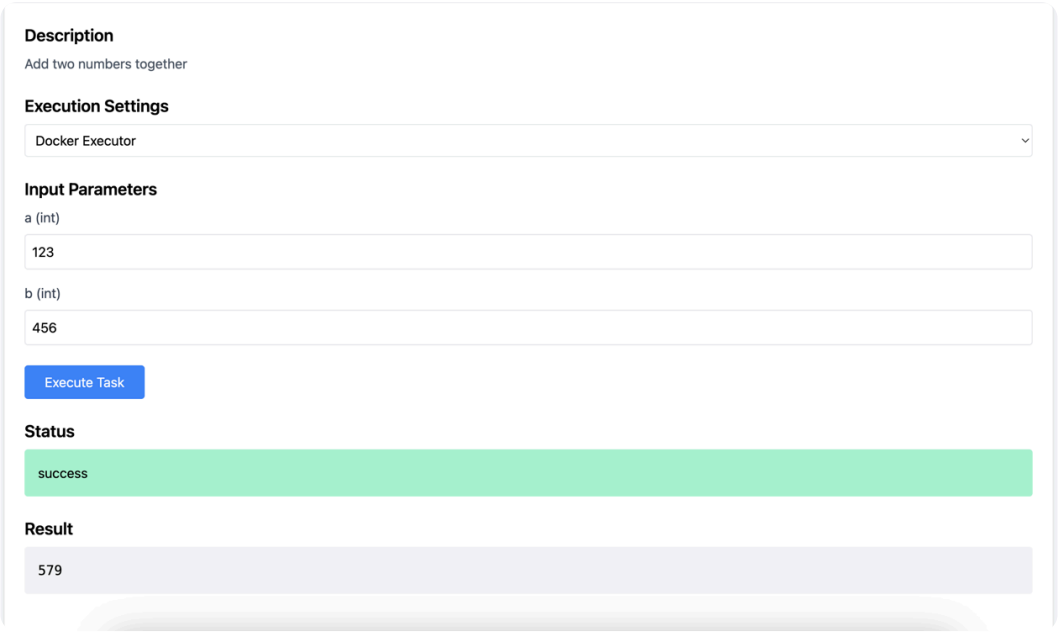

Figure 4: Framework generated task execution UI

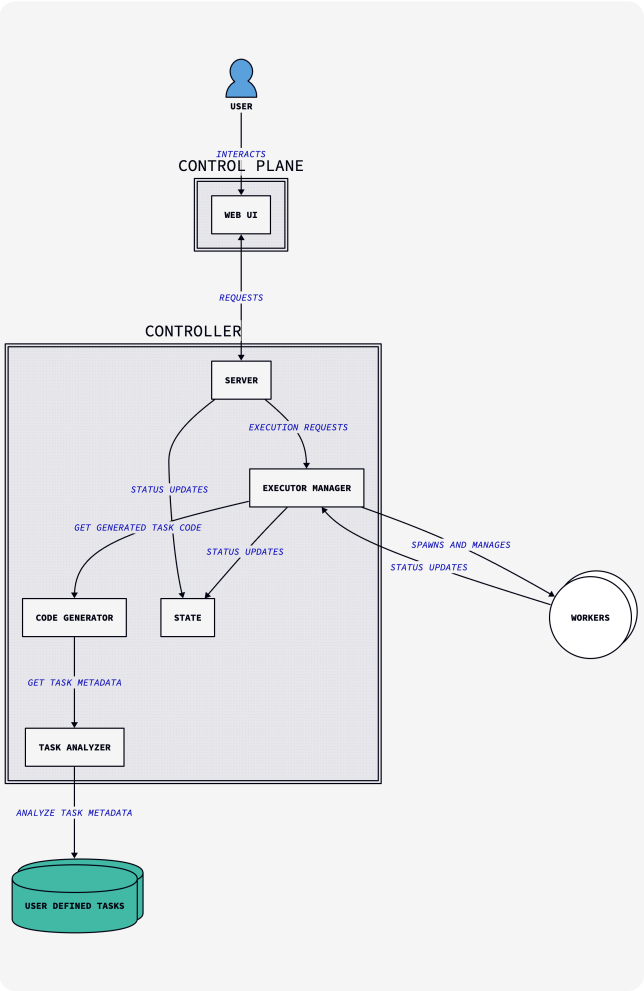

Figure 5: System architecture

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[10] 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.

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

4 The Reactive Framework

> **Note**
> The following sections **4** - **9** are also available on the documentation site[11]. The contents are largely identical and the site should provide a better reading experience.

Following the initial exploration of metaprogramming techniques in the first half of the project, the next logical step was to apply the concepts in a more complex environment. To this end, the "Reactive Framework" was developed during the second half of the project. This framework serves a dual purpose:

1. **Embodying Core Concepts for Future Work:** While not a full-fledged stream processing engine, the reactive framework intentionally incorporates core concepts relevant to such systems. Features like `Collections` representing data sources, `Mappers` defining transformation pipelines, a `ComputeGraph` managing dependencies and propagation of changes, and `Resources` exposing data streams via Server-Sent Events (SSE) provide a tangible link to the broader goal of building distributed streaming applications. It acts as a practical testbed for API design philosophies applicable to more complex streaming scenarios.

2. **Providing a Platform for Metaprogramming Comparison:** The primary motivation was to create a concrete example demonstrating the practical benefits and tradeoffs of applying metaprogramming to API design. To achieve this, the framework was intentionally designed with **two distinct APIs** offering identical core functionality:
   - A **Classic API**, adhering to traditional Python object-oriented principles.
   - A **Metaprogramming API**, leveraging decorators and other metaprogramming techniques for a more concise and declarative developer experience.

   This dual-API approach allows for a direct, side-by-side comparison, highlighting how metaprogramming can simplify complex interactions and reduce boilerplate.

[11] https://cfsy.github.io/fyp-docs/docs/category/reactive-demo

5 Introduction to the Reactive Framework

The Reactive Backend Framework is a Python-based system for building reactive applications where data and computation outputs can be efficiently streamed to clients in real-time. This framework leverages reactive programming principles to create responsive, data-driven applications with minimal boilerplate code.

5.1 Core Concepts

The framework is built around several key concepts that work together to create an efficient reactive computation engine:

5.1.1 Reactive Programming

Reactive programming is a declarative programming paradigm focused on data flows and propagation of changes. In this paradigm, when source data changes, the changes automatically propagate to dependent computations and outputs.

Our framework implements this paradigm by:
1. Building a dependency graph of data and computations
2. Efficiently propagating changes through this graph
3. Streaming updates to clients via Server-Sent Events (SSE)

5.1.2 Collections

Collections are the core data structures over which reactive computations operate. They serve as the vertices of the reactive computation graph.

5.1.3 Mappers

Mappers define transformations between collections. They form the edges of the reactive computation graph, specifying how data is transformed from one collection to another.

There are two primary types of mappers:
1. **OneToOneMapper**: Transforms each value in a collection to a new value, preserving the key
2. **ManyToOneMapper**: Aggregates multiple values with the same key into a single value

5.1.4 Compute Graph

The compute graph manages the dependencies between collections and ensures that changes propagate efficiently through the system.

The compute graph:

- Tracks dependencies between collections
- Invalidates affected collections when source data changes
- Performs recomputation in an efficient topological order
- Prevents circular dependencies and redundant computations

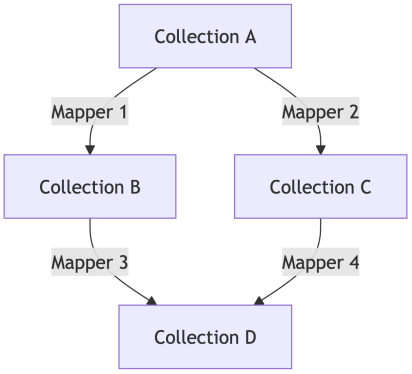

Figure 6: Compute Graph

5.1.5 Resources

Resources expose collections to external clients, providing a way to create, access, and subscribe to reactive data sources.

