Python API

Operations

`docetl.schemas.MapOp = map.MapOperation.schema` `module-attribute`

`docetl.schemas.ResolveOp = resolve.ResolveOperation.schema` `module-attribute`

`docetl.schemas.ReduceOp = reduce.ReduceOperation.schema` `module-attribute`

`docetl.schemas.ParallelMapOp = map.ParallelMapOperation.schema` `module-attribute`

`docetl.schemas.FilterOp = filter.FilterOperation.schema` `module-attribute`

`docetl.schemas.EquijoinOp = equijoin.EquijoinOperation.schema` `module-attribute`

`docetl.schemas.SplitOp = split.SplitOperation.schema` `module-attribute`

`docetl.schemas.GatherOp = gather.GatherOperation.schema` `module-attribute`

`docetl.schemas.UnnestOp = unnest.UnnestOperation.schema` `module-attribute`

`docetl.schemas.SampleOp = sample.SampleOperation.schema` `module-attribute` `docetl.schemas.ClusterOp = cluster.ClusterOperation.schema` `module-attribute`

Dataset and Pipeline

`docetl.schemas.Dataset = dataset.Dataset.schema` `module-attribute`

`docetl.schemas.ParsingTool`

Bases: `BaseModel`

Represents a parsing tool used for custom data parsing in the pipeline.

Attributes:

Name	Type	Description
`name`	`str`	The name of the parsing tool. This should be unique within the pipeline configuration.
`function_code`	`str`	The Python code defining the parsing function. This code will be executed to parse the input data according to the specified logic. It should return a list of strings, where each string is its own document.

Example

```
parsing_tools:
  - name: ocr_parser
    function_code: |
      import pytesseract
      from pdf2image import convert_from_path
      def ocr_parser(filename: str) -> list[str]:
          images = convert_from_path(filename)
          text = ""
          for image in images:
              text += pytesseract.image_to_string(image)
          return [text]
```

Source code in `docetl/base_schemas.py`

~~~python
class ParsingTool(BaseModel):
    """
    Represents a parsing tool used for custom data parsing in the
pipeline.

    Attributes:
        name (str): The name of the parsing tool. This should be unique
within the pipeline configuration.
        function_code (str): The Python code defining the parsing
function. This code will be executed
                             to parse the input data according to the
specified logic. It should return a list of strings, where each string is
its own document.

    Example:
        ```yaml
        parsing_tools:
          - name: ocr_parser
            function_code: |
              import pytesseract
              from pdf2image import convert_from_path
              def ocr_parser(filename: str) -> list[str]:
                  images = convert_from_path(filename)
                  text = ""
                  for image in images:
                      text += pytesseract.image_to_string(image)
                  return [text]
        ~~~
    """

    name: str
    function_code: str
```

`docetl.schemas.PipelineStep`

Bases: `BaseModel`

Represents a step in the pipeline.

Attributes:

Name	Type	Description
`name`	`str`	The name of the step.
```

| Name | Type | Description |
|---|---|---|
| `operations` | `list[dict[str, Any] \| str]` | A list of operations to be applied in this step. Each operation can be either a string (the name of the operation) or a dictionary (for more complex configurations). |
| `input` | `str \| None` | The input for this step. It can be either the name of a dataset or the name of a previous step. If not provided, the step will use the output of the previous step as its input. |

**Example**

```python
# Simple step with a single operation
process_step = PipelineStep(
    name="process_step",
    input="my_dataset",
    operations=["process"]
)

# Step with multiple operations
summarize_step = PipelineStep(
    name="summarize_step",
    input="process_step",
    operations=["summarize"]
)

# Step with a more complex operation configuration
custom_step = PipelineStep(
    name="custom_step",
    input="previous_step",
    operations=[
        {
            "custom_operation": {
                "model": "gpt-4",
                "prompt": "Perform a custom analysis on the following text:"
            }
        }
    ]
)
```

These examples show different ways to configure pipeline steps, from simple single-operation steps to more complex configurations with custom parameters.

Source code in `docetl/base_schemas.py`

~~~python
class PipelineStep(BaseModel):
    """
    Represents a step in the pipeline.

