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`

```python
class DSLRunner(ConfigWrapper):
    """
    DSLRunner orchestrates pipeline execution by building and traversing
a DAG of OpContainers.
    The runner uses a two-phase approach:

    1. Build Phase:
       - Parses YAML config into a DAG of OpContainers
       - Each operation becomes a node connected to its dependencies
       - Special handling for equijoins which have two parent nodes
       - Validates operation syntax and schema compatibility

    2. Execution Phase:
       - Starts from the final operation and pulls data through the DAG
       - Handles caching/checkpointing of intermediate results
       - Tracks costs and execution metrics
       - 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
    """

    @classproperty
    def schema(cls):
        # Accessing the schema loads all operations, so only do this
        # when we actually need it...
        # Yes, this means DSLRunner.schema isn't really accessible to
        # static type checkers. But it /is/ available for dynamic
        # checking, and for generating json schema.

        OpType = functools.reduce(
            lambda a, b: a | b, [op.schema for op in
get_operations().values()]
        )
        # More pythonic implementation of the above, but only works in
python 3.11:
        # OpType = Union[*[op.schema for op in
get_operations().values()]]

        class Pipeline(BaseModel):
            config: dict[str, Any] | None
            parsing_tools: list[schemas.ParsingTool] | None
            datasets: dict[str, schemas.Dataset]
            operations: list[OpType]
            pipeline: schemas.PipelineSpec

        return Pipeline

    @classproperty
    def json_schema(cls):
        return cls.schema.model_json_schema()

    def __init__(self, config: dict, max_threads: int | None = None,
**kwargs):
        """
```

```
        Initialize the DSLRunner with a YAML configuration file.

        Args:
            max_threads (int, optional): Maximum number of threads to
use. Defaults to None.
        """
        super().__init__(
            config,
            base_name=kwargs.pop("base_name", None),
            yaml_file_suffix=kwargs.pop("yaml_file_suffix", None),
            max_threads=max_threads,
            **kwargs,
        )
        self.total_cost = 0
        self._initialize_state()
        self._setup_parsing_tools()
        self._build_operation_graph(config)
        self._compute_operation_hashes()

        # Run initial validation
        self._from_df_accessors = kwargs.get("from_df_accessors", False)
        if not self._from_df_accessors:
            self.syntax_check()

    def _initialize_state(self) -> None:
        """Initialize basic runner state and datasets"""
        self.datasets = {}
        self.intermediate_dir = (
            self.config.get("pipeline", {}).get("output",
{}).get("intermediate_dir")
        )

    def _setup_parsing_tools(self) -> None:
        """Set up parsing tools from configuration"""
        self.parsing_tool_map = create_parsing_tool_map(
            self.config.get("parsing_tools", None)
        )

    def _build_operation_graph(self, config: dict) -> None:
        """Build the DAG of operations from configuration"""
        self.config = config
        self.op_container_map = {}
        self.last_op_container = None

        for step in self.config["pipeline"]["steps"]:
            self._validate_step(step)

            if step.get("input"):
                self._add_scan_operation(step)
            else:
                self._add_equijoin_operation(step)

            self._add_step_operations(step)
            self._add_step_boundary(step)

    def _validate_step(self, step: dict) -> None:
        """Validate step configuration"""
        assert "name" in step.keys(), f"Step {step} does not have a name"
        assert "operations" in step.keys(), f"Step {step} does not have
`operations`"