Key features of resources:

- Resources are parameterized by a Pydantic model
- Each resource instantiation creates a configured collection
- Resources can be streamed to clients via Server-Sent Events
- Multiple clients can subscribe to the same resource instance

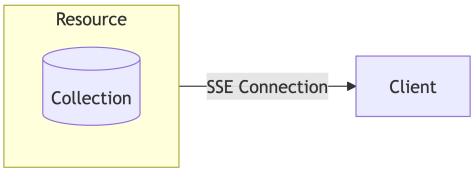

Figure 7: Resource

5.1.6 Server-Sent Events (SSE)

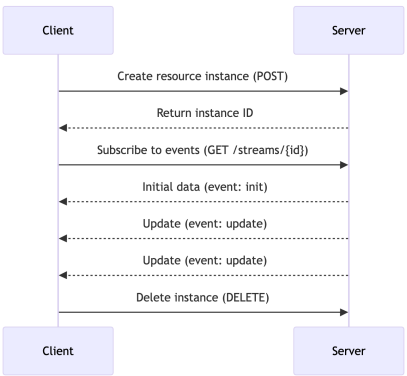

Figure 8: Server Sent Events

The framework uses Server-Sent Events to stream reactive updates to clients. Clients first perform a POST request to the resource endpoint to instantiate a resource, the server will respond with an instance ID. The client can then begin streaming data through SSE using the instance ID.

5.1.7 Overall Architecture

Here's a diagram showing how clients access resources through the service:

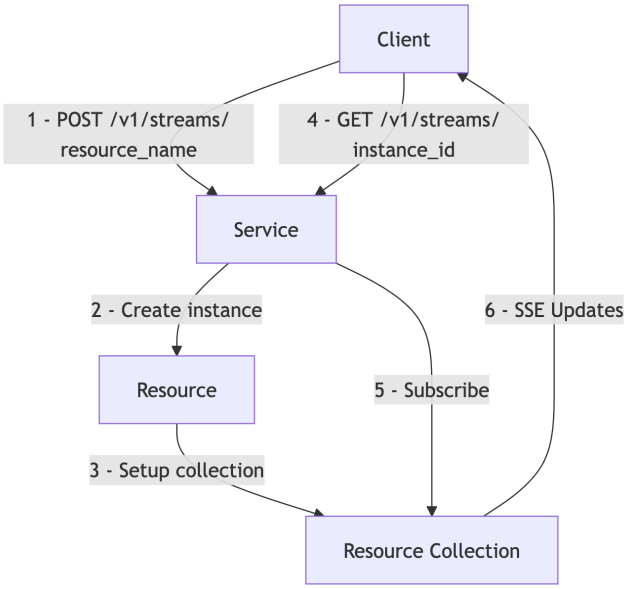

Figure 9: Access Flow

Here's a diagram showing the architecture of the service:

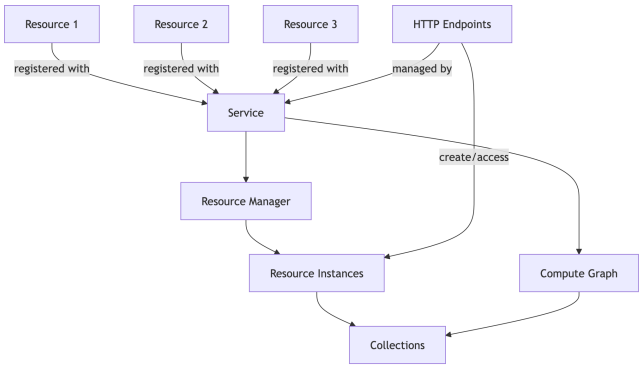

Figure 10: Overall Architecture

5.2 API Implementations

The framework provides two API implementations with identical functionality but different programming styles:

5.2.1 Classic API

The Classic API follows traditional object-oriented patterns with explicit class inheritance and method implementations. It provides a familiar interface for developers used to working with standard Python libraries.

5.2.2 Metaprogramming API

The Metaprogramming API leverages Python's advanced features like decorators, metaclasses, and AST analysis to provide a more concise and declarative interface. This API reduces boilerplate code and makes applications more readable while maintaining the same functionality.

6 Reactive Framework - Classic API

The Classic API was deliberately designed to mirror common practices found in established Python web frameworks and libraries. The goal was to provide a familiar, explicit, and object-oriented interface that would serve as a clear baseline for comparison against the Metaprogramming API. This section will walk you through the basics of using the Classic API.

6.1 Basic Concepts

The Classic API is built around several key classes:

- `ComputedCollection` - Collection data structure with reactive capabilities
- `Mapper` (with variants like `OneToOneMapper` and `ManyToOneMapper`) - Define transformations
- `Resource` - Expose collections to clients
- `Service` - Host and serve resources

6.2 Creating a Simple Application

> **Note**
> You can run this example locally. It is available under the `examples/basic`[12] directory.

Figure 11: Simple Example

Let's create a simple application. In this example, we have a simulated sensor that generates a value every second. The sensor value is kept in the `raw_data` collection.

```
# Create a service
service = Service("data_processor", port=8080)

# Create a collection for raw data
raw_data = ComputedCollection("raw_data", service.compute_graph)
```

[12] https://github.com/CFSY/meta-reactive/tree/main/examples/basic

We then define a mapper `MultiValueMapper` that multiplies the sensor value by a multiple and returns a formatted message.

```
# Define data models
class DataPoint(BaseModel):
    value: float
    timestamp: str

# Define a mapper
class MultiValueMapper(OneToOneMapper):
    def __init__(self, multiplier: float):
        self.multiplier = multiplier

    def map_value(self, value):
        if value is None:
            return None
        return {"value": value.value * self.multiplier, "timestamp": value.timestamp}
```

Lastly we define a resource `DataProcessorResource` that allows a client to stream data from a sensor.

```
# Define resource parameters
class ProcessorParams(ResourceParams):
    multiplier: float

# Define a resource
class DataProcessorResource(Resource):
    def __init__(self, data_collection, compute_graph):
        super().__init__(ProcessorParams, compute_graph)
        self.data = data_collection

    def setup_resource_collection(self, params):
        # Create a derived collection by mapping the input data
        multiplied_data = self.data.map(MultiValueMapper, params.multiplier)
        return multiplied_data
```

Additionally, we register the resource with the service under the name “processor”. The resource will then be made available under the “processor” endpoint.

```
# Create and register our resource
processor = DataProcessorResource(raw_data, service.compute_graph)
service.add_resource("processor", processor)
```

6.2.1 Connecting a Client

You can connect to this service using an HTTP client that supports Server-Sent Events.

First, we instantiate a resource with the parameters `{"multiplier": 3.0}`.

We then connect to the stream with the provided instance ID.

```
async def run_client():
    async with aiohttp.ClientSession() as session:
        # Create stream
        async with session.post(
            "http://localhost:8080/v1/streams/processor", json={"multiplier": 3.0}
        ) as response:
            stream_data = await response.json()
            stream_id = stream_data["instance_id"]
            print(f"Stream created with ID: {stream_id}")

        # Connect to stream
        async with session.get(
            f"http://localhost:8080/v1/streams/{stream_id}",
            headers={
                "Accept": "text/event-stream",
                "Cache-Control": "no-cache",
                "Connection": "keep-alive",
            },
        ) as response:
            async for line in response.content:
                if line.startswith(b"data: "):
                    data = json.loads(line[6:].decode("utf-8"))
                    print(f"Received: {data}")


if __name__ == "__main__":
    asyncio.run(run_client())
```

This is just a basic example. In the next sections, we'll explore each component in more detail and cover more advanced usage patterns.

6.3 Collections - Classic API

Collections are the fundamental data structures in the Reactive Framework. They store data as key-value pairs and form the vertices of the reactive computation graph.

6.3.1 Types of Collections

The framework contains two main types of collections:

1. **Base Collection** - The core data structure without reactive capabilities
2. **Computed Collection** - Extended with reactive computation capabilities

> **Note**
> Users should not have a need to use the base `Collection` directly. Instead, they should use the `ComputedCollection`, which are the primary way to interact with the framework.