    Attributes:
        name (str): The name of the step.
        operations (list[dict[str, Any] | str]): A list of operations to
be applied in this step.
            Each operation can be either a string (the name of the
operation) or a dictionary
            (for more complex configurations).
        input (str | None): The input for this step. It can be either the
name of a dataset
            or the name of a previous step. If not provided, the step will
use the output
            of the previous step as its input.

    Example:
        ```python
        # Simple step with a single operation
        process_step = PipelineStep(
            name="process_step",
            input="my_dataset",
            operations=["process"]
        )

        # Step with multiple operations
        summarize_step = PipelineStep(
            name="summarize_step",
            input="process_step",
            operations=["summarize"]
        )

        # Step with a more complex operation configuration
        custom_step = PipelineStep(
            name="custom_step",
            input="previous_step",
            operations=[
                {
                    "custom_operation": {
                        "model": "gpt-4",
                        "prompt": "Perform a custom analysis on the
following text:"
                    }
                }
            ]
        )
        ~~~

    These examples show different ways to configure pipeline steps, from
simple
    single-operation steps to more complex configurations with custom
parameters.
    """

    name: str
```
```

```
        operations: list[dict[str, Any] | str]
        input: str | None = None
```

`docetl.schemas.PipelineOutput`

Bases: `BaseModel`

Represents the output configuration for a pipeline.

Attributes:

Name	Type	Description
`type`	`str`	The type of output. This could be 'file', 'database', etc.
`path`	`str`	The path where the output will be stored. This could be a file path, database connection string, etc., depending on the type.
`intermediate_dir`	`str | None`	The directory to store intermediate results, if applicable. Defaults to None.

Example

```
output = PipelineOutput(
    type="file",
    path="/path/to/output.json",
    intermediate_dir="/path/to/intermediate/results"
)
```

Source code in `docetl/base_schemas.py`

~~~python
class PipelineOutput(BaseModel):
    """
    Represents the output configuration for a pipeline.

    Attributes:
        type (str): The type of output. This could be 'file', 'database',
etc.
        path (str): The path where the output will be stored. This could
be a file path,
                    database connection string, etc., depending on the
type.
        intermediate_dir (str | None): The directory to store
intermediate results,
                                       if applicable. Defaults to
None.

    Example:
        ```python
        output = PipelineOutput(
            type="file",
            path="/path/to/output.json",
            intermediate_dir="/path/to/intermediate/results"
        )
        ~~~
    """

    type: str
    path: str
    intermediate_dir: str | None = None
```

`docetl.api.Pipeline`

Represents a complete document processing pipeline.

Attributes:

Name	Type	Description
`name`	`str`	The name of the pipeline.
`datasets`	`dict[str, Dataset]`	A dictionary of datasets used in the pipeline, where keys are dataset names and values are Dataset objects.
`operations`	`list[OpType]`	A list of operations to be performed in the pipeline.
```

| Name | Type | Description |
|---|---|---|
| `steps` | `list[PipelineStep]` | A list of steps that make up the pipeline. |
| `output` | `PipelineOutput` | The output configuration for the pipeline. |
| `parsing_tools` | `list[ParsingTool]` | A list of parsing tools used in the pipeline. Defaults to an empty list. |
| `default_model` | `str \| None` | The default language model to use for operations that require one. Defaults to None. |

**Example**

```python
def custom_parser(text: str) -> list[str]:
    # this will convert the text in the column to uppercase
    # You should return a list of strings, where each string is a separate
document
    return [text.upper()]

pipeline = Pipeline(
    name="document_processing_pipeline",
    datasets={
        "input_data": Dataset(type="file", path="/path/to/input.json", parsing=
[{"name": "custom_parser", "input_key": "content", "output_key":
"uppercase_content"}]),
    },
    parsing_tools=[custom_parser],
    operations=[
        MapOp(
            name="process",
            type="map",
            prompt="Determine what type of document this is: {{
input.uppercase_content }}",
            output={"schema": {"document_type": "string"}}
        ),
        ReduceOp(
            name="summarize",
            type="reduce",
            reduce_key="document_type",
            prompt="Summarize the processed contents: {% for item in inputs %}
{{ item.uppercase_content }} {% endfor %}",
            output={"schema": {"summary": "string"}}
        )
    ],
    steps=[
        PipelineStep(name="process_step", input="input_data", operations=
["process"]),
        PipelineStep(name="summarize_step", input="process_step", operations=
["summarize"])
    ],
    output=PipelineOutput(type="file", path="/path/to/output.json"),
    default_model="gpt-4o-mini"
)
```

This example shows a complete pipeline configuration with datasets, operations, steps, and output settings.

Source code in `docetl/api.py`

