

# docetl Core

`docetl.DSLRunner`

Bases: `ConfigWrapper`

DSLRunner orchestrates pipeline execution by building and traversing a DAG of OpContainers. The runner uses a two-phase approach:

1. Build Phase:
2. Parses YAML config into a DAG of OpContainers
3. Each operation becomes a node connected to its dependencies
4. Special handling for equijoins which have two parent nodes
5. Validates operation syntax and schema compatibility
6. Execution Phase:
7. Starts from the final operation and pulls data through the DAG
8. Handles caching/checkpointing of intermediate results
9. Tracks costs and execution metrics
10. Manages dataset loading and result persistence

The separation between build and execution phases allows for: - Pipeline validation before any execution - Cost estimation and optimization - Partial pipeline execution for testing

Source code in `docetl/runner.py`

```

55 class DSLRunner(ConfigWrapper):
56     """
57     DSLRunner orchestrates pipeline execution by building and traversing
58     a DAG of OpContainers.
59     The runner uses a two-phase approach:
60
61     1. Build Phase:
62         - Parses YAML config into a DAG of OpContainers
63         - Each operation becomes a node connected to its dependencies
64         - Special handling for equijoins which have two parent nodes
65         - Validates operation syntax and schema compatibility
66
67     2. Execution Phase:
68         - Starts from the final operation and pulls data through the DAG
69         - Handles caching/checkpointing of intermediate results
70         - Tracks costs and execution metrics
71         - Manages dataset loading and result persistence
72
73     The separation between build and execution phases allows for:
74     - Pipeline validation before any execution
75     - Cost estimation and optimization
76     - Partial pipeline execution for testing
77     """
78
79     @classproperty
80     def schema(cls):
81         # Accessing the schema loads all operations, so only do this
82         # when we actually need it...
83         # Yes, this means DSLRunner.schema isn't really accessible to
84         # static type checkers. But it /is/ available for dynamic
85         # checking, and for generating json schema.
86
87         OpType = functools.reduce(
88             lambda a, b: a | b, [op.schema for op in
89 get_operations().values()]
90         )
91         # More pythonic implementation of the above, but only works in
92         # python 3.11:
93         # OpType = Union*[op.schema for op in
94         get_operations().values()]
95
96         class Pipeline(BaseModel):
97             config: dict[str, Any] | None
98             parsing_tools: list[schemas.ParsingTool] | None
99             datasets: dict[str, schemas.Dataset]
100             operations: list[OpType]
101             pipeline: schemas.PipelineSpec
102
103         return Pipeline
104
105     @classproperty
106     def json_schema(cls):
107         return cls.schema.model_json_schema()
108
109     def __init__(self, config: dict, max_threads: int | None = None,
110 **kwargs):
111         """

```

```

112         Initialize the DSLRunner with a YAML configuration file.
113
114         Args:
115             max_threads (int, optional): Maximum number of threads to
116 use. Defaults to None.
117         """
118         super().__init__(
119             config,
120             base_name=kwargs.pop("base_name", None),
121             yaml_file_suffix=kwargs.pop("yaml_file_suffix", None),
122             max_threads=max_threads,
123             **kwargs,
124         )
125         self.total_cost = 0
126         self._initialize_state()
127         self._setup_parsing_tools()
128         self._build_operation_graph(config)
129         self._compute_operation_hashes()
130
131         # Run initial validation
132         self._from_df_accessors = kwargs.get("from_df_accessors", False)
133         if not self._from_df_accessors:
134             self.syntax_check()
135
136         def _initialize_state(self) -> None:
137             """Initialize basic runner state and datasets"""
138             self.datasets = {}
139             self.intermediate_dir = (
140                 self.config.get("pipeline", {}).get("output",
141 {})).get("intermediate_dir")
142         )
143
144         def _setup_parsing_tools(self) -> None:
145             """Set up parsing tools from configuration"""
146             self.parsing_tool_map = create_parsing_tool_map(
147                 self.config.get("parsing_tools", None)
148             )
149
150         def _build_operation_graph(self, config: dict) -> None:
151             """Build the DAG of operations from configuration"""
152             self.config = config
153             self.op_container_map = {}
154             self.last_op_container = None
155
156             for step in self.config["pipeline"]["steps"]:
157                 self._validate_step(step)
158
159                 if step.get("input"):
160                     self._add_scan_operation(step)
161                 else:
162                     self._add_equijoin_operation(step)
163
164                 self._add_step_operations(step)
165                 self._add_step_boundary(step)
166
167         def _validate_step(self, step: dict) -> None:
168             """Validate step configuration"""
169             assert "name" in step.keys(), f"Step {step} does not have a name"
170             assert "operations" in step.keys(), f"Step {step} does not have
171 `operations`"
172

```

```

173     def _add_scan_operation(self, step: dict) -> None:
174         """Add a scan operation for input datasets"""
175         scan_op_container = OpContainer(
176             f"{step['name']}/scan_{step['input']}",
177             self,
178             {
179                 "type": "scan",
180                 "dataset_name": step["input"],
181                 "name": f"scan_{step['input']}",
182             },
183         )
184         self.op_container_map[f"{step['name']}/scan_{step['input']}"] = (
185             scan_op_container
186         )
187         if self.last_op_container:
188             scan_op_container.add_child(self.last_op_container)
189         self.last_op_container = scan_op_container
190
191     def _add_equijoin_operation(self, step: dict) -> None:
192         """Add an equijoin operation with its scan operations"""
193         equijoin_operation_name = list(step["operations"][0].keys())[0]
194         left_dataset_name = list(step["operations"][0].values())[0]
195         ["left"]
196         right_dataset_name = list(step["operations"][0].values())[0]
197         ["right"]
198
199         left_scan_op_container = OpContainer(
200             f"{step['name']}/scan_{left_dataset_name}",
201             self,
202             {
203                 "type": "scan",
204                 "dataset_name": left_dataset_name,
205                 "name": f"scan_{left_dataset_name}",
206             },
207         )
208         if self.last_op_container:
209             left_scan_op_container.add_child(self.last_op_container)
210         right_scan_op_container = OpContainer(
211             f"{step['name']}/scan_{right_dataset_name}",
212             self,
213             {
214                 "type": "scan",
215                 "dataset_name": right_dataset_name,
216                 "name": f"scan_{right_dataset_name}",
217             },
218         )
219         if self.last_op_container:
220             right_scan_op_container.add_child(self.last_op_container)
221         equijoin_op_container = OpContainer(
222             f"{step['name']}/{equijoin_operation_name}",
223             self,
224             self.find_operation(equijoin_operation_name),
225             left_name=left_dataset_name,
226             right_name=right_dataset_name,
227         )
228
229         equijoin_op_container.add_child(left_scan_op_container)
230         equijoin_op_container.add_child(right_scan_op_container)
231
232         self.last_op_container = equijoin_op_container
233         self.op_container_map[f"

```

```

234 {step['name']}/{equijoin_operation_name}] = (
235     equijoin_op_container
236 )
237 self.op_container_map[f"{step['name']}/scan_{left_dataset_name}"]
238 = (
239     left_scan_op_container
240 )
241 self.op_container_map[f"
242 {step['name']}/scan_{right_dataset_name}"] = (
243     right_scan_op_container
244 )
245
246 def _add_step_operations(self, step: dict) -> None:
247     """Add operations for a step"""
248     op_start_idx = 1 if not step.get("input") else 0
249
250     for operation_name in step["operations"][op_start_idx:]:
251         if not isinstance(operation_name, str):
252             raise ValueError(
253                 f"Operation {operation_name} in step {step['name']}
254 should be a string. "
255                 "If you intend for it to be an equijoin, don't
256 specify an input in the step."
257             )
258
259         op_container = OpContainer(
260             f"{step['name']}/{operation_name}",
261             self,
262             self.find_operation(operation_name),
263         )
264         op_container.add_child(self.last_op_container)
265         self.last_op_container = op_container
266         self.op_container_map[f"{step['name']}/{operation_name}"] =
267         op_container
268
269 def _add_step_boundary(self, step: dict) -> None:
270     """Add a step boundary node"""
271     step_boundary = StepBoundary(
272         f"{step['name']}/boundary",
273         self,
274         {"type": "step_boundary", "name": f"
275 {step['name']}/boundary"},
276     )
277     step_boundary.add_child(self.last_op_container)
278     self.op_container_map[f"{step['name']}/boundary"] = step_boundary
279     self.last_op_container = step_boundary
280
281 def _compute_operation_hashes(self) -> None:
282     """Compute hashes for operations to enable caching"""
283     op_map = {op["name"]: op for op in self.config["operations"]}
284     self.step_op_hashes = defaultdict(dict)
285
286     for step in self.config["pipeline"]["steps"]:
287         for idx, op in enumerate(step["operations"]):
288             op_name = op if isinstance(op, str) else list(op.keys())
289 [0]
290
291             all_ops_until_and_including_current = (
292                 [op_map[prev_op] for prev_op in step["operations"]
293                 [:idx]]
294                 + [op_map[op_name]]

```

```

295         + [self.config.get("system_prompt", {})]
296     )
297
298     for op in all_ops_until_and_including_current:
299         if "model" not in op:
300             op["model"] = self.default_model
301
302         all_ops_str =
303 json.dumps(all_ops_until_and_including_current)
304         self.step_op_hashes[step["name"]][op_name] =
305 hashlib.sha256(
306             all_ops_str.encode()
307         ).hexdigest()
308
309     def get_output_path(self, require=False):
310         output_path = self.config.get("pipeline", {}).get("output",
311 {}).get("path")
312         if output_path:
313             if not (
314                 output_path.lower().endswith(".json")
315                 or output_path.lower().endswith(".csv")
316             ):
317                 raise ValueError(
318                     f"Output path '{output_path}' is not a JSON or CSV
319 file. Please provide a path ending with '.json' or '.csv'."
320                 )
321             elif require:
322                 raise ValueError(
323                     "No output path specified in the configuration. Please
324 provide an output path ending with '.json' or '.csv' in the configuration
325 to use the save() method."
326                 )
327
328             return output_path
329
330     def syntax_check(self):
331         """
332         Perform a syntax check on all operations defined in the
333 configuration.
334         """
335         self.console.log("[yellow]Checking operations...[/yellow]")
336
337         # Just validate that it's a json file if specified
338         self.get_output_path()
339         current = self.last_op_container
340
341         try:
342             # Walk the last op container to check syntax
343             op_containers = []
344             if self.last_op_container:
345                 op_containers = [self.last_op_container]
346
347             while op_containers:
348                 current = op_containers.pop(0)
349                 syntax_result = current.syntax_check()
350                 self.console.log(syntax_result, end="")
351                 # Add all children to the queue
352                 op_containers.extend(current.children)
353         except Exception as e:
354             raise ValueError(
355                 f"Syntax check failed for operation '{current.name}':