```

```python
    def _add_scan_operation(self, step: dict) -> None:
        """Add a scan operation for input datasets"""
        scan_op_container = OpContainer(
            f"{step['name']}/scan_{step['input']}",
            self,
            {
                "type": "scan",
                "dataset_name": step["input"],
                "name": f"scan_{step['input']}",
            },
        )
        self.op_container_map[f"{step['name']}/scan_{step['input']}"] = (
            scan_op_container
        )
        if self.last_op_container:
            scan_op_container.add_child(self.last_op_container)
        self.last_op_container = scan_op_container

    def _add_equijoin_operation(self, step: dict) -> None:
        """Add an equijoin operation with its scan operations"""
        equijoin_operation_name = list(step["operations"][0].keys())[0]
        left_dataset_name = list(step["operations"][0].values())[0]
["left"]
        right_dataset_name = list(step["operations"][0].values())[0]
["right"]

        left_scan_op_container = OpContainer(
            f"{step['name']}/scan_{left_dataset_name}",
            self,
            {
                "type": "scan",
                "dataset_name": left_dataset_name,
                "name": f"scan_{left_dataset_name}",
            },
        )
        if self.last_op_container:
            left_scan_op_container.add_child(self.last_op_container)
        right_scan_op_container = OpContainer(
            f"{step['name']}/scan_{right_dataset_name}",
            self,
            {
                "type": "scan",
                "dataset_name": right_dataset_name,
                "name": f"scan_{right_dataset_name}",
            },
        )
        if self.last_op_container:
            right_scan_op_container.add_child(self.last_op_container)
        equijoin_op_container = OpContainer(
            f"{step['name']}/{equijoin_operation_name}",
            self,
            self.find_operation(equijoin_operation_name),
            left_name=left_dataset_name,
            right_name=right_dataset_name,
        )

        equijoin_op_container.add_child(left_scan_op_container)
        equijoin_op_container.add_child(right_scan_op_container)

        self.last_op_container = equijoin_op_container
        self.op_container_map[f"
```

```
{step['name']}/{equijoin_operation_name}"] = (
            equijoin_op_container
        )
        self.op_container_map[f"{step['name']}/scan_{left_dataset_name}"]
= (
            left_scan_op_container
        )
        self.op_container_map[f"
{step['name']}/scan_{right_dataset_name}"] = (
            right_scan_op_container
        )

    def _add_step_operations(self, step: dict) -> None:
        """Add operations for a step"""
        op_start_idx = 1 if not step.get("input") else 0

        for operation_name in step["operations"][op_start_idx:]:
            if not isinstance(operation_name, str):
                raise ValueError(
                    f"Operation {operation_name} in step {step['name']}
should be a string. "
                    "If you intend for it to be an equijoin, don't
specify an input in the step."
                )

            op_container = OpContainer(
                f"{step['name']}/{operation_name}",
                self,
                self.find_operation(operation_name),
            )
            op_container.add_child(self.last_op_container)
            self.last_op_container = op_container
            self.op_container_map[f"{step['name']}/{operation_name}"] =
op_container

    def _add_step_boundary(self, step: dict) -> None:
        """Add a step boundary node"""
        step_boundary = StepBoundary(
            f"{step['name']}/boundary",
            self,
            {"type": "step_boundary", "name": f"
{step['name']}/boundary"},
        )
        step_boundary.add_child(self.last_op_container)
        self.op_container_map[f"{step['name']}/boundary"] = step_boundary
        self.last_op_container = step_boundary

    def _compute_operation_hashes(self) -> None:
        """Compute hashes for operations to enable caching"""
        op_map = {op["name"]: op for op in self.config["operations"]}
        self.step_op_hashes = defaultdict(dict)

        for step in self.config["pipeline"]["steps"]:
            for idx, op in enumerate(step["operations"]):
                op_name = op if isinstance(op, str) else list(op.keys())
[0]

                all_ops_until_and_including_current = (
                    [op_map[prev_op] for prev_op in step["operations"]
[:idx]]
                    + [op_map[op_name]]
```

```
                    + [self.config.get("system_prompt", {})]
                )

                for op in all_ops_until_and_including_current:
                    if "model" not in op:
                        op["model"] = self.default_model

                all_ops_str =
json.dumps(all_ops_until_and_including_current)
                self.step_op_hashes[step["name"]][op_name] =
hashlib.sha256(
                    all_ops_str.encode()
                ).hexdigest()

    def get_output_path(self, require=False):