~~~python
class Pipeline:
    """
    Represents a complete document processing pipeline.

    Attributes:
        name (str): The name of the pipeline.
        datasets (dict[str, Dataset]): A dictionary of datasets used in
the pipeline,
                                       where keys are dataset names and
values are Dataset objects.
        operations (list[OpType]): A list of operations to be performed
in the pipeline.
        steps (list[PipelineStep]): A list of steps that make up the
pipeline.
        output (PipelineOutput): The output configuration for the
pipeline.
        parsing_tools (list[ParsingTool]): A list of parsing tools used
in the pipeline.
                                           Defaults to an empty list.
        default_model (str | None): The default language model to use for
operations
                                       that require one. Defaults to
None.

    Example:
        ```python
        def custom_parser(text: str) -> list[str]:
            # this will convert the text in the column to uppercase
            # You should return a list of strings, where each string is a
separate document
            return [text.upper()]

        pipeline = Pipeline(
            name="document_processing_pipeline",
            datasets={
                "input_data": Dataset(type="file",
path="/path/to/input.json", parsing=[{"name": "custom_parser",
"input_key": "content", "output_key": "uppercase_content"}]),
            },
            parsing_tools=[custom_parser],
            operations=[
                MapOp(
                    name="process",
                    type="map",
                    prompt="Determine what type of document this is: {{
input.uppercase_content }}",
                    output={"schema": {"document_type": "string"}}
                ),
                ReduceOp(
                    name="summarize",
                    type="reduce",
                    reduce_key="document_type",
                    prompt="Summarize the processed contents: {% for item
in inputs %}{{ item.uppercase_content }} {% endfor %}",
                    output={"schema": {"summary": "string"}}
                )
            ],
~~~

```
            steps=[
                PipelineStep(name="process_step", input="input_data",
operations=["process"]),
                PipelineStep(name="summarize_step", input="process_step",
operations=["summarize"])
            ],
            output=PipelineOutput(type="file",
path="/path/to/output.json"),
            default_model="gpt-4o-mini"
        )
        ```

    This example shows a complete pipeline configuration with datasets,
operations,
    steps, and output settings.
    """

    def __init__(
        self,
        name: str,
        datasets: dict[str, Dataset],
        operations: list[OpType],
        steps: list[PipelineStep],
        output: PipelineOutput,
        parsing_tools: list[ParsingTool | Callable] = [],
        default_model: str | None = None,
        rate_limits: dict[str, int] | None = None,
        optimizer_config: dict[str, Any] = {},
        **kwargs,
    ):
        self.name = name
        self.datasets = datasets
        self.operations = operations
        self.steps = steps
        self.output = output
        self.parsing_tools = [
            (
                tool
                if isinstance(tool, ParsingTool)
                else ParsingTool(
                    name=tool.__name__,
function_code=inspect.getsource(tool)
)
)
            for tool in parsing_tools
        ]
        self.default_model = default_model
        self.rate_limits = rate_limits
        self.optimizer_config = optimizer_config