```

```

356     {str(e)}"
357         )
358
359         self.console.log("[green]✓ All operations passed syntax
360 check[/green]")
361
362     def print_query_plan(self, show_boundaries=False):
363         """
364         Print a visual representation of the entire query plan using
365         indentation and arrows.
366         Operations are color-coded by step to show the pipeline structure
367         while maintaining
368         dependencies between steps.
369         """
370         if not self.last_op_container:
371             self.console.log("\n[bold]Pipeline Steps:[/bold]")
372             self.console.log(
373                 Panel("No operations in pipeline", title="Query Plan",
374 width=100)
375             )
376             self.console.log()
377             return
378
379     def _print_op(
380         op: OpContainer, indent: int = 0, step_colors: dict[str, str]
381         | None = None
382     ) -> str:
383         # Handle boundary operations based on show_boundaries flag
384         if isinstance(op, StepBoundary):
385             if show_boundaries:
386                 output = []
387                 indent_str = " " * indent
388                 step_name = op.name.split("/")[0]
389                 color = step_colors.get(step_name, "white")
390                 output.append(
391                     f"{indent_str}[{color}][bold]{op.name}[/bold]
392 [{color}]"
393                 )
394                 output.append(f"{indent_str}Type: step_boundary")
395                 if op.children:
396                     output.append(f"{indent_str}[yellow]▼[/yellow]")
397                     for child in op.children:
398                         output.append(_print_op(child, indent + 1,
399 step_colors))
400                 return "\n".join(output)
401             elif op.children:
402                 return _print_op(op.children[0], indent, step_colors)
403             return ""
404
405         # Build the string for the current operation with indentation
406         indent_str = " " * indent
407         output = []
408
409         # Color code the operation name based on its step
410         step_name = op.name.split("/")[0]
411         color = step_colors.get(step_name, "white")
412         output.append(f"{indent_str}[{color}][bold]{op.name}[/bold]
413 [{color}]"
414         )
415         output.append(f"{indent_str}Type: {op.config['type']}")
416
417         # Add schema if available

```

```

417         if "output" in op.config and "schema" in op.config["output"]:
418             output.append(f"{indent_str}Output Schema:")
419             for field, field_type in op.config["output"]
420 ["schema"].items():
421                 escaped_type = escape(str(field_type))
422                 output.append(
423                     f"{indent_str} {field}: [bright_white]
424 {escaped_type}[/bright_white]"
425                 )
426
427         # Add children
428         if op.children:
429             if op.is_equijoin:
430                 output.append(f"{indent_str}[yellow]▼ LEFT[/yellow]")
431                 output.append(_print_op(op.children[0], indent + 1,
432 step_colors))
433                 output.append(f"{indent_str}[yellow]▼
434 RIGHT[/yellow]")
435                 output.append(_print_op(op.children[1], indent + 1,
436 step_colors))
437             else:
438                 output.append(f"{indent_str}[yellow]▼[/yellow]")
439                 for child in op.children:
440                     output.append(_print_op(child, indent + 1,
441 step_colors))
442
443             return "\n".join(output)
444
445         # Get all step boundaries and extract unique step names
446         step_boundaries = [
447             op
448             for name, op in self.op_container_map.items()
449             if isinstance(op, StepBoundary)
450         ]
451         step_boundaries.sort(key=lambda x: x.name)
452
453         # Create a color map for steps - using distinct colors
454         colors = ["cyan", "magenta", "green", "yellow", "blue", "red"]
455         step_names = [b.name.split("/")[0] for b in step_boundaries]
456         step_colors = {
457             name: colors[i % len(colors)] for i, name in
458 enumerate(step_names)
459         }
460
461         # Print the legend
462         self.console.log("\n[bold]Pipeline Steps:[/bold]")
463         for step_name, color in step_colors.items():
464             self.console.log(f"[{color}]■[/{color}] {step_name}")
465
466         # Print the full query plan starting from the last step boundary
467         query_plan = _print_op(self.last_op_container,
468 step_colors=step_colors)
469         self.console.log(Panel(query_plan, title="Query Plan",
470 width=100))
471         self.console.log()
472
473         def find_operation(self, op_name: str) -> dict:
474             for operation_config in self.config["operations"]:
475                 if operation_config["name"] == op_name:
476                     return operation_config
477             raise ValueError(f"Operation '{op_name}' not found in

```



```

478 configuration.")
479
480 def load_run_save(self) -> float:
481     """
482     Execute the entire pipeline defined in the configuration.
483     """
484     output_path = self.get_output_path(require=True)
485
486     # Print the query plan
487     self.print_query_plan()
488
489     start_time = time.time()
490
491     if self.last_op_container:
492         self.load()
493         self.console.rule("[bold]Pipeline Execution[/bold]")
494         output, _, _ = self.last_op_container.next()
495         self.save(output)
496
497     execution_time = time.time() - start_time
498
499     # Print execution summary
500     summary = (
501         f"Cost: [green]${self.total_cost:.2f}[/green]\n"
502         f"Time: {execution_time:.2f}s\n"
503         + (
504             f"Cache: [dim]{self.intermediate_dir}[/dim]\n"
505             if self.intermediate_dir
506             else ""
507         )
508         + f"Output: [dim]{output_path}[/dim]"
509     )
510     self.console.log(Panel(summary, title="Execution Summary"))
511
512     return self.total_cost
513
514 def load(self) -> None:
515     """
516     Load all datasets defined in the configuration.
517     """
518     datasets = {}
519     self.console.rule("[bold>Loading Datasets[/bold]")
520
521     for name, dataset_config in self.config["datasets"].items():
522         if dataset_config["type"] == "file":
523             datasets[name] = Dataset(
524                 self,
525                 "file",
526                 dataset_config["path"],
527                 source="local",
528                 parsing=dataset_config.get("parsing", []),
529                 user_defined_parsing_tool_map=self.parsing_tool_map,
530             )
531             self.console.log(
532                 f"[green]✓[/green] Loaded dataset '{name}' from
533 {dataset_config['path']}"
534             )
535         elif dataset_config["type"] == "memory":
536             datasets[name] = Dataset(
537                 self,
538                 "memory",

```

```

539         dataset_config["path"],
540         source="local",
541         parsing=dataset_config.get("parsing", []),
542         user_defined_parsing_tool_map=self.parsing_tool_map,
543     )
544     self.console.log(
545         f"[green]✓[/green] Loaded dataset '{name}' from in-
memory data"
546     )
547     else:
548         raise ValueError(f"Unsupported dataset type:
{dataset_config['type']}")
549
550     self.datasets = {
551         name: (
552             dataset
553             if isinstance(dataset, Dataset)
554             else Dataset(self, "memory", dataset)
555         )
556         for name, dataset in datasets.items()
557     }
558     self.console.log()
559
560     def save(self, data: list[dict]) -> None:
561         """
562         Save the final output of the pipeline.
563         """
564         self.get_output_path(require=True)
565
566         output_config = self.config["pipeline"]["output"]
567         if output_config["type"] == "file":
568             # Create the directory if it doesn't exist
569             if os.path.isdir(output_config["path"]):
570                 os.makedirs(os.path.dirname(output_config["path"]),
571                             exist_ok=True)
572             if output_config["path"].lower().endswith(".json"):
573                 with open(output_config["path"], "w") as file:
574                     json.dump(data, file, indent=2)
575             else: # CSV
576                 import csv
577
578                 with open(output_config["path"], "w", newline="") as
579                     file:
580                         writer = csv.DictWriter(file,
581                                                 fieldnames=data[0].keys())
582                         limited_data = [
583                             {k: d.get(k, None) for k in data[0].keys()} for d
584                             in data
585                         ]
586                         writer.writeheader()
587                         writer.writerows(limited_data)
588
589                 self.console.log(
590                     f"[green]✓[/green] Saved to [dim]{output_config['path']}
[/dim]\n"
591                 )
592             else:
593                 raise ValueError(
594                     f"Unsupported output type: {output_config['type']}.
Supported types: file"
595                 )
596
597
598
599

```

```

600     def _load_from_checkpoint_if_exists(
601         self, step_name: str, operation_name: str
602     ) -> list[dict] | None:
603         if self.intermediate_dir is None or
604 self.config.get("bypass_cache", False):
605             return None
606
607         intermediate_config_path = os.path.join(
608             self.intermediate_dir, ".docetl_intermediate_config.json"
609         )
610
611         if not os.path.exists(intermediate_config_path):
612             return None
613
614         # Make sure the step and op name is in the checkpoint config path
615         if (
616             step_name not in self.step_op_hashes
617             or operation_name not in self.step_op_hashes[step_name]
618         ):
619             return None
620
621         # See if the checkpoint config is the same as the current step op
622 hash
623         with open(intermediate_config_path, "r") as f:
624             intermediate_config = json.load(f)
625
626         if (
627             intermediate_config.get(step_name, {}).get(operation_name,
628             ""))
629             != self.step_op_hashes[step_name][operation_name]
630         ):
631             return None
632
633         checkpoint_path = os.path.join(
634             self.intermediate_dir, step_name, f"{operation_name}.json"
635         )
636         # check if checkpoint exists
637         if os.path.exists(checkpoint_path):
638             if f"{step_name}_{operation_name}" not in self.datasets:
639                 self.datasets[f"{step_name}_{operation_name}"] = Dataset(
640                     self, "file", checkpoint_path, "local"
641                 )
642
643                 self.console.log(
644                     f"[green]✓[/green] [italic]Loaded checkpoint for
645 operation '{operation_name}' in step '{step_name}' from {checkpoint_path}
646 [/italic]"
647                 )
648
649                 return self.datasets[f"
650 {step_name}_{operation_name}"].load()
651             return None
652
653     def clear_intermediate(self) -> None:
654         """
655         Clear the intermediate directory.
656         """
657         # Remove the intermediate directory
658         if self.intermediate_dir:
659             shutil.rmtree(self.intermediate_dir)
660         return