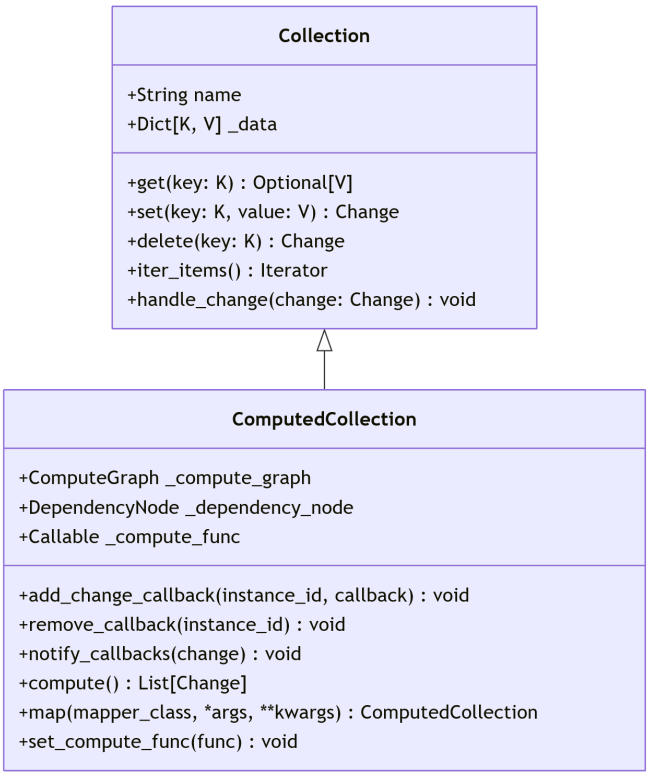

Figure 12: Collection Class Diagram

6.3.2 Creating Collections

6.3.2.1 Base Collection

```python
from reactive.core.collection import Collection

# Create a collection of string keys and integer values
collection = Collection[str, int]("my_collection")
```

6.3.2.2 Computed Collection

```python
from reactive.core.compute_graph import ComputeGraph, ComputedCollection

# Create a compute graph
compute_graph = ComputeGraph()

# Create a computed collection
collection = ComputedCollection[str, int]("my_computed_collection", compute_graph)
```

6.3.3 Working with Collections

6.3.3.1 Setting Values

```
# Set a value
collection.set("key1", 42)
collection.set("key2", 100)
collection.set("key3", 200)
```

6.3.3.1.0.0.0.1 Getting Values

```
# Get a single value
value = collection.get("key1")  # Returns 42 or None if key doesn't exist

# Get all values as a dictionary
all_data = collection.get_all()  # Returns {'key1': 42, 'key2': 100, 'key3': 200}
```

6.3.3.2 Deleting Values

```
# Delete a value
collection.delete("key1")
```

6.3.3.3 Iterating Over Items

```
# Iterate over all key-value pairs
for key, value in collection.iter_items():
    print(f"{key}: {value}")
```

6.3.4 Computed Collections

Computed collections extend the base collection with reactive capabilities:

6.3.4.1 Responding to Changes

When a computed collection is created, it registers itself with the compute graph:

```
# This collection will participate in reactive computations
computed_collection = ComputedCollection[str, int]("my_collection", compute_graph)
```

6.3.4.2 Using Mappers

You can define computations through mappers:

```
from reactive.classic.mapper import OneToOneMapper

class DoubleValueMapper(OneToOneMapper):
    def map_value(self, value):
        return value * 2

# Create a derived collection
doubled_collection = source_collection.map(DoubleValueMapper)
```

6.4 Mappers - Classic API

Mappers are a key component in the Reactive Framework that define transformations between collections. They form the edges of the reactive computation graph, specifying how data is transformed from source collections to target collections.

6.4.1 Types of Mappers

The framework provides two main types of mappers:

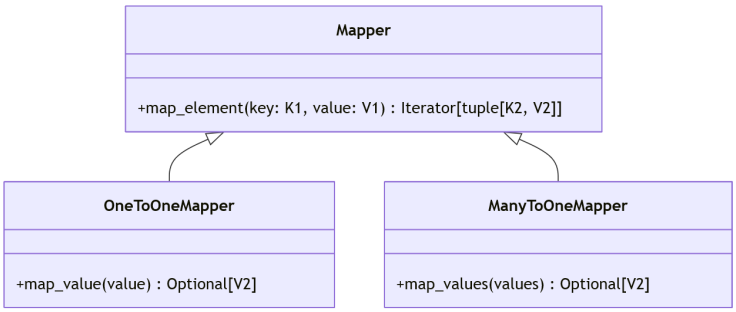

Figure 13: Mapper Class Diagram

1. **OneToOneMapper** - Transforms individual values, preserving keys
2. **ManyToOneMapper** - Aggregates multiple values into a single result

> **Note**
> The other mapper types were not implemented for simplicity

6.4.2 Creating Mappers

Mappers are implemented by subclassing one of the mapper types and implementing the appropriate mapping method.

6.4.2.1 OneToOneMapper

```python
from reactive.classic.mapper import OneToOneMapper
from typing import Optional

class DoubleValueMapper(OneToOneMapper[str, int, str, int]):
    def map_value(self, value: int) -> Optional[int]:
        return value * 2
```

6.4.2.2 ManyToOneMapper

```python
from reactive.classic.mapper import ManyToOneMapper
from typing import List, Optional

class AverageMapper(ManyToOneMapper[str, float, str, float]):
    def map_values(self, values: List[float]) -> Optional[float]:
        if not values:
            return None
        return sum(values) / len(values)
```

6.4.3 Using Mappers with Collections

Mappers are applied to collections using the `map` method, which creates a new computed collection:

```
# Create source collection
temperatures = ComputedCollection[str, float]("temperatures", compute_graph)

# Apply a mapper to create a derived collection
class CelsiusToFahrenheitMapper(OneToOneMapper[str, float, str, float]):
    def map_value(self, celsius: float) -> float:
        return (celsius * 9/5) + 32

fahrenheit_temps = temperatures.map(CelsiusToFahrenheitMapper)
```

You can also pass parameters to mappers:

```
class ScaledMapper(OneToOneMapper[str, float, str, float]):
    def __init__(self, scale_factor: float):
        self.scale_factor = scale_factor

    def map_value(self, value: float) -> float:
        return value * self.scale_factor

# Create a collection with values scaled by 2.5
scaled_temps = temperatures.map(ScaledMapper, 2.5)
```

Note

Mapper parameters must be explicitly added to the mapper through the constructor. This is necessary for the framework's reactivity. Internally, the framework checks a mapper's constructor arguments for reactive components and adds them to the compute graph for dependency tracking.

6.5 Resources - Classic API

Resources are the interface between your application's reactive data and external clients. They expose collections through HTTP endpoints and allow clients to subscribe to data changes via Server-Sent Events (SSE).

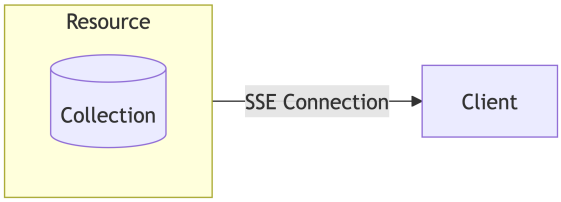

Figure 14: Resource

6.5.1 Creating Resources

1. Define Parameter Model

Resources can accept parameters that helps you customize its behavior. To start, define a Pydantic model for your resource parameters:

```
from pydantic import BaseModel
from reactive.classic.resource import ResourceParams

# Define resource parameters
class ProcessorParams(ResourceParams):
    multiplier: float
```

2. Implement Resource Class

Then, create a resource class that sets up the reactive collection:

```
from reactive.classic.resource import Resource
from reactive.core.compute_graph import ComputedCollection

# Define a resource
class DataProcessorResource(Resource):
    def __init__(self, data_collection, compute_graph):
        super().__init__(ProcessorParams, compute_graph)
        self.data = data_collection

    def setup_resource_collection(self, params):
        # Create a derived collection by mapping the input data
        multiplied_data = self.data.map(MultiValueMapper, params.multiplier)
        return multiplied_data
```

6.5.2 Registering Resources with a Service

Register your resource with a service to make it available to clients:

```
from reactive.classic.service import Service
from reactive.core.compute_graph import ComputedCollection

# Create a service
service = Service("data_processor", port=1234)

# Create a collection for raw data
raw_data = ComputedCollection("raw_data", service.compute_graph)

# Create and register our resource
processor = DataProcessorResource(raw_data, service.compute_graph)
service.add_resource("processor", processor)
```

6.5.3 Accessing a resource

There are two main steps for a client to access a resource:

