

Python API

Operations

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

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

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

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

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

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

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

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

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

```
docctl.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
<code>name</code>	<code>str</code>	The name of the parsing tool. This should be unique within the pipeline configuration.
<code>function_code</code>	<code>str</code>	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`

```
20 class ParsingTool(BaseModel):
21     """
22     Represents a parsing tool used for custom data parsing in the
23     pipeline.
24
25     Attributes:
26         name (str): The name of the parsing tool. This should be unique
27         within the pipeline configuration.
28         function_code (str): The Python code defining the parsing
29         function. This code will be executed
30                             to parse the input data according to the
31         specified logic. It should return a list of strings, where each string is
32         its own document.
33
34     Example:
35         ```yaml
36         parsing_tools:
37             - name: ocr_parser
38               function_code: |
39                   import pytesseract
40                   from pdf2image import convert_from_path
41                   def ocr_parser(filename: str) -> list[str]:
42                       images = convert_from_path(filename)
43                       text = ""
44                       for image in images:
45                           text += pytesseract.image_to_string(image)
46                   return [text]
47         ```
48
49     """
50
51     name: str
52     function_code: str
```

`docetl.schemas.PipelineStep`

Bases: `BaseModel`

Represents a step in the pipeline.

Attributes:

Name	Type	Description
<code>name</code>	<code>str</code>	The name of the step.

Name	Type	Description
<code>operations</code>	<code>list[dict[str, Any] str]</code>	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).
<code>input</code>	<code>str None</code>	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



```
# 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`

```

49 class PipelineStep(BaseModel):
50     """
51     Represents a step in the pipeline.
52
53     Attributes:
54         name (str): The name of the step.
55         operations (list[dict[str, Any] | str]): A list of operations to
56         be applied in this step.
57             Each operation can be either a string (the name of the
58             operation) or a dictionary
59             (for more complex configurations).
60         input (str | None): The input for this step. It can be either the
61         name of a dataset
62         or the name of a previous step. If not provided, the step will
63         use the output
64         of the previous step as its input.
65
66     Example:
67     ```python
68     # Simple step with a single operation
69     process_step = PipelineStep(
70         name="process_step",
71         input="my_dataset",
72         operations=["process"]
73     )
74
75     # Step with multiple operations
76     summarize_step = PipelineStep(
77         name="summarize_step",
78         input="process_step",
79         operations=["summarize"]
80     )
81
82     # Step with a more complex operation configuration
83     custom_step = PipelineStep(
84         name="custom_step",
85         input="previous_step",
86         operations=[
87             {
88                 "custom_operation": {
89                     "model": "gpt-4",
90                     "prompt": "Perform a custom analysis on the
91 following text:"
92                 }
93             }
94         ]
95     )
96     ```
97
98     These examples show different ways to configure pipeline steps, from
99     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`

```
102 class PipelineOutput(BaseModel):
103     """
104     Represents the output configuration for a pipeline.
105
106     Attributes:
107         type (str): The type of output. This could be 'file', 'database',
108     etc.
109         path (str): The path where the output will be stored. This could
110     be a file path,
111                     database connection string, etc., depending on the
112     type.
113         intermediate_dir (str | None): The directory to store
114     intermediate results,
115                                     if applicable. Defaults to
116     None.
117
118     Example:
119         ```python
120         output = PipelineOutput(
121             type="file",
122             path="/path/to/output.json",
123             intermediate_dir="/path/to/intermediate/results"
124         )
125         ```
126
127     type: str
128     path: str
129     intermediate_dir: str | None = None
```

`docetl.api.Pipeline`

Represents a complete document processing pipeline.

Attributes:

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

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

**Example**