```

```

661
662         raise ValueError("Intermediate directory not set. Cannot clear
663 intermediate.")
664
665     def _save_checkpoint(
666         self, step_name: str, operation_name: str, data: list[dict]
667     ) -> None:
668         """
669         Save a checkpoint of the current data after an operation.
670
671         This method creates a JSON file containing the current state of
672 the data
673         after an operation has been executed. The checkpoint is saved in
674 a directory
675         structure that reflects the step and operation names.
676
677         Args:
678             step_name (str): The name of the current step in the
679 pipeline.
680             operation_name (str): The name of the operation that was just
681 executed.
682             data (list[dict]): The current state of the data to be
683 checkpointed.
684
685         Note:
686             The checkpoint is saved only if a checkpoint directory has
687 been specified
688             when initializing the DSLRunner.
689         """
690         checkpoint_path = os.path.join(
691             self.intermediate_dir, step_name, f"{operation_name}.json"
692         )
693         if os.path.isdir(checkpoint_path):
694             os.makedirs(os.path.dirname(checkpoint_path), exist_ok=True)
695         with open(checkpoint_path, "w") as f:
696             json.dump(data, f)
697
698         # Update the intermediate config file with the hash for this
699 step/operation
700         # so that future runs can validate and reuse this checkpoint.
701         if self.intermediate_dir:
702             intermediate_config_path = os.path.join(
703                 self.intermediate_dir, ".docetl_intermediate_config.json"
704             )
705
706             # Initialize or load existing intermediate configuration
707             if os.path.exists(intermediate_config_path):
708                 try:
709                     with open(intermediate_config_path, "r") as cfg_file:
710                         intermediate_config: dict[str, dict[str, str]] =
711 json.load(
712                             cfg_file
713                         )
714                 except json.JSONDecodeError:
715                     # If the file is corrupted, start fresh to avoid
716 crashes
717                     intermediate_config = {}
718             else:
719                 intermediate_config = {}
720
721         # Ensure nested dict structure exists

```

```

722         step_dict = intermediate_config.setdefault(step_name, {})
723
724         # Write (or overwrite) the hash for the current operation
725         step_dict[operation_name] = self.step_op_hashes[step_name]
726     [operation_name]
727
728     # Persist the updated configuration
729     with open(intermediate_config_path, "w") as cfg_file:
730         json.dump(intermediate_config, cfg_file, indent=2)
731
732     self.console.log(
733         f"[green]✓ [italic]Intermediate saved for operation
734     '{operation_name}' in step '{step_name}' at {checkpoint_path}[/italic]
735     [/green]"
736     )
737
738     def should_optimize(
739         self, step_name: str, op_name: str, **kwargs
740     ) -> tuple[str, float, list[dict[str, Any]], list[dict[str, Any]]]:
741         self.load()
742
743         # Augment the kwargs with the runner's config if not already
744         provided
745         kwargs["litellm_kwargs"] = self.config.get("optimizer_config",
746     {}).get(
747             "litellm_kwargs", {}
748         )
749         kwargs["rewrite_agent_model"] =
750     self.config.get("optimizer_config", {}).get(
751         "rewrite_agent_model", "gpt-4o"
752     )
753         kwargs["judge_agent_model"] = self.config.get("optimizer_config",
754     {}).get(
755             "judge_agent_model", "gpt-4o-mini"
756         )
757
758         builder = Optimizer(self, **kwargs)
759         self.optimizer = builder
760         result = builder.should_optimize(step_name, op_name)
761         return result
762
763     def optimize(
764         self,
765         save: bool = False,
766         return_pipeline: bool = True,
767         **kwargs,
768     ) -> tuple[dict | "DSLRunner", float]:
769
770         if not self.last_op_container:
771             raise ValueError("No operations in pipeline. Cannot
772     optimize.")
773
774         self.load()
775
776         # Augment the kwargs with the runner's config if not already
777         provided
778         kwargs["litellm_kwargs"] = self.config.get("optimizer_config",
779     {}).get(
780             "litellm_kwargs", {}
781         )
782         kwargs["rewrite_agent_model"] =

```

```

783 self.config.get("optimizer_config", {}).get(
784     "rewrite_agent_model", "gpt-4o"
785 )
786 kwargs["judge_agent_model"] = self.config.get("optimizer_config",
787 {}).get(
788     "judge_agent_model", "gpt-4o-mini"
789 )
790
791 save_path = kwargs.get("save_path", None)
792 # Pop the save_path from kwargs
793 kwargs.pop("save_path", None)
794
795 builder = Optimizer(
796     self,
797     **kwargs,
798 )
799 self.optimizer = builder
800 llm_api_cost = builder.optimize()
801 operations_cost = self.total_cost
802 self.total_cost += llm_api_cost
803
804 # Log the cost of optimization
805 self.console.log(
806     f"[green italic]💰 Total cost: ${self.total_cost:.4f}[/green
807 italic]"
808 )
809 self.console.log(
810     f"[green italic]├ Operation execution cost:
811 ${operations_cost:.4f}[/green italic]"
812 )
813 self.console.log(
814     f"[green italic]└ Optimization cost: ${llm_api_cost:.4f}
815 [/green italic]"
816 )
817
818 if save:
819     # If output path is provided, save the optimized config to
820     that path
821     if kwargs.get("save_path"):
822         save_path = kwargs["save_path"]
823         if not os.path.isabs(save_path):
824             save_path = os.path.join(os.getcwd(), save_path)
825         builder.save_optimized_config(save_path)
826         self.optimized_config_path = save_path
827     else:
828         builder.save_optimized_config(f"
829 {self.base_name}_opt.yaml")
830         self.optimized_config_path = f"{self.base_name}_opt.yaml"
831
832     if return_pipeline:
833         return (
834             DSLRunner(builder.clean_optimized_config(),
835 self.max_threads),
836             self.total_cost,
837         )
838
839     return builder.clean_optimized_config(), self.total_cost
840
841 def _run_operation(
842     self,
843     op_config: dict[str, Any],

```

```

        input_data: list[dict[str, Any]] | dict[str, Any],
        return_instance: bool = False,
        is_build: bool = False,
    ) -> list[dict[str, Any]] | tuple[list[dict[str, Any]],
BaseOperation, float]:
    """
    Run a single operation based on its configuration.

    This method creates an instance of the appropriate operation
    class and executes it.
    It also updates the total operation cost.

    Args:
        op_config (dict[str, Any]): The configuration of the
        operation to run.
        input_data (list[dict[str, Any]]): The input data for the
        operation.
        return_instance (bool, optional): If True, return the
        operation instance along with the output data.

    Returns:
        list[dict[str, Any]] | tuple[list[dict[str, Any]],
BaseOperation, float]:
        If return_instance is False, returns the output data.
        If return_instance is True, returns a tuple of the output
        data, the operation instance, and the cost.
    """
    operation_class = get_operation(op_config["type"])

    oc_kwargs = {
        "runner": self,
        "config": op_config,
        "default_model": self.config["default_model"],
        "max_threads": self.max_threads,
        "console": self.console,
        "status": self.status,
    }
    operation_instance = operation_class(**oc_kwargs)
    if op_config["type"] == "equijoin":
        output_data, cost = operation_instance.execute(
            input_data["left_data"], input_data["right_data"]
        )
    elif op_config["type"] == "filter":
        output_data, cost = operation_instance.execute(input_data,
is_build)
    else:
        output_data, cost = operation_instance.execute(input_data)

    self.total_cost += cost

    if return_instance:
        return output_data, operation_instance
    else:
        return output_data

    def _flush_partial_results(
        self, operation_name: str, batch_index: int, data: list[dict]
    ) -> None:
        """
        Save partial (batch-level) results from an operation to a
        directory named

```

```
        '<operation_name>_batches' inside the intermediate directory.

    Args:
        operation_name (str): The name of the operation, e.g.
        'extract_medications'.
        batch_index (int): Zero-based index of the batch.
        data (list[dict]): Batch results to write to disk.
    """
    if not self.intermediate_dir:
        return

    op_batches_dir = os.path.join(
        self.intermediate_dir, f"{operation_name}_batches"
    )
    os.makedirs(op_batches_dir, exist_ok=True)

    # File name: 'batch_0.json', 'batch_1.json', etc.
    checkpoint_path = os.path.join(op_batches_dir,
        f"batch_{batch_index}.json")

    with open(checkpoint_path, "w") as f:
        json.dump(data, f)

    self.console.log(
        f"[green]✓[/green] [italic]Partial checkpoint saved for
        '{operation_name}', "
        f"batch {batch_index} at '{checkpoint_path}'[/italic]"
    )
```

`__init__(config, max_threads=None, **kwargs)`

Initialize the DSLRunner with a YAML configuration file.

Parameters:

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



Source code in `docetl/runner.py`

```

105 def __init__(self, config: dict, max_threads: int | None = None,
106 **kwargs):
107     """
108     Initialize the DSLRunner with a YAML configuration file.
109
110     Args:
111         max_threads (int, optional): Maximum number of threads to use.
112     Defaults to None.
113     """
114     super().__init__(
115         config,
116         base_name=kwargs.pop("base_name", None),
117         yaml_file_suffix=kwargs.pop("yaml_file_suffix", None),
118         max_threads=max_threads,
119         **kwargs,
120     )
121     self.total_cost = 0
122     self._initialize_state()
123     self._setup_parsing_tools()
124     self._build_operation_graph(config)
125     self._compute_operation_hashes()
126
127     # Run initial validation
128     self._from_df_accessors = kwargs.get("from_df_accessors", False)
129     if not self._from_df_accessors:
130         self.syntax_check()

```

**`clear_intermediate()`**

Clear the intermediate directory.

Source code in `docetl/runner.py`

```

593 def clear_intermediate(self) -> None:
594     """
595     Clear the intermediate directory.
596     """
597     # Remove the intermediate directory
598     if self.intermediate_dir:
599         shutil.rmtree(self.intermediate_dir)
600         return
601
602     raise ValueError("Intermediate directory not set. Cannot clear
intermediate.")

```

**`load()`**

Load all datasets defined in the configuration.