        # Add other kwargs to self.other_config
        self.other_config = kwargs

        self._load_env()

    def _load_env(self):
        import os

        from dotenv import load_dotenv

        # Get the current working directory
```
```

```
        cwd = os.getcwd()

        # Load .env file from the current working directory if it exists
        env_file = os.path.join(cwd, ".env")
        if os.path.exists(env_file):
            load_dotenv(env_file)

    def optimize(
        self,
        max_threads: int | None = None,
        resume: bool = False,
        save_path: str | None = None,
    ) -> "Pipeline":
        """
        Optimize the pipeline using the Optimizer.

        Args:
            max_threads (int | None): Maximum number of threads to use
for optimization.
            model (str): The model to use for optimization. Defaults to
"gpt-4o".
            resume (bool): Whether to resume optimization from a previous
state. Defaults to False.
            timeout (int): Timeout for optimization in seconds. Defaults
to 60.

        Returns:
            Pipeline: An optimized version of the pipeline.
        """
        config = self._to_dict()
        runner = DSLRunner(
            config,
            base_name=os.path.join(os.getcwd(), self.name),
            yaml_file_suffix=self.name,
            max_threads=max_threads,
        )
        optimized_config, _ = runner.optimize(
            resume=resume,
            return_pipeline=False,
            save_path=save_path,
        )

        updated_pipeline = Pipeline(
            name=self.name,
            datasets=self.datasets,
            operations=self.operations,
            steps=self.steps,
            output=self.output,
            default_model=self.default_model,
            parsing_tools=self.parsing_tools,
            optimizer_config=self.optimizer_config,
        )
        updated_pipeline._update_from_dict(optimized_config)
        return updated_pipeline

    def run(self, max_threads: int | None = None) -> float:
        """
        Run the pipeline using the DSLRunner.

        Args:
            max_threads (int | None): Maximum number of threads to use
```

```
for execution.

        Returns:
            float: The total cost of running the pipeline.
        """
        config = self._to_dict()
        runner = DSLRunner(
            config,
            base_name=os.path.join(os.getcwd(), self.name),
            yaml_file_suffix=self.name,
            max_threads=max_threads,
        )
        result = runner.load_run_save()
        return result

    def to_yaml(self, path: str) -> None:
        """
        Convert the Pipeline object to a YAML string and save it to a
file.

        Args:
            path (str): Path to save the YAML file.

        Returns:
            None
        """
        config = self._to_dict()
        with open(path, "w") as f:
            yaml.safe_dump(config, f)

        print(f"[green]Pipeline saved to {path}[/green]")

    def _to_dict(self) -> dict[str, Any]:
        """
        Convert the Pipeline object to a dictionary representation.

        Returns:
            dict[str, Any]: Dictionary representation of the Pipeline.
        """
        d = {
            "datasets": {
                name: dataset.dict() for name, dataset in
self.datasets.items()
            },
            "operations": [
                {k: v for k, v in op.dict().items() if v is not None}
                for op in self.operations
            ],
            "pipeline": {
                "steps": [
                    {k: v for k, v in step.dict().items() if v is not
None}
                    for step in self.steps
                ],
                "output": self.output.dict(),
            },
            "default_model": self.default_model,
            "parsing_tools": (
                [tool.dict() for tool in self.parsing_tools]
                if self.parsing_tools
                else None
            ),
            "optimizer_config": self.optimizer_config,
            **self.other_config,
        }
        if self.rate_limits:
            d["rate_limits"] = self.rate_limits
        return d

    def _update_from_dict(self, config: dict[str, Any]):
        """
        Update the Pipeline object from a dictionary representation.

        Args:
            config (dict[str, Any]): Dictionary representation of the
Pipeline.
        """
        self.datasets = {
            name: Dataset(
                type=dataset["type"],
                source=dataset["source"],
                path=dataset["path"],
                parsing=dataset.get("parsing"),
            )
            for name, dataset in config["datasets"].items()
        }
        self.operations = []
        for op in config["operations"]:
            op_type = op.pop("type")
            if op_type == "map":
                self.operations.append(MapOp(**op, type=op_type))
            elif op_type == "resolve":
                self.operations.append(ResolveOp(**op, type=op_type))
            elif op_type == "reduce":
                self.operations.append(ReduceOp(**op, type=op_type))
            elif op_type == "parallel_map":
                self.operations.append(ParallelMapOp(**op, type=op_type))
            elif op_type == "filter":
                self.operations.append(FilterOp(**op, type=op_type))
            elif op_type == "equijoin":
                self.operations.append(EquijoinOp(**op, type=op_type))
            elif op_type == "split":
                self.operations.append(SplitOp(**op, type=op_type))
            elif op_type == "gather":
                self.operations.append(GatherOp(**op, type=op_type))
            elif op_type == "unnest":
                self.operations.append(UnnestOp(**op, type=op_type))
            elif op_type == "cluster":
                self.operations.append(ClusterOp(**op, type=op_type))
            elif op_type == "sample":
                self.operations.append(SampleOp(**op, type=op_type))
        self.steps = [PipelineStep(**step) for step in config["pipeline"]
["steps"]]
        self.output = PipelineOutput(**config["pipeline"]["output"])
        self.default_model = config.get("default_model")
        self.parsing_tools = (
            [ParsingTool(**tool) for tool in config.get("parsing_tools",
[])]
            if config.get("parsing_tools")
            else []
        )
```