```

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`

```

80 class Pipeline:
81     """
82     Represents a complete document processing pipeline.
83
84     Attributes:
85         name (str): The name of the pipeline.
86         datasets (dict[str, Dataset]): A dictionary of datasets used in
87 the pipeline,
88                                     where keys are dataset names and
89 values are Dataset objects.
90         operations (list[OpType]): A list of operations to be performed
91 in the pipeline.
92         steps (list[PipelineStep]): A list of steps that make up the
93 pipeline.
94         output (PipelineOutput): The output configuration for the
95 pipeline.
96         parsing_tools (list[ParsingTool]): A list of parsing tools used
97 in the pipeline.
98                                     Defaults to an empty list.
99         default_model (str | None): The default language model to use for
100 operations
101                                     that require one. Defaults to
102 None.
103
104     Example:
105     ```python
106     def custom_parser(text: str) -> list[str]:
107         # this will convert the text in the column to uppercase
108         # You should return a list of strings, where each string is a
109 separate document
110         return [text.upper()]
111
112     pipeline = Pipeline(
113         name="document_processing_pipeline",
114         datasets={
115             "input_data": Dataset(type="file",
116 path="/path/to/input.json", parsing=[{"name": "custom_parser",
117 "input_key": "content", "output_key": "uppercase_content"}]),
118         },
119         parsing_tools=[custom_parser],
120         operations=[
121             MapOp(
122                 name="process",
123                 type="map",
124                 prompt="Determine what type of document this is: {{
125 input.uppercase_content }}",
126                 output={"schema": {"document_type": "string"}}
127             ),
128             ReduceOp(
129                 name="summarize",
130                 type="reduce",
131                 reduce_key="document_type",
132                 prompt="Summarize the processed contents: {% for item
133 in inputs %}{{ item.uppercase_content }} {% endfor %}",
134                 output={"schema": {"summary": "string"}}
135             )
136         ],

```

```

137         steps=[
138             PipelineStep(name="process_step", input="input_data",
139 operations=["process"]),
140             PipelineStep(name="summarize_step", input="process_step",
141 operations=["summarize"])
142         ],
143         output=PipelineOutput(type="file",
144 path="/path/to/output.json"),
145         default_model="gpt-4o-mini"
146     )
147     ...
148
149     This example shows a complete pipeline configuration with datasets,
150 operations,
151 steps, and output settings.
152     """
153
154     def __init__(
155         self,
156         name: str,
157         datasets: dict[str, Dataset],
158         operations: list[OpType],
159         steps: list[PipelineStep],
160         output: PipelineOutput,
161         parsing_tools: list[ParsingTool | Callable] = [],
162         default_model: str | None = None,
163         rate_limits: dict[str, int] | None = None,
164         optimizer_config: dict[str, Any] = {},
165         **kwargs,
166     ):
167         self.name = name
168         self.datasets = datasets
169         self.operations = operations
170         self.steps = steps
171         self.output = output
172         self.parsing_tools = [
173             (
174                 tool
175                 if isinstance(tool, ParsingTool)
176                 else ParsingTool(
177                     name=tool.__name__,
178 function_code=inspect.getsource(tool)
179                 )
180             )
181             for tool in parsing_tools
182         ]
183         self.default_model = default_model
184         self.rate_limits = rate_limits
185         self.optimizer_config = optimizer_config
186
187         # Add other kwargs to self.other_config
188         self.other_config = kwargs
189
190         self._load_env()
191
192     def _load_env(self):
193         import os
194
195         from dotenv import load_dotenv
196
197         # Get the current working directory