1. Resource Instantiation

To instantiate a resource, make a POST request to the endpoint `/v1/streams/{resource_name}`. `resource_name` is the name provided during registration, in our example it is `"processor"`. Additionally, provide the parameters as a JSON body.

```python
async with aiohttp.ClientSession() as session:
    # Create stream
    async with session.post(
        "http://localhost:1234/v1/streams/processor", json={"multiplier": 3.0}
    ) as response:
        stream_data = await response.json()
        stream_id = stream_data["instance_id"]
        print(f"Stream created with ID: {stream_id}")
```

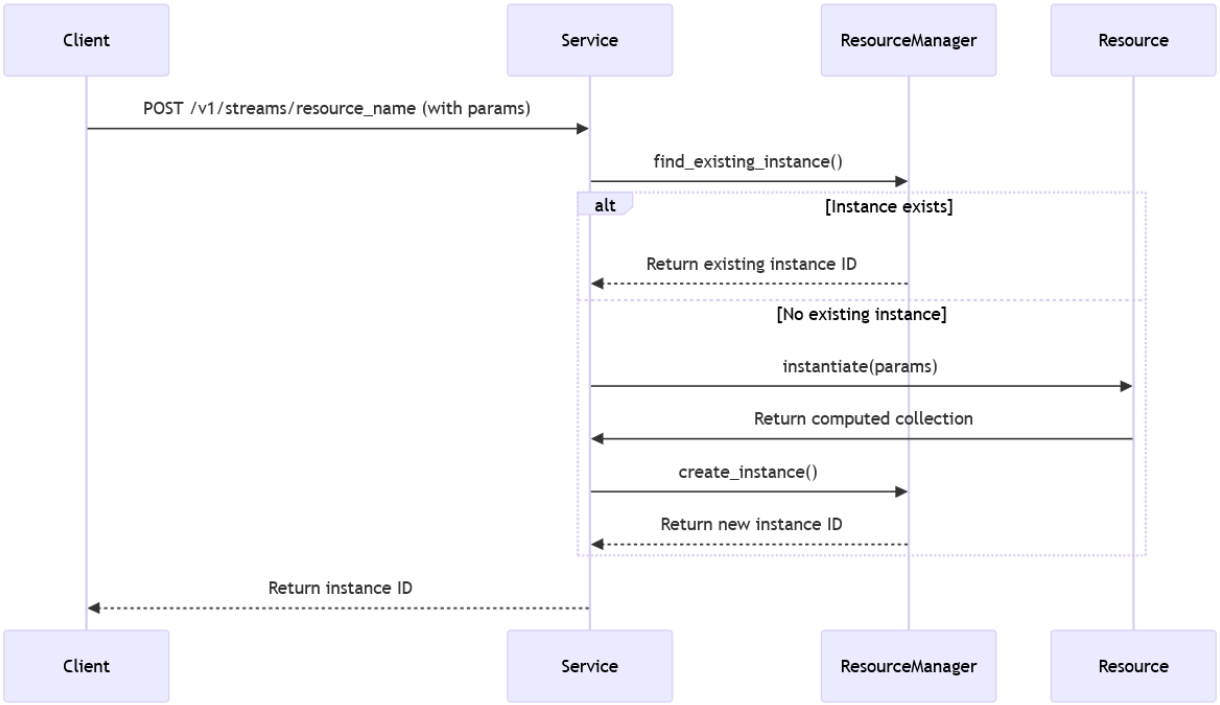

Figure 15: Resource Instantiation Flow

When a client requests a resource, the framework:

- Validates the parameters
- Checks if an instance with these parameters already exists
- Creates a new instance if needed, else it finds a shared instance with the same parameters
- Returns an instance ID to the client

2. Subscribing to Resources

Clients can then subscribe to resources using Server-Sent Events by calling the `/v1/streams/{instance_id}` endpoint with the provided instance ID.

```python
async with aiohttp.ClientSession() as session:
    # Connect to stream
    async with session.get(
        f"http://localhost:1234/v1/streams/{stream_id}",
        headers={
            "Accept": "text/event-stream",
            "Cache-Control": "no-cache",
            "Connection": "keep-alive",
        },
    ) as response:
        async for line in response.content:
            if line.startswith(b"data: "):
                data = json.loads(line[6:].decode("utf-8"))
                print(f"Received: {data}")
```

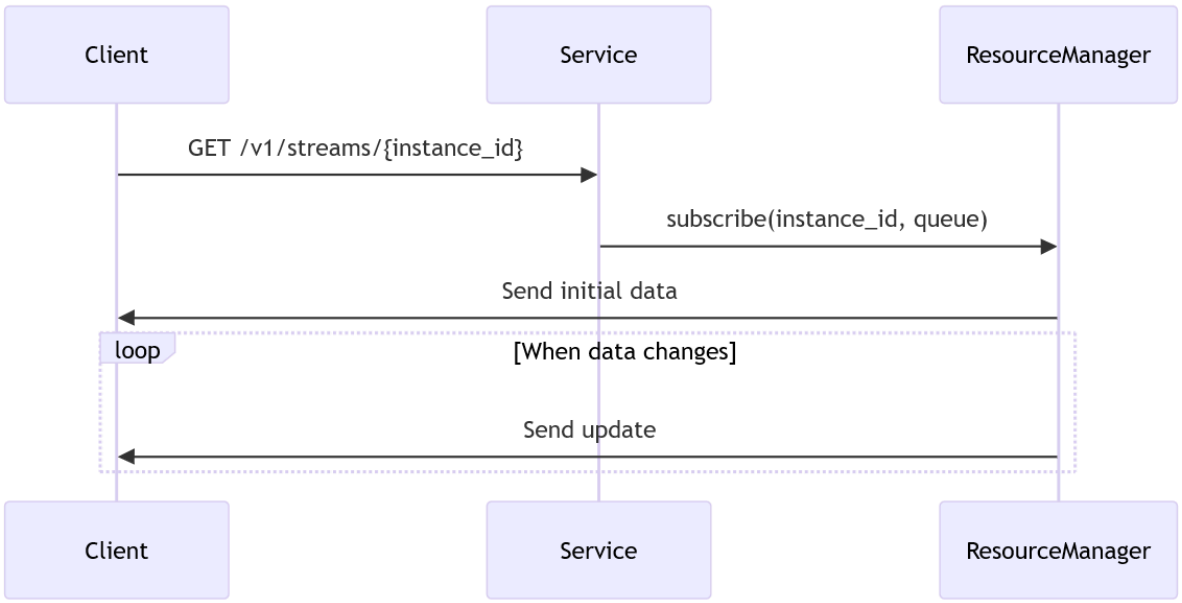

Figure 16: Resource Subscription Flow

6.6 Service - Classic API

The Service class is responsible for hosting resources and exposing them to clients through HTTP endpoints. It sets up a web server, manages resource instances, and handles client connections.

6.6.1 Creating a Service

```python
from reactive.classic.service import Service

# Create a service that listens on localhost:1234
service = Service("my_service", host="localhost", port=1234)
```

6.6.2 Adding Resources

Resources must be registered with the service to be accessible to clients:

```python
# Create collections
users_collection = ComputedCollection[str, dict]("users", service.compute_graph)
activities_collection = ComputedCollection[str, list]("activities", service.compute_graph)

# Create resources
user_resource = UserResource(users_collection, service.compute_graph)
activity_resource = ActivityResource(activities_collection, service.compute_graph)

# Register resources with the service
service.add_resource("users", user_resource)
service.add_resource("activities", activity_resource)
```

6.6.3 HTTP Endpoints

The service exposes the following HTTP endpoints:

1. **Create Stream**: `POST /v1/streams/<resource_name>`
   - Creates a new resource instance with specified parameters
   - Returns an instance ID
2. **Subscribe to Stream**: `GET /v1/streams/<instance_id>`
   - Establishes a Server-Sent Events connection to receive updates
   - Returns initial data and subsequent updates