Source code in `docetl/runner.py`

```

469 def load(self) -> None:
470     """
471     Load all datasets defined in the configuration.
472     """
473     datasets = {}
474     self.console.rule("[bold]Loading Datasets[/bold]")
475
476     for name, dataset_config in self.config["datasets"].items():
477         if dataset_config["type"] == "file":
478             datasets[name] = Dataset(
479                 self,
480                 "file",
481                 dataset_config["path"],
482                 source="local",
483                 parsing=dataset_config.get("parsing", []),
484                 user_defined_parsing_tool_map=self.parsing_tool_map,
485             )
486             self.console.log(
487                 f"[green]✓[/green] Loaded dataset '{name}' from
488 {dataset_config['path']}"
489             )
490         elif dataset_config["type"] == "memory":
491             datasets[name] = Dataset(
492                 self,
493                 "memory",
494                 dataset_config["path"],
495                 source="local",
496                 parsing=dataset_config.get("parsing", []),
497                 user_defined_parsing_tool_map=self.parsing_tool_map,
498             )
499             self.console.log(
500                 f"[green]✓[/green] Loaded dataset '{name}' from in-memory
501 data"
502             )
503         else:
504             raise ValueError(f"Unsupported dataset type:
505 {dataset_config['type']}")
506
507     self.datasets = {
508         name: (
509             dataset
510             if isinstance(dataset, Dataset)
511             else Dataset(self, "memory", dataset)
512         )
513         for name, dataset in datasets.items()
514     }
515     self.console.log()

```

**load\_run\_save()**

Execute the entire pipeline defined in the configuration.

Source code in `docetl/runner.py`

```

435 def load_run_save(self) -> float:
436     """
437     Execute the entire pipeline defined in the configuration.
438     """
439     output_path = self.get_output_path(require=True)
440
441     # Print the query plan
442     self.print_query_plan()
443
444     start_time = time.time()
445
446     if self.last_op_container:
447         self.load()
448         self.console.rule("[bold]Pipeline Execution[/bold]")
449         output, _, _ = self.last_op_container.next()
450         self.save(output)
451
452     execution_time = time.time() - start_time
453
454     # Print execution summary
455     summary = (
456         f"Cost: [green]${self.total_cost:.2f}[/green]\n"
457         f"Time: {execution_time:.2f}s\n"
458         + (
459             f"Cache: [dim]{self.intermediate_dir}[/dim]\n"
460             if self.intermediate_dir
461             else ""
462         )
463         + f"Output: [dim]{output_path}[/dim]"
464     )
465     self.console.log(Panel(summary, title="Execution Summary"))
466
467     return self.total_cost

```

**`print_query_plan(show_boundaries=False)`**

Print a visual representation of the entire query plan using indentation and arrows. Operations are color-coded by step to show the pipeline structure while maintaining dependencies between steps.

Source code in `docetl/runner.py`

```

334 def print_query_plan(self, show_boundaries=False):
335     """
336     Print a visual representation of the entire query plan using
337     indentation and arrows.
338     Operations are color-coded by step to show the pipeline structure
339     while maintaining
340     dependencies between steps.
341     """
342     if not self.last_op_container:
343         self.console.log("\n[bold]Pipeline Steps:[/bold]")
344         self.console.log(
345             Panel("No operations in pipeline", title="Query Plan",
346                 width=100)
347         )
348         self.console.log()
349         return
350
351     def _print_op(
352         op: OpContainer, indent: int = 0, step_colors: dict[str, str] |
353         None = None
354     ) -> str:
355         # Handle boundary operations based on show_boundaries flag
356         if isinstance(op, StepBoundary):
357             if show_boundaries:
358                 output = []
359                 indent_str = " " * indent
360                 step_name = op.name.split("/")[0]
361                 color = step_colors.get(step_name, "white")
362                 output.append(
363                     f"{indent_str}[{color}][bold]{op.name}[/bold]
364                 [{color}]"
365                 )
366                 output.append(f"{indent_str}Type: step_boundary")
367                 if op.children:
368                     output.append(f"{indent_str}[yellow]▼[/yellow]")
369                     for child in op.children:
370                         output.append(_print_op(child, indent + 1,
371 step_colors))
372                 return "\n".join(output)
373             elif op.children:
374                 return _print_op(op.children[0], indent, step_colors)
375             return ""
376
377         # Build the string for the current operation with indentation
378         indent_str = " " * indent
379         output = []
380
381         # Color code the operation name based on its step
382         step_name = op.name.split("/")[0]
383         color = step_colors.get(step_name, "white")
384         output.append(f"{indent_str}[{color}][bold]{op.name}[/bold]
385         [{color}]"
386         )
387         output.append(f"{indent_str}Type: {op.config['type']}")
388
389         # Add schema if available
390         if "output" in op.config and "schema" in op.config["output"]:
391             output.append(f"{indent_str}Output Schema:")

```

```

391         for field, field_type in op.config["output"]
392     ["schema"].items():
393         escaped_type = escape(str(field_type))
394         output.append(
395             f"{indent_str} {field}: [bright_white]{escaped_type}
396 [/bright_white]"
397         )
398
399     # Add children
400     if op.children:
401         if op.is_equijoin:
402             output.append(f"{indent_str}[yellow]▼ LEFT[/yellow]")
403             output.append(_print_op(op.children[0], indent + 1,
404 step_colors))
405             output.append(f"{indent_str}[yellow]▼ RIGHT[/yellow]")
406             output.append(_print_op(op.children[1], indent + 1,
407 step_colors))
408         else:
409             output.append(f"{indent_str}[yellow]▼[/yellow]")
410             for child in op.children:
411                 output.append(_print_op(child, indent + 1,
412 step_colors))
413
414     return "\n".join(output)
415
416     # Get all step boundaries and extract unique step names
417     step_boundaries = [
418         op
419         for name, op in self.op_container_map.items()
420         if isinstance(op, StepBoundary)
421     ]
422     step_boundaries.sort(key=lambda x: x.name)
423
424     # Create a color map for steps - using distinct colors
425     colors = ["cyan", "magenta", "green", "yellow", "blue", "red"]
426     step_names = [b.name.split("/")[0] for b in step_boundaries]
427     step_colors = {
428         name: colors[i % len(colors)] for i, name in
429         enumerate(step_names)
430     }
431
432     # Print the legend
433     self.console.log("\n[bold]Pipeline Steps:[/bold]")
434     for step_name, color in step_colors.items():
435         self.console.log(f"[{color}]■[/{color}] {step_name}")
436
437     # Print the full query plan starting from the last step boundary
438     query_plan = _print_op(self.last_op_container,
439 step_colors=step_colors)
440     self.console.log(Panel(query_plan, title="Query Plan", width=100))
441     self.console.log()

```

#### save(data)

Save the final output of the pipeline.

Source code in `docetl/runner.py`

```

514 def save(self, data: list[dict]) -> None:
515     """
516     Save the final output of the pipeline.
517     """
518     self.get_output_path(require=True)
519
520     output_config = self.config["pipeline"]["output"]
521     if output_config["type"] == "file":
522         # Create the directory if it doesn't exist
523         if os.path.dirname(output_config["path"]):
524             os.makedirs(os.path.dirname(output_config["path"]),
525 exist_ok=True)
526         if output_config["path"].lower().endswith(".json"):
527             with open(output_config["path"], "w") as file:
528                 json.dump(data, file, indent=2)
529         else: # CSV
530             import csv
531
532             with open(output_config["path"], "w", newline="") as file:
533                 writer = csv.DictWriter(file, fieldnames=data[0].keys())
534                 limited_data = [
535                     {k: d.get(k, None) for k in data[0].keys()} for d in
536 data
537                 ]
538                 writer.writeheader()
539                 writer.writerows(limited_data)
540             self.console.log(
541                 f"[green]✓[/green] Saved to [dim]{output_config['path']}"
542             [/dim]\n"
543             )
544         else:
545             raise ValueError(
546                 f"Unsupported output type: {output_config['type']}. Supported
547 types: file"
548             )

```

`syntax_check()`

Perform a syntax check on all operations defined in the configuration.

Source code in `docetl/runner.py`

```
305 def syntax_check(self):
306     """
307     Perform a syntax check on all operations defined in the
308     configuration.
309     """
310     self.console.log("[yellow]Checking operations...[/yellow]")
311
312     # Just validate that it's a json file if specified
313     self.get_output_path()
314     current = self.last_op_container
315
316     try:
317         # Walk the last op container to check syntax
318         op_containers = []
319         if self.last_op_container:
320             op_containers = [self.last_op_container]
321
322         while op_containers:
323             current = op_containers.pop(0)
324             syntax_result = current.syntax_check()
325             self.console.log(syntax_result, end="")
326             # Add all children to the queue
327             op_containers.extend(current.children)
328         except Exception as e:
329             raise ValueError(
330                 f"Syntax check failed for operation '{current.name}':
331                 {str(e)}"
332             )
333
334     self.console.log("[green]✓ All operations passed syntax
335     check[/green]")
```

`docetl.Optimizer`

Orchestrates the optimization of a DocETL pipeline by analyzing and potentially rewriting operations marked for optimization. Works with the runner's pull-based execution model to maintain lazy evaluation while improving pipeline efficiency.

**Source code in** `docetl/optimizer.py`

```

48 class Optimizer:
49     """
50     Orchestrates the optimization of a DocETL pipeline by analyzing and
51     potentially rewriting
52     operations marked for optimization. Works with the runner's pull-
53     based execution model
54     to maintain lazy evaluation while improving pipeline efficiency.
55     """
56
57     def __init__(
58         self,
59         runner: "DSLRunner",
60         rewrite_agent_model: str = "gpt-4o",
61         judge_agent_model: str = "gpt-4o-mini",
62         litellm_kwargs: dict[str, Any] = {},
63         resume: bool = False,
64         timeout: int = 60,
65     ):
66         """
67         Initialize the optimizer with a runner instance and
68         configuration.
69         Sets up optimization parameters, caching, and cost tracking.
70
71         Args:
72             yaml_file (str): Path to the YAML configuration file.
73             model (str): The name of the language model to use. Defaults
74             to "gpt-4o".
75             resume (bool): Whether to resume optimization from a previous
76             run. Defaults to False.
77             timeout (int): Timeout in seconds for operations. Defaults to
78             60.
79
80         Attributes:
81             config (Dict): Stores the loaded configuration from the YAML
82             file.
83             console (Console): Rich console for formatted output.
84             max_threads (int): Maximum number of threads for parallel
85             processing.
86             base_name (str): Base name used for file paths.
87             yaml_file_suffix (str): Suffix for YAML configuration files.
88             runner (DSLRunner): The DSL runner instance.
89             status: Status tracking for the runner.
90             optimized_config (Dict): A copy of the original config to be
91             optimized.
92             llm_client (LLMClient): Client for interacting with the
93             language model.
94             timeout (int): Timeout for operations in seconds.
95             resume (bool): Whether to resume from previous optimization.
96             captured_output (CapturedOutput): Captures output during
97             optimization.
98             sample_cache (Dict): Maps operation names to tuples of
99             (output_data, sample_size).
100             optimized_ops_path (str): Path to store optimized operations.
101             sample_size_map (Dict): Maps operation types to sample sizes.
102
103             The method also calls print_optimizer_config() to display the
104             initial configuration.