```

```

198         cwd = os.getcwd()
199
200         # Load .env file from the current working directory if it exists
201         env_file = os.path.join(cwd, ".env")
202         if os.path.exists(env_file):
203             load_dotenv(env_file)
204
205     def optimize(
206         self,
207         max_threads: int | None = None,
208         resume: bool = False,
209         save_path: str | None = None,
210     ) -> "Pipeline":
211         """
212         Optimize the pipeline using the Optimizer.
213
214         Args:
215             max_threads (int | None): Maximum number of threads to use
216             for optimization.
217             model (str): The model to use for optimization. Defaults to
218             "gpt-4o".
219             resume (bool): Whether to resume optimization from a previous
220             state. Defaults to False.
221             timeout (int): Timeout for optimization in seconds. Defaults
222             to 60.
223
224         Returns:
225             Pipeline: An optimized version of the pipeline.
226         """
227         config = self._to_dict()
228         runner = DSLRunner(
229             config,
230             base_name=os.path.join(os.getcwd(), self.name),
231             yaml_file_suffix=self.name,
232             max_threads=max_threads,
233         )
234         optimized_config, _ = runner.optimize(
235             resume=resume,
236             return_pipeline=False,
237             save_path=save_path,
238         )
239
240         updated_pipeline = Pipeline(
241             name=self.name,
242             datasets=self.datasets,
243             operations=self.operations,
244             steps=self.steps,
245             output=self.output,
246             default_model=self.default_model,
247             parsing_tools=self.parsing_tools,
248             optimizer_config=self.optimizer_config,
249         )
250         updated_pipeline._update_from_dict(optimized_config)
251         return updated_pipeline
252
253     def run(self, max_threads: int | None = None) -> float:
254         """
255         Run the pipeline using the DSLRunner.
256
257         Args:
258             max_threads (int | None): Maximum number of threads to use

```

```

259     for execution.
260
261     Returns:
262         float: The total cost of running the pipeline.
263     """
264     config = self._to_dict()
265     runner = DSLRunner(
266         config,
267         base_name=os.path.join(os.getcwd(), self.name),
268         yaml_file_suffix=self.name,
269         max_threads=max_threads,
270     )
271     result = runner.load_run_save()
272     return result
273
274     def to_yaml(self, path: str) -> None:
275     """
276     Convert the Pipeline object to a YAML string and save it to a
277     file.
278
279     Args:
280         path (str): Path to save the YAML file.
281
282     Returns:
283         None
284     """
285     config = self._to_dict()
286     with open(path, "w") as f:
287         yaml.safe_dump(config, f)
288
289     print(f"[green]Pipeline saved to {path}[/green]")
290
291     def _to_dict(self) -> dict[str, Any]:
292     """
293     Convert the Pipeline object to a dictionary representation.
294
295     Returns:
296         dict[str, Any]: Dictionary representation of the Pipeline.
297     """
298     d = {
299         "datasets": {
300             name: dataset.dict() for name, dataset in
301 self.datasets.items()
302         },
303         "operations": [
304             {k: v for k, v in op.dict().items() if v is not None}
305             for op in self.operations
306         ],
307         "pipeline": {
308             "steps": [
309 None}
310                 {k: v for k, v in step.dict().items() if v is not
311                 for step in self.steps
312             ],
313             "output": self.output.dict(),
314         },
315         "default_model": self.default_model,
316         "parsing_tools": (
317             [tool.dict() for tool in self.parsing_tools]
318             if self.parsing_tools
319             else None

```

```

320         ),
321         "optimizer_config": self.optimizer_config,
322         **self.other_config,
323     }
324     if self.rate_limits:
325         d["rate_limits"] = self.rate_limits
326     return d
327
328     def _update_from_dict(self, config: dict[str, Any]):
329         """
330         Update the Pipeline object from a dictionary representation.
331
332         Args:
333             config (dict[str, Any]): Dictionary representation of the
334 Pipeline.
335         """
336         self.datasets = {
337             name: Dataset(
338                 type=dataset["type"],
339                 source=dataset["source"],
340                 path=dataset["path"],
341                 parsing=dataset.get("parsing"),
342             )
343             for name, dataset in config["datasets"].items()
344         }
345         self.operations = []
346         for op in config["operations"]:
347             op_type = op.pop("type")
348             if op_type == "map":
349                 self.operations.append(MapOp(**op, type=op_type))
350             elif op_type == "resolve":
351                 self.operations.append(ResolveOp(**op, type=op_type))
352             elif op_type == "reduce":
353                 self.operations.append(ReduceOp(**op, type=op_type))
354             elif op_type == "parallel_map":
355                 self.operations.append(ParallelMapOp(**op, type=op_type))
356             elif op_type == "filter":
357                 self.operations.append(FilterOp(**op, type=op_type))
358             elif op_type == "equijoin":
359                 self.operations.append(EquijoinOp(**op, type=op_type))
360             elif op_type == "split":
361                 self.operations.append(SplitOp(**op, type=op_type))
362             elif op_type == "gather":
363                 self.operations.append(GatherOp(**op, type=op_type))
364             elif op_type == "unnest":
365                 self.operations.append(UnnestOp(**op, type=op_type))
366             elif op_type == "cluster":
367                 self.operations.append(ClusterOp(**op, type=op_type))
368             elif op_type == "sample":
369                 self.operations.append(SampleOp(**op, type=op_type))
370         self.steps = [PipelineStep(**step) for step in config["pipeline"]
371 ["steps"]]
372         self.output = PipelineOutput(**config["pipeline"]["output"])
373         self.default_model = config.get("default_model")
374         self.parsing_tools = (
375             [ParsingTool(**tool) for tool in config.get("parsing_tools",
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

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

Optimize the pipeline using the Optimizer.

Parameters:

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

Returns:

Name	Type	Description
<code>Pipeline</code>	<code>Pipeline</code>	An optimized version of the pipeline.

Source code in `docetl/api.py`