`optimize(max_threads=None, resume=False, save_path=None)`

Optimize the pipeline using the Optimizer.

Parameters:

Name	Type	Description	Default
`max_threads`	`int | None`	Maximum number of threads to use for optimization.	`None`
`model`	`str`	The model to use for optimization. Defaults to "gpt-4o".	*required*
`resume`	`bool`	Whether to resume optimization from a previous state. Defaults to False.	`False`
`timeout`	`int`	Timeout for optimization in seconds. Defaults to 60.	*required*

Returns:

Name	Type	Description
`Pipeline`	`Pipeline`	An optimized version of the pipeline.

Source code in `docetl/api.py`

```python
def optimize(
    self,
    max_threads: int | None = None,
    resume: bool = False,
    save_path: str | None = None,
) -> "Pipeline":
    """
    Optimize the pipeline using the Optimizer.

    Args:
        max_threads (int | None): Maximum number of threads to use for
optimization.
        model (str): The model to use for optimization. Defaults to "gpt-
4o".
        resume (bool): Whether to resume optimization from a previous
state. Defaults to False.
        timeout (int): Timeout for optimization in seconds. Defaults to
60.

    Returns:
        Pipeline: An optimized version of the pipeline.
    """
    config = self._to_dict()
    runner = DSLRunner(
        config,
        base_name=os.path.join(os.getcwd(), self.name),
        yaml_file_suffix=self.name,
        max_threads=max_threads,
    )
    optimized_config, _ = runner.optimize(
        resume=resume,
        return_pipeline=False,
        save_path=save_path,
    )

    updated_pipeline = Pipeline(
        name=self.name,
        datasets=self.datasets,
        operations=self.operations,
        steps=self.steps,
        output=self.output,
        default_model=self.default_model,
        parsing_tools=self.parsing_tools,
        optimizer_config=self.optimizer_config,
    )
    updated_pipeline._update_from_dict(optimized_config)
    return updated_pipeline
```

`run(max_threads=None)`

Run the pipeline using the DSLRunner.

Parameters:

Name	Type	Description	Default
`max_threads`	`int | None`	Maximum number of threads to use for execution.	`None`

Returns:

Name	Type	Description
`float`	`float`	The total cost of running the pipeline.

Source code in `docetl/api.py`

```python
def run(self, max_threads: int | None = None) -> float:
    """
    Run the pipeline using the DSLRunner.

    Args:
        max_threads (int | None): Maximum number of threads to use for
execution.

    Returns:
        float: The total cost of running the pipeline.
    """
    config = self._to_dict()
    runner = DSLRunner(
        config,
        base_name=os.path.join(os.getcwd(), self.name),
        yaml_file_suffix=self.name,
        max_threads=max_threads,
    )
    result = runner.load_run_save()
    return result
```

`to_yaml(path)`

Convert the Pipeline object to a YAML string and save it to a file.

Parameters:

Name	Type	Description	Default
`path`	`str`	Path to save the YAML file.	*required*

Returns:

Type	Description
`None`	None

Source code in `docetl/api.py`

```python
def to_yaml(self, path: str) -> None:
    """
    Convert the Pipeline object to a YAML string and save it to a file.

    Args:
        path (str): Path to save the YAML file.

    Returns:
        None
    """
    config = self._to_dict()
    with open(path, "w") as f:
        yaml.safe_dump(config, f)

    print(f"[green]Pipeline saved to {path}[/green]")
```