```



```

105         """
106         self.config = runner.config
107         self.console = runner.console
108         self.max_threads = runner.max_threads
109
110         self.base_name = runner.base_name
111         self.yaml_file_suffix = runner.yaml_file_suffix
112         self.runner = runner
113         self.status = runner.status
114
115         self.optimized_config = copy.deepcopy(self.config)
116
117         # Get the rate limits from the optimizer config
118         rate_limits = self.config.get("optimizer_config",
119 {}).get("rate_limits", {})
120
121         self.llm_client = LLMClient(
122             runner,
123             rewrite_agent_model,
124             judge_agent_model,
125             rate_limits,
126             **litellm_kwargs,
127         )
128         self.timeout = timeout
129         self.resume = resume
130         self.captured_output = CapturedOutput()
131
132         # Add sample cache for build operations
133         self.sample_cache = {} # Maps operation names to (output_data,
134 sample_size)
135
136         home_dir = os.environ.get("DOCETL_HOME_DIR",
137 os.path.expanduser("~"))
138         cache_dir = os.path.join(home_dir,
139 f".docetl/cache/{runner.yaml_file_suffix}")
140         os.makedirs(cache_dir, exist_ok=True)
141
142         # Hash the config to create a unique identifier
143         config_hash =
144 hashlib.sha256(str(self.config).encode()).hexdigest()
145         self.optimized_ops_path = f"{cache_dir}/{config_hash}.yaml"
146
147         # Update sample size map
148         self.sample_size_map = SAMPLE_SIZE_MAP
149         if self.config.get("optimizer_config", {}).get("sample_sizes",
150 {}):
151             self.sample_size_map.update(self.config["optimizer_config"]
152 ["sample_sizes"])
153
154         if not self.runner._from_df_accessors:
155             self.print_optimizer_config()
156
157         def print_optimizer_config(self):
158             """
159             Print the current configuration of the optimizer.
160
161             This method uses the Rich console to display a formatted output
162             of the optimizer's
163             configuration. It includes details such as the YAML file path,
164             sample sizes for
165             different operation types, maximum number of threads, the

```

```

166     language model being used,
167         and the timeout setting.
168
169     The output is color-coded and formatted for easy readability,
170     with a header and
171     separator lines to clearly delineate the configuration
172     information.
173     """
174     self.console.log(
175         Panel.fit(
176             "[bold cyan]Optimizer Configuration[/bold cyan]\n"
177             f"[yellow]Sample Size:[/yellow] {self.sample_size_map}\n"
178             f"[yellow]Max Threads:[/yellow] {self.max_threads}\n"
179             f"[yellow]Rewrite Agent Model:[/yellow]
180 {self.llm_client.rewrite_agent_model}\n"
181             f"[yellow]Judge Agent Model:[/yellow]
182 {self.llm_client.judge_agent_model}\n"
183             f"[yellow]Rate Limits:[/yellow]
184 {self.config.get('optimizer_config', {}).get('rate_limits', {})}\n",
185             title="Optimizer Configuration",
186         )
187     )
188
189     def _insert_empty_resolve_operations(self):
190         """
191         Determines whether to insert resolve operations in the pipeline.
192
193         For each reduce operation in the tree, checks if it has any map
194         operation as a descendant
195         without a resolve operation in between. If found, inserts an
196         empty resolve operation
197         right after the reduce operation.
198
199         The method modifies the operation container tree in-place.
200
201         Returns:
202             None
203         """
204         if not self.runner.last_op_container:
205             return
206
207         def find_map_without_resolve(container, visited=None):
208             """Helper to find first map descendant without a resolve
209             operation in between."""
210             if visited is None:
211                 visited = set()
212
213             if container.name in visited:
214                 return None
215             visited.add(container.name)
216
217             if not container.children:
218                 return None
219
220             for child in container.children:
221                 if child.config["type"] == "map":
222                     return child
223                 if child.config["type"] == "resolve":
224                     continue
225             map_desc = find_map_without_resolve(child, visited)
226             if map_desc:

```

```

227         return map_desc
228     return None
229
230     # Walk down the operation container tree
231     containers_to_check = [self.runner.last_op_container]
232     while containers_to_check:
233         current = containers_to_check.pop(0)
234
235         # Skip if this is a boundary or has no children
236         if isinstance(current, StepBoundary) or not current.children:
237             containers_to_check.extend(current.children)
238             continue
239
240         # Get the step name from the container's name
241         step_name = current.name.split("/")[0]
242
243         # Check if current container is a reduce operation
244         if current.config["type"] == "reduce" and current.config.get(
245             "synthesize_resolve", True
246         ):
247             reduce_key = current.config.get("reduce_key", "_all")
248             if isinstance(reduce_key, str):
249                 reduce_key = [reduce_key]
250
251             if "_all" not in reduce_key:
252                 # Find map descendant without resolve
253                 map_desc = find_map_without_resolve(current)
254                 if map_desc:
255                     # Synthesize an empty resolver
256                     self.console.log(
257                         "[yellow]Synthesizing empty resolver
258 operation:[/yellow]"
259                     )
260                     self.console.log(
261                         f" • [cyan]Reduce operation:[/cyan] [bold]
262 {current.name}[/bold]"
263                     )
264                     self.console.log(
265                         f" • [cyan]Step:[/cyan] [bold]{step_name}
266 [/bold]"
267                     )
268
269                     # Create new resolve operation config
270                     new_resolve_name = (
271                         f"synthesized_resolve_{len(self.config['operations'])}"
272                     )
273                     new_resolve_config = {
274                         "name": new_resolve_name,
275                         "type": "resolve",
276                         "empty": True,
277                         "optimize": True,
278                         "embedding_model": "text-embedding-3-small",
279                         "resolution_model": self.config.get(
280                             "default_model", "gpt-4o-mini"
281                         ),
282                         "comparison_model": self.config.get(
283                             "default_model", "gpt-4o-mini"
284                         ),
285                         "_intermediates": {
286                             "map_prompt":

```

```

288 map_desc.config.get("prompt"),
289             "reduce_key": reduce_key,
290         },
291     }
292
293     # Add to operations list
294
295     self.config["operations"].append(new_resolve_config)
296
297     # Create new resolve container
298     new_resolve_container = OpContainer(
299         f"{step_name}/{new_resolve_name}",
300         self.runner,
301         new_resolve_config,
302     )
303
304     # Insert the new container between reduce and its
305     children
306
307     new_resolve_container.children = current.children
308     for child in new_resolve_container.children:
309         child.parent = new_resolve_container
310     current.children = [new_resolve_container]
311     new_resolve_container.parent = current
312
313     # Add to container map
314     self.runner.op_container_map[
315         f"{step_name}/{new_resolve_name}"
316     ] = new_resolve_container
317
318     # Add children to the queue
319
320     containers_to_check.extend(new_resolve_container.children)
321
322     def _add_map_prompts_to_reduce_operations(self):
323         """
324         Add relevant map prompts to reduce operations based on their
325         reduce keys.
326
327         This method walks the operation container tree to find map
328         operations and their
329         output schemas, then associates those with reduce operations that
330         use those keys.
331         When a reduce operation is found, it looks through its
332         descendants to find the
333         relevant map operations and adds their prompts.
334
335         The method modifies the operation container tree in-place.
336         """
337         if not self.runner.last_op_container:
338             return
339
340         def find_map_prompts_for_keys(container, keys, visited=None):
341             """Helper to find map prompts for given keys in the
342             container's descendants."""
343             if visited is None:
344                 visited = set()
345
346             if container.name in visited:
347                 return []
348             visited.add(container.name)

```

```

349         prompts = []
350         if container.config["type"] == "map":
351             output_schema = container.config.get("output",
352 {})).get("schema", {})
353             if any(key in output_schema for key in keys):
354                 prompts.append(container.config.get("prompt", ""))
355
356         for child in container.children:
357             prompts.extend(find_map_prompts_for_keys(child, keys,
358 visited))
359
360         return prompts
361
362     # Walk down the operation container tree
363     containers_to_check = [self.runner.last_op_container]
364     while containers_to_check:
365         current = containers_to_check.pop(0)
366
367         # Skip if this is a boundary or has no children
368         if isinstance(current, StepBoundary) or not current.children:
369             containers_to_check.extend(current.children)
370             continue
371
372         # If this is a reduce operation, find relevant map prompts
373         if current.config["type"] == "reduce":
374             reduce_keys = current.config.get("reduce_key", [])
375             if isinstance(reduce_keys, str):
376                 reduce_keys = [reduce_keys]
377
378             # Find map prompts in descendants
379             relevant_prompts = find_map_prompts_for_keys(current,
380 reduce_keys)
381
382             if relevant_prompts:
383                 current.config["_intermediates"] =
384 current.config.get(
385                     "_intermediates", {}
386                 )
387                 current.config["_intermediates"]["last_map_prompt"] =
388 (
389                     relevant_prompts[-1]
390                 )
391
392             # Add children to the queue
393             containers_to_check.extend(current.children)
394
395     def should_optimize(
396         self, step_name: str, op_name: str
397     ) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]], float]:
398         """
399         Analyzes whether an operation should be optimized by running it
400         on a sample of input data
401         and evaluating potential optimizations. Returns the optimization
402         suggestion and relevant data.
403         """
404         self.console.rule("[bold cyan]Beginning Pipeline Assessment[/bold
405 cyan]")
406
407         self._insert_empty_resolve_operations()
408
409         node_of_interest = self.runner.op_container_map[f"