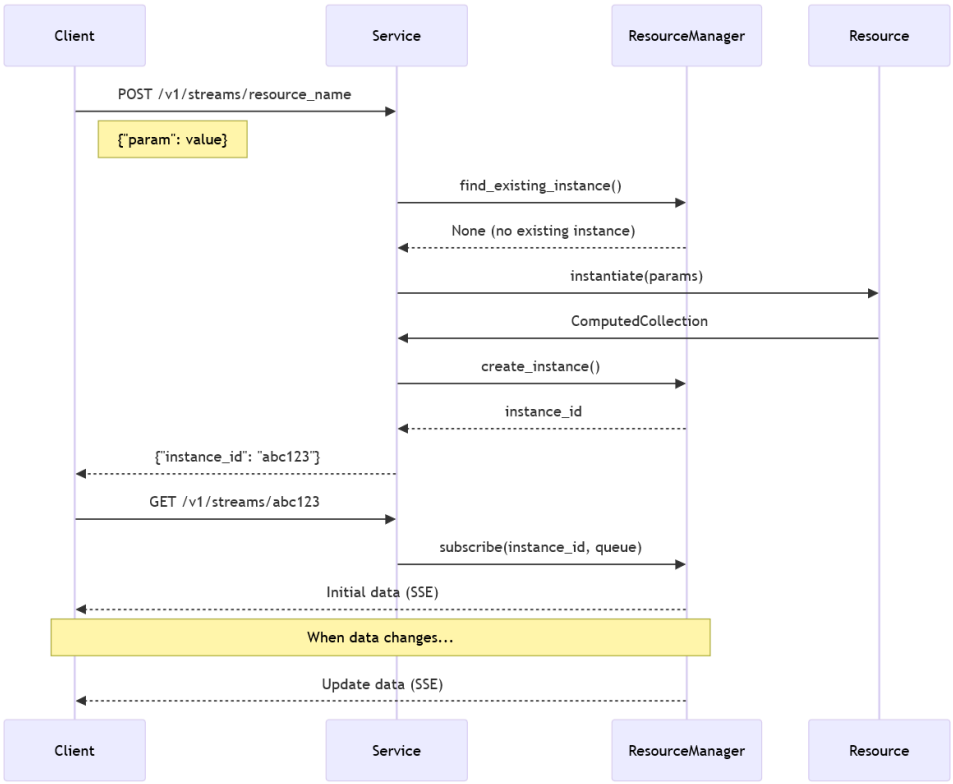

Figure 17: Full Service Flow

6.6.4 Server-Sent Events Format

The server sends updates using the Server-Sent Events format:

```
event: init
data: [[key1, value1], [key2, value2], ...]

event: update
data: [[key1, [new_value1]], [key2, [new_value2]], ...]
```

6.6.5 Running the Service

The service is run asynchronously:

```python
async def main():
    # Create service
    service = Service("my_service", port=1234)

    # Add resources
    # ...

    # Start the service
    await service.start()

if __name__ == "__main__":
    asyncio.run(main())
```

7 Reactive Framework - Metaprogramming API

The Metaprogramming API provides a more concise and declarative approach to creating reactive applications using Python's metaprogramming features.

7.1 Basic Concepts

The Metaprogramming API leverages Python's decorators, metaclasses, and AST analysis to provide a more streamlined interface to the reactive framework. It offers the same functionality as the Classic API but with less boilerplate code.

Key components include:

- Decorator-based mappers (`@one_to_one`, `@many_to_one`)
- Decorator-based resource definition with `@resource` decorator
- Simplified service configuration
- Automatic dependency detection

7.2 Creating a Simple Application

> **Note**
> You can run this example locally. It is available under the `examples/basic`[13] directory.

Figure 18: Simple Example

Let's create the same simple example application we did in the Classic API section. The creation of the `raw_data` collection remains unchanged.

```
# Create a service
service = Service("data_processor", port=8080)
# Create a collection for raw data
raw_data = ComputedCollection("raw_data", service.compute_graph)
```

Next we define the mapper `MultiValueMapper` that multiplies the sensor value by a multiple and returns a formatted message.

```
@one_to_one
def multiply_value(value, multiplier: float):
    if value is None:
        return None
    return {"value": value.value * multiplier, "timestamp": value.timestamp}
```

[13] https://github.com/CFSY/meta-reactive

Lastly we define a resource `DataProcessorResource` that allows a client to stream data from a sensor.

```
# Define a resource with decorator
@resource
def processor(multiplier: float):
    # The data collection is automatically detected as a dependency
    multiplied_data = map_collection(raw_data, multiply_value, multiplier)
    return multiplied_data
```

7.2.1 Differences

1. **Mapper**: Instead of creating a class as in the classic API, we use the `@one_to_one` decorator to create a mapper function that transforms each data point by applying a multiplier.
2. **Resource**: We use the `@resource` decorator to define our processor resource. The parameters of the function become the parameters of the resource.

7.2.2 Connecting a Client

The client code is identical to what you'd use with the Classic API

7.2.3 Key Features of the Metaprogramming API

1. **Decorator-Based Design**: Transforms functions into framework components
2. **Automatic Dependency Detection**: Framework analyzes code to detect dependencies
3. **Reduced Boilerplate**: Eliminates the need for explicit class definitions
4. **Global Resource Registry**: Resources are automatically registered with services

This is just a basic example. In the next sections, we'll explore each component in more detail and cover more advanced usage patterns.

7.3 Collections - Metaprogramming API

The Metaprogramming API uses the same collection classes as the Classic API but provides more convenient ways to work with them. This section covers how to create and manipulate collections using the Metaprogramming API.

7.3.1 Creating Collections

Creating collections works the same way as in the Classic API:

```
# Create a service
service = Service("my_service")

# Create a computed collection
users = ComputedCollection[str, dict]("users", service.compute_graph)
```

7.3.2 Basic Operations

Basic operations on collections are also the same:

```
# Set values
users.set("user1", {"name": "Alice", "email": "alice@example.com"})
users.set("user2", {"name": "Bob", "email": "bob@example.com"})

# Get values
user = users.get("user1")  # Returns the user dict or None

# Delete values
users.delete("user2")

# Iterate over items
for user_id, user_data in users.iter_items():
    print(f"{user_id}: {user_data['name']}")
```

7.3.3 Transforming Collections

The key difference with the Metaprogramming API is how you define and apply transformations. In the metaprogramming API, mapping collections is done using mapper functions decorated with `@one_to_one` or `@many_to_one`, along with the `map_collection` function.

```
# Define a transformation using a decorator
@one_to_one
def extract_name(user_data):
    return user_data.get("name")
# Apply the transformation to create a new
collection
names = map_collection(users, extract_name)
```

7.3.4 Dependency Detection

Dependencies between collections are automatically detected when they're used in mapper functions or resource definitions, reducing the need for explicit declarations.

```
# This collection will automatically be detected as a dependency
reference_data = ComputedCollection[str, dict]("reference", service.compute_graph)

@one_to_one
def enrich_user(user_data):
    # The framework detects this reference to reference_data
    ref = reference_data.get(user_data.get("id"))
    if ref:
        result = dict(user_data)
        result["reference"] = ref
        return result
    return user_data

# Apply the transformation - reference_data is automatically included as a dependency
enriched_users = map_collection(users, enrich_user)
```

In the above example, the framework automatically detects that `enrich_user` depends on `reference_data` and establishes the appropriate dependency in the compute graph.

7.4 Mappers - Metaprogramming API

Mappers are functions that transform data from one collection to another. In the metaprogramming API, mappers are defined using decorators, which dramatically simplifies their implementation compared to the class-based approach in the classic API.

7.4.1 Types of Mappers

The framework provides two main types of mappers:

1. **One-to-One Mappers**: Transform a single value into another value with the same key
2. **Many-to-One Mappers**: Transform a list of values with the same key into a single value

7.4.1.1 Defining One-to-One Mappers

```python
@one_to_one
def temperature_to_fahrenheit(celsius_data):
    """Convert temperature from Celsius to Fahrenheit"""
    if celsius_data is None:
        return None

    celsius = celsius_data["temperature"]
    fahrenheit = celsius * 9/5 + 32

    return {
        "temperature": fahrenheit,
        "unit": "F",
        "timestamp": celsius_data["timestamp"]
    }
```

7.4.1.2 Defining Many-to-One Mappers

```python
@many_to_one
def calculate_average(temperature_readings):
    """Calculate average temperature from a list of readings"""
    if not temperature_readings:
        return None

    avg_temp = sum(reading["temperature"] for reading in temperature_readings) /
len(temperature_readings)

    return {
        "average_temperature": avg_temp,
        "sample_count": len(temperature_readings),
        "timestamp": temperature_readings[-1]["timestamp"]
    }
```

7.4.1.3 Passing Parameters to Mappers

You can pass additional parameters to mappers when mapping collections:

```
@one_to_one
def scale_value(value, factor):
    """Scale a value by a factor"""
    if value is None:
        return None
    return value * factor

# Use the mapper with a specific factor
scaled_collection = map_collection(original_collection, scale_value, 2.5)
```

7.4.2 Automatic Dependency Detection

One of the most powerful features of the metaprogramming API is its ability to automatically detect dependencies between collections. When a mapper function references a collection, the framework will detect this and establish the appropriate dependency:

```
reference_collection = ComputedCollection("reference", compute_graph)

@one_to_one
def enrich_data(value):
    # The reference_collection is automatically detected as a dependency
    reference = reference_collection.get("config")
    return {
        "original_value": value,
        "enriched_value": value * reference["factor"]
    }

enriched_collection = map_collection(data_collection, enrich_data)
```

The metaprogramming API's mapper approach significantly reduces boilerplate code compared to the classic API, while maintaining all the functionality and adding automatic dependency detection.