        output_path = self.config.get("pipeline", {}).get("output",
{}).get("path")
        if output_path:
            if not (
                output_path.lower().endswith(".json")
                or output_path.lower().endswith(".csv")
            ):
                raise ValueError(
                    f"Output path '{output_path}' is not a JSON or CSV
file. Please provide a path ending with '.json' or '.csv'."
                )
        elif require:
            raise ValueError(
                "No output path specified in the configuration. Please
provide an output path ending with '.json' or '.csv' in the configuration
to use the save() method."
            )

        return output_path

    def syntax_check(self):
        """
        Perform a syntax check on all operations defined in the
configuration.
        """
        self.console.log("[yellow]Checking operations...[/yellow]")

        # Just validate that it's a json file if specified
        self.get_output_path()
        current = self.last_op_container

        try:
            # Walk the last op container to check syntax
            op_containers = []
            if self.last_op_container:
                op_containers = [self.last_op_container]

            while op_containers:
                current = op_containers.pop(0)
                syntax_result = current.syntax_check()
                self.console.log(syntax_result, end="")
                # Add all children to the queue
                op_containers.extend(current.children)
        except Exception as e:
            raise ValueError(
                f"Syntax check failed for operation '{current.name}':
```

```
{str(e)}"
            )

        self.console.log("[green]✓ All operations passed syntax
check[/green]")

    def print_query_plan(self, 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.
        """
        if not self.last_op_container:
            self.console.log("\n[bold]Pipeline Steps:[/bold]")
            self.console.log(
                Panel("No operations in pipeline", title="Query Plan",
width=100)
            )
            self.console.log()
            return

        def _print_op(
            op: OpContainer, indent: int = 0, step_colors: dict[str, str]
| None = None
        ) -> str:
            # Handle boundary operations based on show_boundaries flag
            if isinstance(op, StepBoundary):
                if show_boundaries:
                    output = []
                    indent_str = "  " * indent
                    step_name = op.name.split("/")[0]
                    color = step_colors.get(step_name, "white")
                    output.append(
                        f"{indent_str}[{color}][bold]{op.name}[/bold]
[/{color}]"
                    )
                    output.append(f"{indent_str}Type: step_boundary")
                    if op.children:
                        output.append(f"{indent_str}[yellow]▼[/yellow]")
                        for child in op.children:
                            output.append(_print_op(child, indent + 1,
step_colors))
                    return "\n".join(output)
                elif op.children:
                    return _print_op(op.children[0], indent, step_colors)
                return ""

            # Build the string for the current operation with indentation
            indent_str = "  " * indent
            output = []

            # Color code the operation name based on its step
            step_name = op.name.split("/")[0]
            color = step_colors.get(step_name, "white")
            output.append(f"{indent_str}[{color}][bold]{op.name}[/bold]
[/{color}]")
            output.append(f"{indent_str}Type: {op.config['type']}")

            # Add schema if available
```

```
            if "output" in op.config and "schema" in op.config["output"]:
                output.append(f"{indent_str}Output Schema:")
                for field, field_type in op.config["output"]
["schema"].items():
                    escaped_type = escape(str(field_type))
                    output.append(
                        f"{indent_str}  {field}: [bright_white]
{escaped_type}[/bright_white]"
                    )

            # Add children
            if op.children:
                if op.is_equijoin:
                    output.append(f"{indent_str}[yellow]▼ LEFT[/yellow]")
                    output.append(_print_op(op.children[0], indent + 1,
step_colors))
                    output.append(f"{indent_str}[yellow]▼
RIGHT[/yellow]")
                    output.append(_print_op(op.children[1], indent + 1,
step_colors))
                else:
                    output.append(f"{indent_str}[yellow]▼[/yellow]")
                    for child in op.children:
                        output.append(_print_op(child, indent + 1,
step_colors))

            return "\n".join(output)

        # Get all step boundaries and extract unique step names
        step_boundaries = [
            op
            for name, op in self.op_container_map.items()
            if isinstance(op, StepBoundary)
        ]
        step_boundaries.sort(key=lambda x: x.name)