```

```

410 {step_name}/{op_name}"]
411
412     # Run the node_of_interest's children
413     input_data = []
414     for child in node_of_interest.children:
415         input_data.append(
416             child.next(
417                 is_build=True,
418
419 sample_size_needed=SAMPLE_SIZE_MAP.get(child.config["type"]),
420             )[0]
421         )
422
423     # Set the step
424     self.captured_output.set_step(step_name)
425
426     # Determine whether we should optimize the node_of_interest
427     if (
428         node_of_interest.config.get("type") == "map"
429         or node_of_interest.config.get("type") == "filter"
430     ):
431         # Create instance of map optimizer
432         map_optimizer = MapOptimizer(
433             self.runner,
434             self.runner._run_operation,
435             is_filter=node_of_interest.config.get("type") ==
436 "filter",
437         )
438         should_optimize_output, input_data, output_data = (
439             map_optimizer.should_optimize(node_of_interest.config,
440 input_data[0])
441         )
442         elif node_of_interest.config.get("type") == "reduce":
443             reduce_optimizer = ReduceOptimizer(
444                 self.runner,
445                 self.runner._run_operation,
446             )
447             should_optimize_output, input_data, output_data = (
448                 reduce_optimizer.should_optimize(node_of_interest.config,
449 input_data[0])
450             )
451         elif node_of_interest.config.get("type") == "resolve":
452             resolve_optimizer = JoinOptimizer(
453                 self.runner,
454                 node_of_interest.config,
455                 target_recall=self.config.get("optimizer_config", {})
456                     .get("resolve", {})
457                     .get("target_recall", 0.95),
458             )
459             _, should_optimize_output =
460 resolve_optimizer.should_optimize(input_data[0])
461
462         # if should_optimize_output is empty, then we should move to
463 the reduce operation
464         if should_optimize_output == "":
465             return "", [], [], 0.0
466         else:
467             return "", [], [], 0.0
468
469     # Return the string and operation cost
470     return (

```

```

471         should_optimize_output,
472         input_data,
473         output_data,
474         self.runner.total_cost + self.llm_client.total_cost,
475     )
476
477     def optimize(self) -> float:
478         """
479         Optimizes the entire pipeline by walking the operation DAG and
480         applying
481         operation-specific optimizers where marked. Returns the total
482         optimization cost.
483         """
484         self.console.rule("[bold cyan]Beginning Pipeline Rewrites[/bold
485         cyan]")
486
487         # If self.resume is True and there's a checkpoint, load it
488         if self.resume:
489             if os.path.exists(self.optimized_ops_path):
490                 # Load the yaml and change the runner with it
491                 with open(self.optimized_ops_path, "r") as f:
492                     partial_optimized_config = yaml.safe_load(f)
493                     self.console.log(
494                         "[yellow>Loading partially optimized pipeline
495                         from checkpoint...[/yellow]"
496                     )
497
498                 self.runner._build_operation_graph(partial_optimized_config)
499             else:
500                 self.console.log(
501                     "[yellow>No checkpoint found, starting optimization
502                     from scratch...[/yellow]"
503                 )
504
505         else:
506             self._insert_empty_resolve_operations()
507
508         # Start with the last operation container and visit each child
509         self.runner.last_op_container.optimize()
510
511         flush_cache(self.console)
512
513         # Print the query plan
514         self.console.rule("[bold cyan]Optimized Query Plan[/bold cyan]")
515         self.runner.print_query_plan()
516
517         return self.llm_client.total_cost
518
519     def _optimize_equijoin(
520         self,
521         op_config: dict[str, Any],
522         left_name: str,
523         right_name: str,
524         left_data: list[dict[str, Any]],
525         right_data: list[dict[str, Any]],
526         run_operation: Callable[
527             [dict[str, Any], list[dict[str, Any]]], list[dict[str, Any]]
528         ],
529     ) -> tuple[list[dict[str, Any]], dict[str, list[dict[str, Any]]],
530     str, str]:
531         """

```

```

532         Optimizes an equijoin operation by analyzing join conditions and
533         potentially inserting
534         map operations to improve join efficiency. Returns the optimized
535         configuration and updated data.
536         """
537         max_iterations = 2
538         new_left_name = left_name
539         new_right_name = right_name
540         new_steps = []
541         for _ in range(max_iterations):
542             join_optimizer = JoinOptimizer(
543                 self.runner,
544                 op_config,
545                 target_recall=self.runner.config.get("optimizer_config",
546 {}))
547                 .get("equijoin", {})
548                 .get("target_recall", 0.95),
549
550             estimated_selectivity=self.runner.config.get("optimizer_config", {})
551                 .get("equijoin", {})
552                 .get("estimated_selectivity", None),
553             )
554             optimized_config, cost, agent_results =
555             join_optimizer.optimize_equijoin(
556                 left_data, right_data
557             )
558             self.runner.total_cost += cost
559             # Update the operation config with the optimized values
560             op_config.update(optimized_config)
561
562             if not agent_results.get("optimize_map", False):
563                 break # Exit the loop if no more map optimizations are
564             necessary
565
566             # Update the status to indicate we're optimizing a map
567             operation
568             output_key = agent_results["output_key"]
569             if self.runner.status:
570                 self.runner.status.update(
571                     f"Optimizing map operation for {output_key}
572 extraction to help with the equijoin"
573                 )
574             map_prompt = agent_results["map_prompt"]
575             dataset_to_transform = (
576                 left_data
577                 if agent_results["dataset_to_transform"] == "left"
578                 else right_data
579             )
580
581             # Create a new step for the map operation
582             map_operation = {
583                 "name": f"synthesized_{output_key}_extraction",
584                 "type": "map",
585                 "prompt": map_prompt,
586                 "model": self.config.get("default_model", "gpt-4o-mini"),
587                 "output": {"schema": {output_key: "string"}},
588                 "optimize": False,
589             }
590
591             # Optimize the map operation
592             if map_operation["optimize"]:

```



```

593         dataset_to_transform_sample = (
594             random.sample(dataset_to_transform,
595 self.sample_size_map.get("map"))
596             if self.config.get("optimizer_config", {}).get(
597                 "random_sample", False
598             )
599             else dataset_to_transform[:
600 self.sample_size_map.get("map")]
601         )
602         optimized_map_operations = self._optimize_map(
603             map_operation, dataset_to_transform_sample
604         )
605     else:
606         optimized_map_operations = [map_operation]
607
608     new_step = {
609         "name": f"synthesized_{output_key}_extraction",
610         "input": (
611             left_name
612             if agent_results["dataset_to_transform"] == "left"
613             else right_name
614         ),
615         "operations": [mo["name"] for mo in
616 optimized_map_operations],
617     }
618     if agent_results["dataset_to_transform"] == "left":
619         new_left_name = new_step["name"]
620     else:
621         new_right_name = new_step["name"]
622
623     new_steps.append((new_step["name"], new_step,
624 optimized_map_operations))
625
626     # Now run the optimized map operation on the entire
627     dataset_to_transform
628     for op in optimized_map_operations:
629         dataset_to_transform = run_operation(op,
630 dataset_to_transform)
631
632     # Update the appropriate dataset for the next iteration
633     if agent_results["dataset_to_transform"] == "left":
634         left_data = dataset_to_transform
635     else:
636         right_data = dataset_to_transform
637
638     if self.runner.status:
639         self.runner.status.update(
640             f"Optimizing equijoin operation with {output_key}
641 extraction"
642         )
643
644     return op_config, new_steps, new_left_name, new_right_name
645
646     def checkpoint_optimized_ops(self) -> None:
647         """
648         Generates the clean config and saves it to the
649 self.optimized_ops_path
650         This is used to resume optimization from a previous run
651         """
652         clean_config = self.clean_optimized_config()
653         with open(self.optimized_ops_path, "w") as f:

```

```

654         yaml.safe_dump(clean_config, f, default_flow_style=False,
655 width=80)
656
657     # Recursively resolve all anchors and aliases
658     @staticmethod
659     def resolve_anchors(data):
660         """
661         Recursively resolve all anchors and aliases in a nested data
662         structure.
663
664         This static method traverses through dictionaries and lists,
665         resolving any YAML anchors and aliases.
666
667         Args:
668             data: The data structure to resolve. Can be a dictionary,
669             list, or any other type.
670
671         Returns:
672             The resolved data structure with all anchors and aliases
673             replaced by their actual values.
674         """
675         if isinstance(data, dict):
676             return {k: Optimizer.resolve_anchors(v) for k, v in
677 data.items()}
678         elif isinstance(data, list):
679             return [Optimizer.resolve_anchors(item) for item in data]
680         else:
681             return data
682
683     def clean_optimized_config(self) -> dict:
684         """
685         Creates a clean YAML configuration from the optimized operation
686         containers,
687         removing internal fields and organizing operations into proper
688         pipeline steps.
689         """
690         if not self.runner.last_op_container:
691             return self.config
692
693         # Create a clean copy of the config
694         datasets = {}
695         for dataset_name, dataset_config in self.config.get("datasets",
696 {}).items():
697             if dataset_config["type"] == "memory":
698                 dataset_config_copy = copy.deepcopy(dataset_config)
699                 dataset_config_copy["path"] = "in-memory data"
700                 datasets[dataset_name] = dataset_config_copy
701             else:
702                 datasets[dataset_name] = dataset_config
703
704         clean_config = {
705             "datasets": datasets,
706             "operations": [],
707             "pipeline": self.runner.config.get(
708                 "pipeline", {}
709             ).copy(), # Copy entire pipeline config
710         }
711
712         # Reset steps to regenerate
713         clean_config["pipeline"]["steps"] = []
714

```

```

715         # Keep track of operations we've seen to avoid duplicates
716         seen_operations = set()
717
718         def clean_operation(op_container: OpContainer) -> dict:
719             """Remove internal fields from operation config"""
720             op_config = op_container.config
721             clean_op = copy.deepcopy(op_config)
722
723             clean_op.pop("_intermediates", None)
724
725             # If op has already been optimized, remove the
726             recursively_optimize and optimize fields
727             if op_container.is_optimized:
728                 for field in ["recursively_optimize", "optimize"]:
729                     clean_op.pop(field, None)
730
731             return clean_op
732
733         def process_container(container, current_step=None):
734             """Process an operation container and its dependencies"""
735             # Skip step boundaries
736             if isinstance(container, StepBoundary):
737                 if container.children:
738                     return process_container(container.children[0],
739 current_step)
740                 return None, None
741
742             # Get step name from container name
743             step_name = container.name.split("/") [0]
744
745             # If this is a new step, create it
746             if not current_step or current_step["name"] != step_name:
747                 current_step = {"name": step_name, "operations": []}
748                 clean_config["pipeline"]["steps"].insert(0, current_step)
749
750             # Skip scan operations but process their dependencies
751             if container.config["type"] == "scan":
752                 if container.children:
753                     return process_container(container.children[0],
754 current_step)
755                 return None, current_step
756
757             # Handle equijoin operations
758             if container.is_equijoin:
759                 # Add operation to list if not seen
760                 if container.name not in seen_operations:
761                     op_config = clean_operation(container)
762                     clean_config["operations"].append(op_config)
763                     seen_operations.add(container.name)
764
765                 # Add to step operations with left and right inputs
766                 current_step["operations"].insert(
767                     0,
768                     {
769                         container.config["name"]: {
770                             "left": container.kwargs["left_name"],
771                             "right": container.kwargs["right_name"],
772                         }
773                     },
774                 )