7.5 Resources - Metaprogramming API

In the metaprogramming API, resources are defined using the `@resource` decorator, which simplifies their creation compared to the class-based approach in the classic API.

7.5.1 Defining Resources

Use the `@resource` decorator to define a function that sets up a resource:

```python
@resource
def processor(multiplier: float):
    multiplied_data = map_collection(raw_data, multiply_value, multiplier)
    return multiplied_data
```

7.5.2 Resource Parameters

Unlike the classic API, there is no need to define a parameter model. The parameters of the resource function become the parameters of the resource. The framework automatically creates a Pydantic model for validating these parameters:

```python
# The resource parameters are derived from the function parameters
@resource
def temperature_monitor(min_threshold: float = 0.0, max_threshold: float = 100.0):
    # Resource implementation
    return monitored_data
```

When a client creates a stream for this resource, they'll need to provide these parameters:

```
POST /v1/streams/temperature_monitor
{
    "min_threshold": 15.0,
    "max_threshold": 30.0
}
```

7.5.2.1 Custom Parameter Models

You can also provide a custom Pydantic model for resource parameters:

```python
from pydantic import BaseModel

class AlertConfig(BaseModel):
    threshold: float
    notify_admin: bool = False
    alert_level: str = "warning"

@resource(param_model=AlertConfig)
def alert_monitor(threshold: float, notify_admin: bool, alert_level: str):
    # Resource implementation using these parameters
    return alerts_collection
```

7.5.2.2 Resource Registration

Resources are automatically registered when they're defined with the `@resource` decorator. Unlike the classic API, you don't need to explicitly add them to the service - the framework takes care of this when the service starts.

The metaprogramming API maintains the same resource access pattern as the classic API but simplifies the resource definition process.

7.6 Service - Metaprogramming API

> **Note**
> In the metaprogramming API, the Service class is a wrapper around the classic Service implementation with automatic resource registration. The API details are identical to the Classic API.

7.6.1 Resource Registration

The main difference from the Classic API is resource registration. Unlike the classic API, you don't need to explicitly add resources to the service. Resources defined with the `@resource` decorator are automatically registered when the service starts:

```python
from reactive.meta import resource, Service

@resource
def my_resource(param1: str, param2: int = 0):
    # Resource implementation
    return result_collection

# Create and start the service
service = Service("my_app")
# No need to call service.add_resource - it happens automatically!
```

8 API Comparison: Classic vs Metaprogramming

The reactive framework provides two API styles that offer the same functionality but with different syntax and levels of abstraction. Let's compare them using the temperature monitor example.

> **Note**
> You can run this example locally. It is available under the `examples/temp-monitor`[14] directory.

8.1 Temperature Monitor Example

The Temperature Monitor system collects readings from multiple temperature sensors placed in different locations (office, server room, warehouse). The system needs to:

1. Compute average temperatures for each sensor
2. Compare temperatures against acceptable ranges for each location
3. Generate alerts when temperatures exceed thresholds
4. Stream alerts to clients in real-time

Here's a diagram showing the architecture of the Temperature Monitor application:

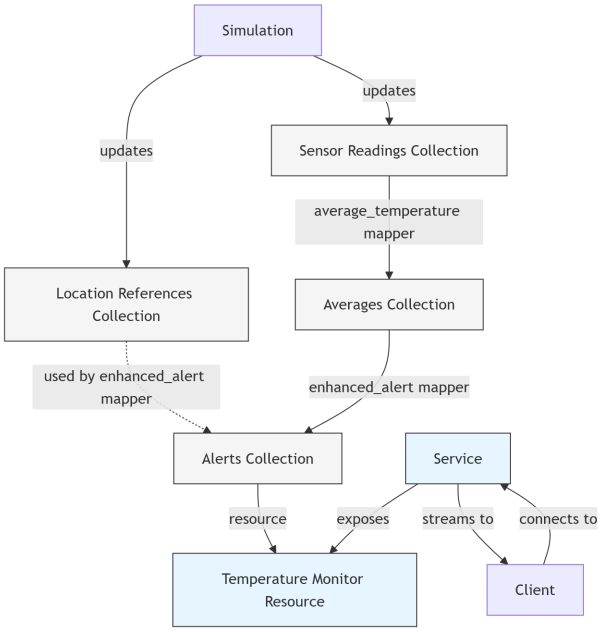

Figure 19: Temperature Monitor Example

1. **Sensor Readings Collection**: Contains lists of temperature readings for each sensor.
2. **Average Temperature Mapper**: Computes the average temperature for each sensor from its readings.
3. **Enhanced Alert Mapper**: Generates alerts by comparing temperatures against reference data.
4. **Location References Collection**: Contains acceptable temperature ranges for each location.
5. **Alerts Collection**: Contains generated alerts for each sensor.
6. **Temperature Monitor Resource**: Exposes the alerts to clients via the service.

[14] https://github.com/CFSY/meta-reactive

The primary advantage of the Metaprogramming API lies in its ability to reduce boilerplate and provide a more declarative syntax, achieving the same functionality as the Classic API with significantly less code. Let's compare key aspects side-by-side, using examples derived from the Temperature Monitor.

8.2 Mapper Definition

8.2.1 Classic API:

Requires defining a class inheriting from a base mapper type.

```
class EnhancedAlertMapper(OneToOneMapper[str, Tuple[float, str], Dict[str, str]]):
    def __init__(
        self, location_references: ComputedCollection, global_threshold: float
    ):
        self.location_references = location_references
        self.global_threshold = global_threshold

    def map_value(self, value: Tuple[float, str]) -> Dict[str, str]:
        avg_temp, location = value

        # Get reference data for this location
        location_info = self.location_references.get(location)

        # Use common evaluation logic
        return evaluate_temperature_alert(
            avg_temp, location, location_info, self.global_threshold
        )
```

8.2.2 Metaprogramming API:

Uses a simple decorator (`@one_to_one`) on a standard function.

```
@one_to_one
def enhanced_alert(value: Tuple[float, str], global_threshold: float) -> Dict[str, str]:
    avg_temp, location = value

    # Get reference data for this location
    location_info = location_ref_collection.get(location)

    # Use common evaluation logic
    return evaluate_temperature_alert(
        avg_temp, location, location_info, global_threshold
        )
```

The Meta API eliminates class definition boilerplate, focusing purely on the transformation logic. Type hints remain for clarity and potential future tooling.

8.3 Resource Definition and Parameter Handling

8.3.1 Classic API:

Requires a `ResourceParams` subclass and a `Resource` subclass, with explicit parameter passing.

```python
class MonitorParams(ResourceParams):
    threshold: float

class TemperatureMonitorResource(Resource[str, dict]):
    def __init__(self, readings, locations, graph):
        super().__init__(MonitorParams, graph)
        self.readings = readings
        self.locations = locations

    def setup_resource_collection(self, params: MonitorParams):
        averages = self.readings.map(AverageTemperatureMapper)
        alerts = averages.map(
            EnhancedAlertMapper, # Mapper class
            self.locations,     # Explicit dependency
            params.threshold   # Explicit parameter
        )
        return alerts
```

8.3.2 Metaprogramming API:

Uses the `@resource` decorator. Function parameters **become** resource parameters, automatically generating a Pydantic model internally. Dependencies are often detected automatically.

```python
# location_ref_collection defined globally/accessibly
# average_temperature, enhanced_alert are decorated mapper functions

@resource # No separate Param class needed
def temperature_monitor(threshold: float): # Function param is resource param
    averages = map_collection(readings_collection, average_temperature)
    # `enhanced_alert` mapper function used directly
    # `threshold` parameter passed directly
    # `location_ref_collection` dependency detected automatically
    alerts = map_collection(averages, enhanced_alert, threshold)
    return alerts
```

The Meta API drastically reduces code. It merges parameter definition and resource logic, infers parameter models, and automatically handles dependencies (like `location_ref_collection` used within `enhanced_alert`), making the code more declarative.