        # Create a color map for steps - using distinct colors
        colors = ["cyan", "magenta", "green", "yellow", "blue", "red"]
        step_names = [b.name.split("/")[0] for b in step_boundaries]
        step_colors = {
            name: colors[i % len(colors)] for i, name in
enumerate(step_names)
        }

        # Print the legend
        self.console.log("\n[bold]Pipeline Steps:[/bold]")
        for step_name, color in step_colors.items():
            self.console.log(f"[{color}]■[/{color}] {step_name}")

        # Print the full query plan starting from the last step boundary
        query_plan = _print_op(self.last_op_container,
step_colors=step_colors)
        self.console.log(Panel(query_plan, title="Query Plan",
width=100))
        self.console.log()

    def find_operation(self, op_name: str) -> dict:
        for operation_config in self.config["operations"]:
            if operation_config["name"] == op_name:
                return operation_config
        raise ValueError(f"Operation '{op_name}' not found in
```

```
configuration.")

    def load_run_save(self) -> float:
        """
        Execute the entire pipeline defined in the configuration.
        """
        output_path = self.get_output_path(require=True)

        # Print the query plan
        self.print_query_plan()

        start_time = time.time()

        if self.last_op_container:
            self.load()
            self.console.rule("[bold]Pipeline Execution[/bold]")
            output, _, _ = self.last_op_container.next()
            self.save(output)

        execution_time = time.time() - start_time

        # Print execution summary
        summary = (
            f"Cost: [green]${self.total_cost:.2f}[/green]\n"
            f"Time: {execution_time:.2f}s\n"
            + (
                f"Cache: [dim]{self.intermediate_dir}[/dim]\n"
                if self.intermediate_dir
                else ""
            )
            + f"Output: [dim]{output_path}[/dim]"
        )
        self.console.log(Panel(summary, title="Execution Summary"))

        return self.total_cost

    def load(self) -> None:
        """
        Load all datasets defined in the configuration.
        """
        datasets = {}
        self.console.rule("[bold]Loading Datasets[/bold]")

        for name, dataset_config in self.config["datasets"].items():
            if dataset_config["type"] == "file":
                datasets[name] = Dataset(
                    self,
                    "file",
                    dataset_config["path"],
                    source="local",
                    parsing=dataset_config.get("parsing", []),
                    user_defined_parsing_tool_map=self.parsing_tool_map,
                )
                self.console.log(
                    f"[green]✓[/green] Loaded dataset '{name}' from
{dataset_config['path']}"
                )
            elif dataset_config["type"] == "memory":
                datasets[name] = Dataset(
                    self,
                    "memory",
```

```
                    dataset_config["path"],
                    source="local",
                    parsing=dataset_config.get("parsing", []),
                    user_defined_parsing_tool_map=self.parsing_tool_map,
                )
                self.console.log(
                    f"[green]✓[/green] Loaded dataset '{name}' from in-memory data"
                )
            else:
                raise ValueError(f"Unsupported dataset type: {dataset_config['type']}")

        self.datasets = {
            name: (
                dataset
                if isinstance(dataset, Dataset)
                else Dataset(self, "memory", dataset)
            )
            for name, dataset in datasets.items()
        }
        self.console.log()

    def save(self, data: list[dict]) -> None:
        """
        Save the final output of the pipeline.
        """
        self.get_output_path(require=True)

        output_config = self.config["pipeline"]["output"]
        if output_config["type"] == "file":
            # Create the directory if it doesn't exist
            if os.path.dirname(output_config["path"]):
                os.makedirs(os.path.dirname(output_config["path"]), exist_ok=True)
            if output_config["path"].lower().endswith(".json"):
                with open(output_config["path"], "w") as file:
                    json.dump(data, file, indent=2)
            else:  # CSV
                import csv

                with open(output_config["path"], "w", newline="") as file:
                    writer = csv.DictWriter(file, fieldnames=data[0].keys())
                    limited_data = [
                        {k: d.get(k, None) for k in data[0].keys()} for d in data
                    ]
                    writer.writeheader()
                    writer.writerows(limited_data)
            self.console.log(
                f"[green]✓[/green] Saved to [dim]{output_config['path']}[/dim]\n"
            )
        else:
            raise ValueError(
                f"Unsupported output type: {output_config['type']}. Supported types: file"
            )