```

```

        # Process both children
        if container.children:
            process_container(container.children[0],
current_step)
            process_container(container.children[1],
current_step)
        else:
            # Add operation to list if not seen
            if container.name not in seen_operations:
                op_config = clean_operation(container)
                clean_config["operations"].append(op_config)
                seen_operations.add(container.name)

            # Add to step operations
            current_step["operations"].insert(0,
container.config["name"])

            # Process children
            if container.children:
                for child in container.children:
                    process_container(child, current_step)

            return container, current_step

# Start processing from the last container
process_container(self.runner.last_op_container)

# Add inputs to steps based on their first operation
for step in clean_config["pipeline"]["steps"]:
    first_op = step["operations"][0]
    if isinstance(first_op, dict): # This is an equijoin
        continue # Equijoin steps don't need an input field
    elif len(step["operations"]) > 0:
        # Find the first non-scan operation's input by looking at
its dependencies
        op_container = self.runner.op_container_map.get(
            f"{step['name']}/{first_op}"
        )
        if op_container and op_container.children:
            child = op_container.children[0]
            while (
                child
                and child.config["type"] == "step_boundary"
                and child.children
            ):
                child = child.children[0]
            if child and child.config["type"] == "scan":
                step["input"] = child.config["dataset_name"]

# Preserve all other config key-value pairs from original config
for key, value in self.config.items():
    if key not in ["datasets", "operations", "pipeline"]:
        clean_config[key] = value

return clean_config

def save_optimized_config(self, optimized_config_path: str):
    """
    Saves the optimized configuration to a YAML file after resolving
all references
    and cleaning up internal optimization artifacts.

```

```
        """
        resolved_config = self.clean_optimized_config()

        with open(optimized_config_path, "w") as f:
            yaml.safe_dump(resolved_config, f, default_flow_style=False,
                           width=80)
            self.console.log(
                f"[green italic]📄 Optimized config saved to "
                f"{optimized_config_path}[/green italic]"
            )
```

```
__init__(runner, rewrite_agent_model='gpt-4o', judge_agent_model='gpt-4o-mini',
         litellm_kwargs={}, resume=False, timeout=60)
```

Initialize the optimizer with a runner instance and configuration. Sets up optimization parameters, caching, and cost tracking.

Parameters:

Name	Type	Description	Default
yaml_file	str	Path to the YAML configuration file.	required
model	str	The name of the language model to use. Defaults to "gpt-4o".	required
resume	bool	Whether to resume optimization from a previous run. Defaults to False.	False
timeout	int	Timeout in seconds for operations. Defaults to 60.	60

Attributes:

Name	Type	Description
config	Dict	Stores the loaded configuration from the YAML file.
console	Console	Rich console for formatted output.
max_threads	int	Maximum number of threads for parallel processing.

Name	Type	Description
<code>base_name</code>	<code>str</code>	Base name used for file paths.
<code>yaml_file_suffix</code>	<code>str</code>	Suffix for YAML configuration files.
<code>runner</code>	<code>DSLRunner</code>	The DSL runner instance.
<code>status</code>	<code>DSLRunner</code>	Status tracking for the runner.
<code>optimized_config</code>	<code>Dict</code>	A copy of the original config to be optimized.
<code>llm_client</code>	<code>LLMClient</code>	Client for interacting with the language model.
<code>timeout</code>	<code>int</code>	Timeout for operations in seconds.
<code>resume</code>	<code>bool</code>	Whether to resume from previous optimization.
<code>captured_output</code>	<code>CapturedOutput</code>	Captures output during optimization.
<code>sample_cache</code>	<code>Dict</code>	Maps operation names to tuples of (output_data, sample_size).
<code>optimized_ops_path</code>	<code>str</code>	Path to store optimized operations.
<code>sample_size_map</code>	<code>Dict</code>	Maps operation types to sample sizes.

The method also calls `print_optimizer_config()` to display the initial configuration.

Source code in `docetl/optimizer.py`

```

55 def __init__(
56     self,
57     runner: "DSLRunner",
58     rewrite_agent_model: str = "gpt-4o",
59     judge_agent_model: str = "gpt-4o-mini",
60     litellm_kwargs: dict[str, Any] = {},
61     resume: bool = False,
62     timeout: int = 60,
63 ):
64     """
65     Initialize the optimizer with a runner instance and configuration.
66     Sets up optimization parameters, caching, and cost tracking.
67
68     Args:
69         yaml_file (str): Path to the YAML configuration file.
70         model (str): The name of the language model to use. Defaults to
71         "gpt-4o".
72         resume (bool): Whether to resume optimization from a previous
73         run. Defaults to False.
74         timeout (int): Timeout in seconds for operations. Defaults to 60.
75
76     Attributes:
77         config (Dict): Stores the loaded configuration from the YAML
78         file.
79         console (Console): Rich console for formatted output.
80         max_threads (int): Maximum number of threads for parallel
81         processing.
82         base_name (str): Base name used for file paths.
83         yaml_file_suffix (str): Suffix for YAML configuration files.
84         runner (DSLRunner): The DSL runner instance.
85         status: Status tracking for the runner.
86         optimized_config (Dict): A copy of the original config to be
87         optimized.
88         llm_client (LLMClient): Client for interacting with the language
89         model.
90         timeout (int): Timeout for operations in seconds.
91         resume (bool): Whether to resume from previous optimization.
92         captured_output (CapturedOutput): Captures output during
93         optimization.
94         sample_cache (Dict): Maps operation names to tuples of
95         (output_data, sample_size).
96         optimized_ops_path (str): Path to store optimized operations.
97         sample_size_map (Dict): Maps operation types to sample sizes.
98
99     The method also calls print_optimizer_config() to display the initial
100     configuration.
101     """
102     self.config = runner.config
103     self.console = runner.console
104     self.max_threads = runner.max_threads
105
106     self.base_name = runner.base_name
107     self.yaml_file_suffix = runner.yaml_file_suffix
108     self.runner = runner
109     self.status = runner.status
110
111     self.optimized_config = copy.deepcopy(self.config)

```

```

112
113     # Get the rate limits from the optimizer config
114     rate_limits = self.config.get("optimizer_config",
115 {} ).get("rate_limits", {})
116
117     self.llm_client = LLMClient(
118         runner,
119         rewrite_agent_model,
120         judge_agent_model,
121         rate_limits,
122         **litellm_kwargs,
123     )
124     self.timeout = timeout
125     self.resume = resume
126     self.captured_output = CapturedOutput()
127
128     # Add sample cache for build operations
129     self.sample_cache = {} # Maps operation names to (output_data,
130 sample_size)
131
132     home_dir = os.environ.get("DOCETL_HOME_DIR", os.path.expanduser("~"))
133     cache_dir = os.path.join(home_dir,
134 f".docetl/cache/{runner.yaml_file_suffix}")
135     os.makedirs(cache_dir, exist_ok=True)
136
137     # Hash the config to create a unique identifier
138     config_hash = hashlib.sha256(str(self.config).encode()).hexdigest()
139     self.optimized_ops_path = f"{cache_dir}/{config_hash}.yaml"
140
141     # Update sample size map
142     self.sample_size_map = SAMPLE_SIZE_MAP
143     if self.config.get("optimizer_config", {}).get("sample_sizes", {}):
144         self.sample_size_map.update(self.config["optimizer_config"]
145 ["sample_sizes"])
146
147     if not self.runner._from_df_accessors:
148         self.print_optimizer_config()

```

#### `checkpoint_optimized_ops()`

Generates the clean config and saves it to the `self.optimized_ops_path` This is used to resume optimization from a previous run

#### Source code in `docetl/optimizer.py`

```

565 def checkpoint_optimized_ops(self) -> None:
566     """
567     Generates the clean config and saves it to the
568     self.optimized_ops_path
569     This is used to resume optimization from a previous run
570     """
571     clean_config = self.clean_optimized_config()
572     with open(self.optimized_ops_path, "w") as f:
573         yaml.safe_dump(clean_config, f, default_flow_style=False,
574 width=80)

```



**`clean_optimized_config()`**

Creates a clean YAML configuration from the optimized operation containers, removing internal fields and organizing operations into proper pipeline steps.

Source code in `docetl/optimizer.py`

```

595 def clean_optimized_config(self) -> dict:
596     """
597     Creates a clean YAML configuration from the optimized operation
598     containers,
599     removing internal fields and organizing operations into proper
600     pipeline steps.
601     """
602     if not self.runner.last_op_container:
603         return self.config
604
605     # Create a clean copy of the config
606     datasets = {}
607     for dataset_name, dataset_config in self.config.get("datasets",
608 {}):
609         if dataset_config["type"] == "memory":
610             dataset_config_copy = copy.deepcopy(dataset_config)
611             dataset_config_copy["path"] = "in-memory data"
612             datasets[dataset_name] = dataset_config_copy
613         else:
614             datasets[dataset_name] = dataset_config
615
616     clean_config = {
617         "datasets": datasets,
618         "operations": [],
619         "pipeline": self.runner.config.get(
620             "pipeline", {}
621         ).copy(), # Copy entire pipeline config
622     }
623
624     # Reset steps to regenerate
625     clean_config["pipeline"]["steps"] = []
626
627     # Keep track of operations we've seen to avoid duplicates
628     seen_operations = set()
629
630     def clean_operation(op_container: OpContainer) -> dict:
631         """Remove internal fields from operation config"""
632         op_config = op_container.config
633         clean_op = copy.deepcopy(op_config)
634
635         clean_op.pop("_intermediates", None)
636
637         # If op has already been optimized, remove the
638         recursively_optimize and optimize fields
639         if op_container.is_optimized:
640             for field in ["recursively_optimize", "optimize"]:
641                 clean_op.pop(field, None)
642
643         return clean_op
644
645     def process_container(container, current_step=None):
646         """Process an operation container and its dependencies"""
647         # Skip step boundaries
648         if isinstance(container, StepBoundary):
649             if container.children:
650                 return process_container(container.children[0],
651 current_step)