8.4 Resource Registration

8.4.1 Classic API:

Requires explicit instantiation and registration with the service.

```
service = Service("temp_monitor", port=1234)
readings = ComputedCollection(...)
locations = ComputedCollection(...)

# Explicit instantiation and registration
monitor = TemperatureMonitorResource(readings, locations, service.compute_graph)
service.add_resource("temperature_monitor", monitor)
```

8.4.2 Metaprogramming API:

Registration is automatic for functions decorated with `@resource`.

```
service = Service("temp_monitor", port=1234)
readings = ComputedCollection(...)
locations = ComputedCollection(...)

@resource
def temperature_monitor(threshold: float):
    # ... (resource logic as above)
    pass # No explicit registration needed

# When service.start() is called, 'temperature_monitor' is automatically found
# and registered because of the @resource decorator.
```

Eliminates manual registration steps, reducing potential errors and simplifying setup. The framework discovers resources automatically.

8.5 Summary of Differences:

Feature	Classic API	Metaprogramming API	Benefit of Meta API
Mapper Def.	Inherited Class	Decorated Function	Less boilerplate, focus on logic
Resource Def.	Inherited Class + Param Class	Decorated Function	Concise, integrated parameter definition
Registration	Explicit `service.add_resource()`	Automatic via `@resource`	Simpler setup, less error-prone
Dependencies	Explicit (constructor/map args)	Automatic Detection	Reduced boilerplate, more declarative

The comparison shows that the Metaprogramming API provides a more streamlined and developer-friendly interface by leveraging metaprogramming to handle boilerplate and automate common tasks like registration and dependency management.

9 Metaprogramming Internals of the Reactive Framework

The Metaprogramming API achieves its concise syntax and automatic behaviors by building upon the Classic API and employing several metaprogramming techniques. It acts as a translation layer, converting the user-friendly meta-constructs into their classic counterparts behind the scenes.

9.1 Decorator-Based Components (`@resource`, `@one_to_one`, `@many_to_one`)

Decorators are the primary mechanism for defining components in the Meta API.

9.1.1 `@resource`

- **Function:** Transforms a user-defined function into a `Resource` instance.
- **Mechanism:**
  1. Captures the decorated function (`func`).
  2. Determines the resource name (either provided or from `func.__name__`).
  3. Analyzes the function's signature and type hints using `inspect.signature` and `typing.get_type_hints` to automatically generate a Pydantic `param_model` if one isn't explicitly provided.
  4. Creates an instance of the Meta API's `Resource` class.
  5. Registers the captured function (`func`) as the `_setup_method` within the `Resource` instance. This method will be called later by the underlying classic resource's `setup_resource_collection`.
  6. Adds the `Resource` instance to the `global_resource_registry`. The `Service` class later iterates over this registry during startup to automatically add resources.
- **Effect:** Allows users to define a resource and its parameter schema within a single function definition, eliminating the need for separate `Resource` and `ResourceParams` classes and explicit registration.

9.1.2 `@one_to_one` / `@many_to_one`

- **Function:** Transforms a user function into a `MapperWrapper` instance.
- **Mechanism:**
  1. Captures the decorated function (`func`) and the specified `MapperType` (OneToOne or ManyToOne).
  2. Creates a `MapperWrapper` instance, storing the function and type.
  3. Crucially, the `MapperWrapper.__init__` calls `_detect_dependencies` which uses the `FrameworkDetector` (see below) to find any `ComputedCollection` instances referenced within the mapper function's code. These detected dependencies are stored.
- **Effect:** Allows users to define mapping logic in simple functions. The wrapper holds the logic and detected dependencies, ready to be used by `map_collection`.

9.2 The `map_collection` Bridge Function

Defined in `reactive/meta/mapper.py`, this function acts as the bridge between the Meta API's mappers and the Classic API's collection mapping mechanism.

Function: Applies a Meta API mapper (a `MapperWrapper`) to a `ComputedCollection`.

Mechanism:

1. Takes the `collection`, the `mapper` (which must be a `MapperWrapper` instance created by the decorators), and any additional `*args`, `**kwargs` to be passed to the user's mapping function.
2. Checks the `mapper_wrapper.mapper_type`.
3. Selects the appropriate **internal Classic Mapper implementation** (`_OneToOneMapperImpl` or `_ManyToOneMapperImpl`). These internal classes are simple wrappers that take the user's function and execute it within the `map_value` or `map_values` method required by the Classic API.
4. Calls the underlying `collection.map` method (from `ComputedCollection`), passing the selected **internal Classic Mapper implementation class** and the user's original mapping function (`mapper_wrapper.mapper_func`) along with the `*args` and `**kwargs`.

Effect: Translates the user-friendly Meta API `map_collection(collection, mapper_func, ...)` call into the Classic API's `collection.map(ClassicMapperImpl, mapper_func, ...)` structure, ensuring compatibility with the core computation graph logic while hiding the Classic Mapper class definitions from the user.

9.3 The `FrameworkDetector` and Automatic Dependency Detection

The `FrameworkDetector` (defined in `reactive/meta/detector.py`) is perhaps the most sophisticated metaprogramming component, enabling automatic dependency detection within user code.

Its primary goal is to analyze the source code of functions (like those decorated with `@one_to_one` or `@resource`) and identify if they reference global or non-local variables that are instances of framework components, specifically `ComputedCollection`. This allows the framework to automatically establish necessary links in the `ComputeGraph` without requiring explicit declarations from the user.

9.3.1 Marking and Identification:

The detector uses a unique attribute name (e.g., `__reactive_meta_component__`) to "mark" functions or classes that are part of the framework or created by its decorators. This marking is done **internally** by helper functions/classes provided by the detector:

- `detector.get_function_decorator()`: Returns a decorator used on functions like `@resource`, `@one_to_one`, etc. This decorator wraps the function and sets the marker attribute on the wrapper.
- `detector.get_metaclass()`: Returns a metaclass used on internal framework classes (like `Service`, `Resource`). This metaclass sets the marker attribute on the class itself and analyzes its methods upon class creation.
- The `is_framework_component()` method simply checks for the presence of this marker attribute.

9.3.2 Detection via AST Analysis:

When `detect_framework_usage` is called (e.g., inside `MapperWrapper.__init__`), it uses the `CodeAnalyzer` sub-component of FrameworkDetector. The following occurs internally:

- **Source Code Retrieval:** `inspect.getsource()` retrieves the source code of the target function/method.
- **Parsing:** `libcst.parse_module()` converts the source code string into an Abstract Syntax Tree (AST). Libcst is chosen for its robustness and ability to preserve formatting information.
- **AST Traversal:** A custom visitor class, `FrameworkReferenceCollector`, walks the AST.
  - It overrides methods like `visit_Call` (for function calls) and `visit_Attribute` (for attribute access).
  - Inside these visitors, it identifies names (e.g., `my_collection` in `my_collection.get()`).
- **Name Resolution:** The crucial step is linking syntax nodes (like `cst.Name`) back to actual Python objects.
  - The visitor uses the `global_namespace` (obtained via `func.__globals__` and `inspect.getmodule(func).__dict__`) and `local_namespace` (if applicable, like closures) passed to it.
  - The `_resolve_name` helper attempts to find the object corresponding to the name in these namespaces.
- **Framework Check:** If a name resolves to an object, `detector.is_framework_component(obj)` is called. If it **is** a marked framework component, its name (e.g., `"my_collection"`, `"my_collection.get()"`) is added to the set of references.
- **Indirect Calls:** The analyzer also uses `FunctionCallExtractor` to find functions called **within** the analyzed function. It then recursively analyzes these called functions (using cached results via `get_framework_references` or calling `detect_framework_usage` again) to find indirect dependencies.
- **Result:** Returns a set of strings representing the names of framework components referenced directly or indirectly within the analyzed code.