```

```python
    def _load_from_checkpoint_if_exists(
        self, step_name: str, operation_name: str
    ) -> list[dict] | None:
        if self.intermediate_dir is None or
self.config.get("bypass_cache", False):
            return None

        intermediate_config_path = os.path.join(
            self.intermediate_dir, ".docetl_intermediate_config.json"
        )

        if not os.path.exists(intermediate_config_path):
            return None

        # Make sure the step and op name is in the checkpoint config path
        if (
            step_name not in self.step_op_hashes
            or operation_name not in self.step_op_hashes[step_name]
        ):
            return None

        # See if the checkpoint config is the same as the current step op
hash
        with open(intermediate_config_path, "r") as f:
            intermediate_config = json.load(f)

        if (
            intermediate_config.get(step_name, {}).get(operation_name,
"")
            != self.step_op_hashes[step_name][operation_name]
        ):
            return None

        checkpoint_path = os.path.join(
            self.intermediate_dir, step_name, f"{operation_name}.json"
        )
        # check if checkpoint exists
        if os.path.exists(checkpoint_path):
            if f"{step_name}_{operation_name}" not in self.datasets:
                self.datasets[f"{step_name}_{operation_name}"] = Dataset(
                    self, "file", checkpoint_path, "local"
                )

                self.console.log(
                    f"[green]✓[/green] [italic]Loaded checkpoint for
operation '{operation_name}' in step '{step_name}' from {checkpoint_path}
[/italic]"
                )

                return self.datasets[f"
{step_name}_{operation_name}"].load()
        return None

    def clear_intermediate(self) -> None:
        """
        Clear the intermediate directory.
        """
        # Remove the intermediate directory
        if self.intermediate_dir:
            shutil.rmtree(self.intermediate_dir)
            return
```

```python

        raise ValueError("Intermediate directory not set. Cannot clear intermediate.")

    def _save_checkpoint(
        self, step_name: str, operation_name: str, data: list[dict]
    ) -> None:
        """
        Save a checkpoint of the current data after an operation.

        This method creates a JSON file containing the current state of the data
        after an operation has been executed. The checkpoint is saved in a directory
        structure that reflects the step and operation names.

        Args:
            step_name (str): The name of the current step in the pipeline.
            operation_name (str): The name of the operation that was just executed.
            data (list[dict]): The current state of the data to be checkpointed.

        Note:
            The checkpoint is saved only if a checkpoint directory has been specified
            when initializing the DSLRunner.
        """
        checkpoint_path = os.path.join(
            self.intermediate_dir, step_name, f"{operation_name}.json"
        )
        if os.path.dirname(checkpoint_path):
            os.makedirs(os.path.dirname(checkpoint_path), exist_ok=True)
        with open(checkpoint_path, "w") as f:
            json.dump(data, f)

        # Update the intermediate config file with the hash for this step/operation
        # so that future runs can validate and reuse this checkpoint.
        if self.intermediate_dir:
            intermediate_config_path = os.path.join(
                self.intermediate_dir, ".docetl_intermediate_config.json"
            )

            # Initialize or load existing intermediate configuration
            if os.path.exists(intermediate_config_path):
                try:
                    with open(intermediate_config_path, "r") as cfg_file:
                        intermediate_config: dict[str, dict[str, str]] = json.load(
                            cfg_file
                        )
                except json.JSONDecodeError:
                    # If the file is corrupted, start fresh to avoid crashes
                    intermediate_config = {}