```

```

652         return None, None
653
654     # Get step name from container name
655     step_name = container.name.split("/")[0]
656
657     # If this is a new step, create it
658     if not current_step or current_step["name"] != step_name:
659         current_step = {"name": step_name, "operations": []}
660         clean_config["pipeline"]["steps"].insert(0, current_step)
661
662     # Skip scan operations but process their dependencies
663     if container.config["type"] == "scan":
664         if container.children:
665             return process_container(container.children[0],
666 current_step)
667         return None, current_step
668
669     # Handle equijoin operations
670     if container.is_equijoin:
671         # Add operation to list if not seen
672         if container.name not in seen_operations:
673             op_config = clean_operation(container)
674             clean_config["operations"].append(op_config)
675             seen_operations.add(container.name)
676
677         # Add to step operations with left and right inputs
678         current_step["operations"].insert(
679             0,
680             {
681                 container.config["name"]: {
682                     "left": container.kwargs["left_name"],
683                     "right": container.kwargs["right_name"],
684                 }
685             },
686         )
687
688     # Process both children
689     if container.children:
690         process_container(container.children[0], current_step)
691         process_container(container.children[1], current_step)
692     else:
693         # Add operation to list if not seen
694         if container.name not in seen_operations:
695             op_config = clean_operation(container)
696             clean_config["operations"].append(op_config)
697             seen_operations.add(container.name)
698
699         # Add to step operations
700         current_step["operations"].insert(0,
701 container.config["name"])
702
703     # Process children
704     if container.children:
705         for child in container.children:
706             process_container(child, current_step)
707
708     return container, current_step
709
710     # Start processing from the last container
711     process_container(self.runner.last_op_container)
712

```

```

713     # Add inputs to steps based on their first operation
714     for step in clean_config["pipeline"]["steps"]:
715         first_op = step["operations"][0]
716         if isinstance(first_op, dict): # This is an equijoin
717             continue # Equijoin steps don't need an input field
718         elif len(step["operations"]) > 0:
719             # Find the first non-scan operation's input by looking at its
720             dependencies
721             op_container = self.runner.op_container_map.get(
722                 f"{step['name']}/{first_op}"
723             )
724             if op_container and op_container.children:
725                 child = op_container.children[0]
726                 while (
727                     child
728                     and child.config["type"] == "step_boundary"
729                     and child.children
730                 ):
731                     child = child.children[0]
732                 if child and child.config["type"] == "scan":
733                     step["input"] = child.config["dataset_name"]

# Preserve all other config key-value pairs from original config
for key, value in self.config.items():
    if key not in ["datasets", "operations", "pipeline"]:
        clean_config[key] = value

return clean_config

```

**optimize()**

Optimizes the entire pipeline by walking the operation DAG and applying operation-specific optimizers where marked. Returns the total optimization cost.

Source code in `docetl/optimizer.py`

```

418 def optimize(self) -> float:
419     """
420     Optimizes the entire pipeline by walking the operation DAG and
421     applying
422     operation-specific optimizers where marked. Returns the total
423     optimization cost.
424     """
425     self.console.rule("[bold cyan]Beginning Pipeline Rewrites[/bold
426     cyan]")
427
428     # If self.resume is True and there's a checkpoint, load it
429     if self.resume:
430         if os.path.exists(self.optimized_ops_path):
431             # Load the yaml and change the runner with it
432             with open(self.optimized_ops_path, "r") as f:
433                 partial_optimized_config = yaml.safe_load(f)
434                 self.console.log(
435                     "[yellow>Loading partially optimized pipeline from
436                     checkpoint...[/yellow]"
437                 )
438
439             self.runner._build_operation_graph(partial_optimized_config)
440         else:
441             self.console.log(
442                 "[yellow>No checkpoint found, starting optimization from
443                 scratch...[/yellow]"
444             )
445
446     else:
447         self._insert_empty_resolve_operations()
448
449     # Start with the last operation container and visit each child
450     self.runner.last_op_container.optimize()
451
452     flush_cache(self.console)
453
454     # Print the query plan
455     self.console.rule("[bold cyan]Optimized Query Plan[/bold cyan]")
456     self.runner.print_query_plan()
457
458     return self.llm_client.total_cost

```

**print\_optimizer\_config()**

Print the current configuration of the optimizer.

This method uses the Rich console to display a formatted output of the optimizer's configuration. It includes details such as the YAML file path, sample sizes for different operation types, maximum number of threads, the language model being used, and the timeout setting.

The output is color-coded and formatted for easy readability, with a header and separator lines to clearly delineate the configuration information.

Source code in `docetl/optimizer.py`

```
137 def print_optimizer_config(self):
138     """
139     Print the current configuration of the optimizer.
140
141     This method uses the Rich console to display a formatted output of
142     the optimizer's
143     configuration. It includes details such as the YAML file path, sample
144     sizes for
145     different operation types, maximum number of threads, the language
146     model being used,
147     and the timeout setting.
148
149     The output is color-coded and formatted for easy readability, with a
150     header and
151     separator lines to clearly delineate the configuration information.
152     """
153     self.console.log(
154         Panel.fit(
155             "[bold cyan]Optimizer Configuration[/bold cyan]\n"
156             f"[yellow]Sample Size:[/yellow] {self.sample_size_map}\n"
157             f"[yellow]Max Threads:[/yellow] {self.max_threads}\n"
158             f"[yellow]Rewrite Agent Model:[/yellow]
159 {self.llm_client.rewrite_agent_model}\n"
            f"[yellow]Judge Agent Model:[/yellow]
{self.llm_client.judge_agent_model}\n"
            f"[yellow]Rate Limits:[/yellow]
{self.config.get('optimizer_config', {}).get('rate_limits', {})}\n",
            title="Optimizer Configuration",
        )
    )
```

**resolve\_anchors(data)** `staticmethod`

Recursively resolve all anchors and aliases in a nested data structure.


This static method traverses through dictionaries and lists, resolving any YAML anchors and aliases.

**Parameters:**

Name	Type	Description	Default
<code>data</code>		The data structure to resolve. Can be a dictionary, list, or any other type.	<i>required</i>

**Returns:**

Type	Description
	The resolved data structure with all anchors and aliases replaced by their actual values.

” Source code in `docetl/optimizer.py` 

```
575 @staticmethod
576 def resolve_anchors(data):
577     """
578     Recursively resolve all anchors and aliases in a nested data
579     structure.
580
581     This static method traverses through dictionaries and lists,
582     resolving any YAML anchors and aliases.
583
584     Args:
585         data: The data structure to resolve. Can be a dictionary, list,
586         or any other type.
587
588     Returns:
589         The resolved data structure with all anchors and aliases replaced
590         by their actual values.
591     """
592     if isinstance(data, dict):
593         return {k: Optimizer.resolve_anchors(v) for k, v in data.items()}
594     elif isinstance(data, list):
595         return [Optimizer.resolve_anchors(item) for item in data]
596     else:
597         return data
```

#### `save_optimized_config(optimized_config_path)`

Saves the optimized configuration to a YAML file after resolving all references and cleaning up internal optimization artifacts.

Source code in `docetl/optimizer.py`

```
734 def save_optimized_config(self, optimized_config_path: str):
735     """
736     Saves the optimized configuration to a YAML file after resolving all
737     references
738     and cleaning up internal optimization artifacts.
739     """
740     resolved_config = self.clean_optimized_config()
741
742     with open(optimized_config_path, "w") as f:
743         yaml.safe_dump(resolved_config, f, default_flow_style=False,
744             width=80)
745         self.console.log(
746             f"[green italic]📄 Optimized config saved to
747             {optimized_config_path}[/green italic]"
748         )
```

**`should_optimize(step_name, op_name)`**

Analyzes whether an operation should be optimized by running it on a sample of input data and evaluating potential optimizations. Returns the optimization suggestion and relevant data.



Source code in `docetl/optimizer.py`

```

346 def should_optimize(
347     self, step_name: str, op_name: str
348 ) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]], float]:
349     """
350     Analyzes whether an operation should be optimized by running it on a
351     sample of input data
352     and evaluating potential optimizations. Returns the optimization
353     suggestion and relevant data.
354     """
355     self.console.rule("[bold cyan]Beginning Pipeline Assessment[/bold
356 cyan]")
357
358     self._insert_empty_resolve_operations()
359
360     node_of_interest = self.runner.op_container_map[f"
361 {step_name}/{op_name}"]
362
363     # Run the node_of_interest's children
364     input_data = []
365     for child in node_of_interest.children:
366         input_data.append(
367             child.next(
368                 is_build=True,
369
370 sample_size_needed=SAMPLE_SIZE_MAP.get(child.config["type"]),
371             )[0]
372         )
373
374     # Set the step
375     self.captured_output.set_step(step_name)
376
377     # Determine whether we should optimize the node_of_interest
378     if (
379         node_of_interest.config.get("type") == "map"
380         or node_of_interest.config.get("type") == "filter"
381     ):
382         # Create instance of map optimizer
383         map_optimizer = MapOptimizer(
384             self.runner,
385             self.runner._run_operation,
386             is_filter=node_of_interest.config.get("type") == "filter",
387         )
388         should_optimize_output, input_data, output_data = (
389             map_optimizer.should_optimize(node_of_interest.config,
390 input_data[0])
391         )
392     elif node_of_interest.config.get("type") == "reduce":
393         reduce_optimizer = ReduceOptimizer(
394             self.runner,
395             self.runner._run_operation,
396         )
397         should_optimize_output, input_data, output_data = (
398             reduce_optimizer.should_optimize(node_of_interest.config,
399 input_data[0])
400         )
401     elif node_of_interest.config.get("type") == "resolve":
402         resolve_optimizer = JoinOptimizer(

```

```
403         self.runner,
404         node_of_interest.config,
405         target_recall=self.config.get("optimizer_config", {})
406         .get("resolve", {})
407         .get("target_recall", 0.95),
408     )
409     _, should_optimize_output =
410     resolve_optimizer.should_optimize(input_data[0])
411
412     # if should_optimize_output is empty, then we should move to the
413     reduce operation
414     if should_optimize_output == "":
415         return "", [], [], 0.0
416     else:
417         return "", [], [], 0.0
418
419     # Return the string and operation cost
420     return (
421         should_optimize_output,
422         input_data,
423         output_data,
424         self.runner.total_cost + self.llm_client.total_cost,
425     )
```