```

187 def optimize(
188     self,
189     max_threads: int | None = None,
190     resume: bool = False,
191     save_path: str | None = None,
192 ) -> "Pipeline":
193     """
194     Optimize the pipeline using the Optimizer.
195
196     Args:
197         max_threads (int | None): Maximum number of threads to use for
198         optimization.
199         model (str): The model to use for optimization. Defaults to "gpt-
200         4o".
201         resume (bool): Whether to resume optimization from a previous
202         state. Defaults to False.
203         timeout (int): Timeout for optimization in seconds. Defaults to
204         60.
205
206     Returns:
207         Pipeline: An optimized version of the pipeline.
208     """
209     config = self._to_dict()
210     runner = DSLRunner(
211         config,
212         base_name=os.path.join(os.getcwd(), self.name),
213         yaml_file_suffix=self.name,
214         max_threads=max_threads,
215     )
216     optimized_config, _ = runner.optimize(
217         resume=resume,
218         return_pipeline=False,
219         save_path=save_path,
220     )
221
222     updated_pipeline = Pipeline(
223         name=self.name,
224         datasets=self.datasets,
225         operations=self.operations,
226         steps=self.steps,
227         output=self.output,
228         default_model=self.default_model,
229         parsing_tools=self.parsing_tools,
230         optimizer_config=self.optimizer_config,
231     )
232     updated_pipeline._update_from_dict(optimized_config)
233     return updated_pipeline

```

`run(max_threads=None)`

Run the pipeline using the DSLRunner.

Parameters:

Name	Type	Description	Default
<code>max_threads</code>	<code>int None</code>	Maximum number of threads to use for execution.	<code>None</code>

Returns:

Name	Type	Description
<code>float</code>	<code>float</code>	The total cost of running the pipeline.

Source code in `docetl/api.py`

```
231 def run(self, max_threads: int | None = None) -> float:
232     """
233     Run the pipeline using the DSLRunner.
234
235     Args:
236         max_threads (int | None): Maximum number of threads to use for
237         execution.
238
239     Returns:
240         float: The total cost of running the pipeline.
241     """
242     config = self._to_dict()
243     runner = DSLRunner(
244         config,
245         base_name=os.path.join(os.getcwd(), self.name),
246         yaml_file_suffix=self.name,
247         max_threads=max_threads,
248     )
249     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
<code>path</code>	<code>str</code>	Path to save the YAML file.	<i>required</i>

Returns:

Type	Description
None	None

Source code in `docetl/api.py`

```
251 def to_yaml(self, path: str) -> None:
252     """
253     Convert the Pipeline object to a YAML string and save it to a file.
254
255     Args:
256         path (str): Path to save the YAML file.
257
258     Returns:
259         None
260     """
261     config = self._to_dict()
262     with open(path, "w") as f:
263         yaml.safe_dump(config, f)
264
265     print(f"[green]Pipeline saved to {path}[/green]")
```