            else:
                intermediate_config = {}

            # Ensure nested dict structure exists
```

```
            step_dict = intermediate_config.setdefault(step_name, {})

            # Write (or overwrite) the hash for the current operation
            step_dict[operation_name] = self.step_op_hashes[step_name]
[operation_name]

            # Persist the updated configuration
            with open(intermediate_config_path, "w") as cfg_file:
                json.dump(intermediate_config, cfg_file, indent=2)

        self.console.log(
            f"[green]✓ [italic]Intermediate saved for operation
'{operation_name}' in step '{step_name}' at {checkpoint_path}[/italic]
[/green]"
        )

    def should_optimize(
        self, step_name: str, op_name: str, **kwargs
    ) -> tuple[str, float, list[dict[str, Any]], list[dict[str, Any]]]:
        self.load()

        # Augment the kwargs with the runner's config if not already
provided
        kwargs["litellm_kwargs"] = self.config.get("optimizer_config",
{}).get(
            "litellm_kwargs", {}
        )
        kwargs["rewrite_agent_model"] =
self.config.get("optimizer_config", {}).get(
            "rewrite_agent_model", "gpt-4o"
        )
        kwargs["judge_agent_model"] = self.config.get("optimizer_config",
{}).get(
            "judge_agent_model", "gpt-4o-mini"
        )

        builder = Optimizer(self, **kwargs)
        self.optimizer = builder
        result = builder.should_optimize(step_name, op_name)
        return result

    def optimize(
        self,
        save: bool = False,
        return_pipeline: bool = True,
        **kwargs,
    ) -> tuple[dict | "DSLRunner", float]:

        if not self.last_op_container:
            raise ValueError("No operations in pipeline. Cannot
optimize.")

        self.load()

        # Augment the kwargs with the runner's config if not already
provided
        kwargs["litellm_kwargs"] = self.config.get("optimizer_config",
{}).get(
            "litellm_kwargs", {}
        )
        kwargs["rewrite_agent_model"] =
```

```
self.config.get("optimizer_config", {}).get(
            "rewrite_agent_model", "gpt-4o"
        )
        kwargs["judge_agent_model"] = self.config.get("optimizer_config",
{}).get(
            "judge_agent_model", "gpt-4o-mini"
        )

        save_path = kwargs.get("save_path", None)
        # Pop the save_path from kwargs
        kwargs.pop("save_path", None)

        builder = Optimizer(
            self,
            **kwargs,
        )
        self.optimizer = builder
        llm_api_cost = builder.optimize()
        operations_cost = self.total_cost
        self.total_cost += llm_api_cost

        # Log the cost of optimization
        self.console.log(
            f"[green italic]💰 Total cost: ${self.total_cost:.4f}[/green
italic]"
        )
        self.console.log(
            f"[green italic]  ├─ Operation execution cost:
${operations_cost:.4f}[/green italic]"
        )
        self.console.log(
            f"[green italic]  └─ Optimization cost: ${llm_api_cost:.4f}
[/green italic]"
        )

        if save:
            # If output path is provided, save the optimized config to
that path
            if kwargs.get("save_path"):
                save_path = kwargs["save_path"]
                if not os.path.isabs(save_path):
                    save_path = os.path.join(os.getcwd(), save_path)
                builder.save_optimized_config(save_path)
                self.optimized_config_path = save_path
            else:
                builder.save_optimized_config(f"
{self.base_name}_opt.yaml")
                self.optimized_config_path = f"{self.base_name}_opt.yaml"

        if return_pipeline:
            return (
                DSLRunner(builder.clean_optimized_config(),
self.max_threads),
                self.total_cost,
            )

        return builder.clean_optimized_config(), self.total_cost

    def _run_operation(
        self,
        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
`max_threads`	`int`	Maximum number of threads to use. Defaults to None.	`None`

Source code in `docetl/runner.py`