9.3.3 Integration

The detector is instantiated once. In our framework, this is done in `common.py`[15]

```
detector = FrameworkDetector("reactive_meta")
framework_function = detector.get_function_decorator()
FrameworkClass = detector.get_metaclass()
```

The marking decorators/metaclasses (`framework_function`, `FrameworkClass`) are exported from `common.py` and used internally within the `reactive.meta` package on components like `@resource`, `@one_to_one`, `Service`, etc. The detector decorators/metaclasses were implemented to be "invisible", making their internal usage extremely easy. Notice how we are decorating the `@one_to_one` decorator.

```
class Resource(Generic[K, V], metaclass=FrameworkClass):
...

@framework_function
def one_to_one(func):
...
```

This seamless integration means dependency detection happens automatically when a user decorates a function, without them needing to interact with the detector directly.

[15] https://github.com/CFSY/meta-reactive/blob/main/src/reactive/meta/common.py

10 Tradeoffs of Metaprogramming in API Design

While the Metaprogramming API in the reactive framework demonstrates significant improvements in developer experience, adopting metaprogramming techniques for API design involves inherent tradeoffs that must be carefully considered.

10.1 Complexity vs. Conciseness (Developer vs. Maintainer)

- **Pro (User)**: Metaprogramming hides underlying complexity, leading to cleaner, more concise user code (as seen in the API comparison). Users focus on **what** they want to achieve (e.g., define a resource) rather than **how** the framework implements it (class inheritance, registration).
- **Con (Maintainer)**: The framework's internal code becomes significantly more complex. Understanding decorators that modify function behavior, metaclasses altering class creation, and AST analysis requires deeper Python knowledge. Debugging issues might involve tracing through generated wrappers or analyzing the AST logic, which is harder than debugging straightforward OOP code. For example, debugging the `FrameworkDetector` requires understanding `libcst` and namespace resolution.

10.2 Explicitness vs. "Magic"

- **Pro (User)**: Automatic behaviors, like dependency detection via the `FrameworkDetector` or automatic resource registration via `@resource`, feel magical and reduce boilerplate. The user doesn't need to explicitly list every `ComputedCollection` used in a mapper.
- **Con (User/Maintainer)**: This "magic" can obscure cause-and-effect. If automatic dependency detection fails or behaves unexpectedly, it can be difficult to diagnose **why** a dependency wasn't registered correctly without understanding the AST analysis internals. The Classic API, being explicit, makes dependencies obvious, aiding traceability.

10.3 Tooling and Static Analysis

- **Con:** Metaprogramming constructs can sometimes challenge static analysis tools.
  - **Type Hinting:** While decorators (`@wraps`) and careful implementation can preserve type hints, complex transformations might confuse type checkers like MyPy, requiring `# type: ignore` annotations or more complex stub files. The automatic Pydantic model generation in `@resource` works well but isn't immediately visible from just reading the function signature without running the code or inspecting the generated resource.
  - **IDE Support:** Features like "go to definition" or autocompletion might sometimes lead to the decorator implementation or wrapper code instead of the user's original function, slightly hindering navigation.
- **Pro:** When done well (like using `functools.wraps`), basic introspection (`help()`, `__name__`, `__doc__`) can be preserved, mitigating some tooling issues.

10.4 Development Approach and Maintainability

As identified during the implementation of the framework, there are different ways to structure a project offering both classic and metaprogramming APIs:

1. **Dual Interfaces:** Implement a core logic engine and have both Classic and Meta APIs interface with it directly.

- **Challenge:** Designing a core sufficiently abstract and flexible to cleanly support two potentially divergent API philosophies from the outset is extremely difficult. It risks over-engineering the core and makes maintenance complex as changes might need synchronization across three layers (Core, Classic, Meta). **This approach was deemed impractical in the early stages of development.**

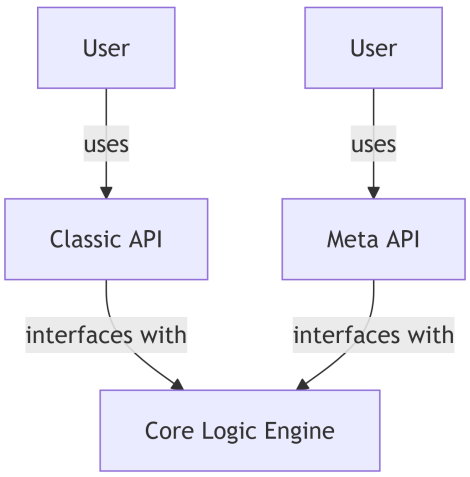

Figure 20: Dual Interfaces

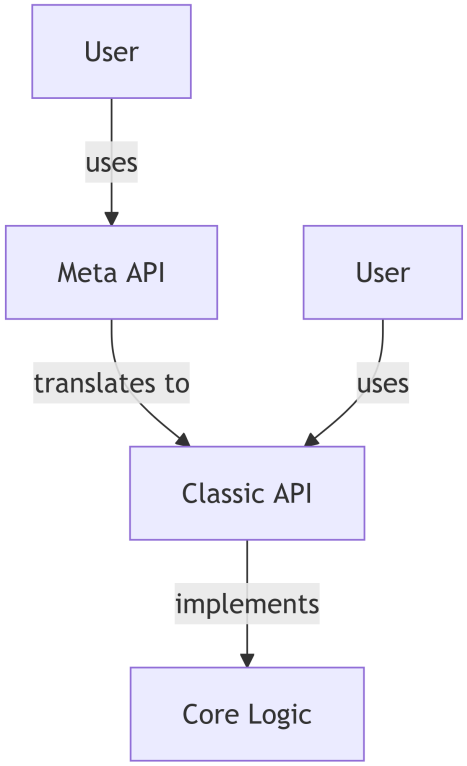

Figure 21: Meta API as a Layer on Classic API

2. **Meta API as a Layer on Classic API:** Implement the full core logic and the Classic API first. Then, build the Metaprogramming API as a translation layer on top of the Classic API.

Pros:

- **Iterative Development:** Allows focusing on core functionality first via the Classic API. The Meta API can be added later.
- **Stability:** The Classic API provides a stable, explicit base.
- **Compatibility:** Guarantees that both APIs ultimately use the same core logic, preventing divergence.
- **Migration Path:** Users could potentially mix and match or migrate gradually from Classic to Meta components if needed.

Cons:

- **Coupling:** The Meta API becomes tightly coupled to the Classic API's implementation details. Changes in the Classic API might require corresponding changes in the Meta layer.
- **Overhead:** There's a small, likely negligible, performance overhead due to the extra layer of indirection (decorators, wrappers calling classic methods).

11 Conclusion

This project explored the landscape of Python metaprogramming, moving from theoretical analysis to practical application. The initial phase involved dissecting techniques used in real-world frameworks like `stateflow` and `SGLang`, identifying patterns such as decorators for transformation, metaclasses for instance control, AST manipulation for analysis, and operator overloading for intuitive syntax.

The second phase culminated in the development of the "Reactive Framework," a system designed specifically to provide a concrete comparison between traditional API design and one enhanced by metaprogramming. By implementing both a Classic API (emphasizing explicit, object-oriented patterns) and a Metaprogramming API (leveraging decorators, automatic dependency detection via AST analysis, and implicit registration), the project provided tangible evidence of the benefits metaprogramming offers for developer experience. The Meta API demonstrably reduced boilerplate code, promoted a more declarative style, and automated common framework interactions, simplifying the process of defining reactive components like mappers and resources.

The implementation also highlighted the inherent tradeoffs. While the Meta API offers a superior user interface, its internal complexity is higher, relying on advanced techniques like AST analysis via `libcst` within the `FrameworkDetector`. The decision to build the Meta API as a layer upon the Classic API proved a pragmatic development strategy, ensuring core logic consistency and allowing for iterative enhancement, though it introduced coupling between the API layers.

Ultimately, this project confirms that Python's metaprogramming capabilities are powerful tools for framework authors. When applied thoughtfully, they can significantly enhance API design, leading to more expressive, concise, and developer-friendly interfaces. However, their use necessitates careful consideration of complexity, maintainability, tooling impact, and the specific needs of the target audience. The Reactive Framework serves as a practical case study illustrating both the potential and the associated challenges of leveraging metaprogramming in Python

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