```
def __init__(self, config: dict, max_threads: int | None = None,
**kwargs):
    """
    Initialize the DSLRunner with a YAML configuration file.

    Args:
        max_threads (int, optional): Maximum number of threads to use.
Defaults to None.
    """
    super().__init__(
        config,
        base_name=kwargs.pop("base_name", None),
        yaml_file_suffix=kwargs.pop("yaml_file_suffix", None),
        max_threads=max_threads,
        **kwargs,
    )
    self.total_cost = 0
    self._initialize_state()
    self._setup_parsing_tools()
    self._build_operation_graph(config)
    self._compute_operation_hashes()

    # Run initial validation
    self._from_df_accessors = kwargs.get("from_df_accessors", False)
    if not self._from_df_accessors:
        self.syntax_check()
```

`clear_intermediate()`

Clear the intermediate directory.

Source code in `docetl/runner.py`

```
def clear_intermediate(self) -> None:
    """
    Clear the intermediate directory.
    """
    # Remove the intermediate directory
    if self.intermediate_dir:
        shutil.rmtree(self.intermediate_dir)
        return

    raise ValueError("Intermediate directory not set. Cannot clear
intermediate.")
```

`load()`

Load all datasets defined in the configuration.

Source code in `docetl/runner.py`

```
def load(self) -> None:
    """
    Load all datasets defined in the configuration.
    """
    datasets = {}
    self.console.rule("[bold]Loading Datasets[/bold]")

    for name, dataset_config in self.config["datasets"].items():
        if dataset_config["type"] == "file":
            datasets[name] = Dataset(
                self,
                "file",
                dataset_config["path"],
                source="local",
                parsing=dataset_config.get("parsing", []),
                user_defined_parsing_tool_map=self.parsing_tool_map,
            )
            self.console.log(
                f"[green]✓[/green] Loaded dataset '{name}' from
{dataset_config['path']}"
            )
        elif dataset_config["type"] == "memory":
            datasets[name] = Dataset(
                self,
                "memory",
                dataset_config["path"],
                source="local",
                parsing=dataset_config.get("parsing", []),
                user_defined_parsing_tool_map=self.parsing_tool_map,
            )
            self.console.log(
                f"[green]✓[/green] Loaded dataset '{name}' from in-memory
data"
            )
        else:
            raise ValueError(f"Unsupported dataset type:
{dataset_config['type']}")

    self.datasets = {
        name: (
            dataset
            if isinstance(dataset, Dataset)
            else Dataset(self, "memory", dataset)
        )
        for name, dataset in datasets.items()
    }
    self.console.log()
```

`load_run_save()`

Execute the entire pipeline defined in the configuration.

Source code in `docetl/runner.py`

```python
def load_run_save(self) -> float:
    """
    Execute the entire pipeline defined in the configuration.
    """
    output_path = self.get_output_path(require=True)

    # Print the query plan
    self.print_query_plan()

    start_time = time.time()

    if self.last_op_container:
        self.load()
        self.console.rule("[bold]Pipeline Execution[/bold]")
        output, _, _ = self.last_op_container.next()
        self.save(output)

    execution_time = time.time() - start_time

    # Print execution summary
    summary = (
        f"Cost: [green]${self.total_cost:.2f}[/green]\n"
        f"Time: {execution_time:.2f}s\n"
        + (
            f"Cache: [dim]{self.intermediate_dir}[/dim]\n"
            if self.intermediate_dir
            else ""
        )
        + f"Output: [dim]{output_path}[/dim]"
    )
    self.console.log(Panel(summary, title="Execution Summary"))

    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`

```
def print_query_plan(self, 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.
    """
    if not self.last_op_container:
        self.console.log("\n[bold]Pipeline Steps:[/bold]")
        self.console.log(
            Panel("No operations in pipeline", title="Query Plan",
width=100)
        )
        self.console.log()
        return

    def _print_op(
        op: OpContainer, indent: int = 0, step_colors: dict[str, str] |
None = None
    ) -> str:
        # Handle boundary operations based on show_boundaries flag
        if isinstance(op, StepBoundary):
            if show_boundaries:
                output = []
                indent_str = "  " * indent
                step_name = op.name.split("/")[0]
                color = step_colors.get(step_name, "white")
                output.append(
                    f"{indent_str}[{color}][bold]{op.name}[/bold]
[/{color}]"
                )
                output.append(f"{indent_str}Type: step_boundary")
                if op.children:
                    output.append(f"{indent_str}[yellow]▼[/yellow]")
                    for child in op.children:
                        output.append(_print_op(child, indent + 1,
step_colors))
                return "\n".join(output)
            elif op.children:
                return _print_op(op.children[0], indent, step_colors)
            return ""

        # Build the string for the current operation with indentation
        indent_str = "  " * indent
        output = []

        # Color code the operation name based on its step
        step_name = op.name.split("/")[0]
        color = step_colors.get(step_name, "white")
        output.append(f"{indent_str}[{color}][bold]{op.name}[/bold]
[/{color}]")
        output.append(f"{indent_str}Type: {op.config['type']}")

        # Add schema if available
        if "output" in op.config and "schema" in op.config["output"]:
            output.append(f"{indent_str}Output Schema:")
```

```
            for field, field_type in op.config["output"]
["schema"].items():
                escaped_type = escape(str(field_type))
                output.append(
                    f"{indent_str}  {field}: [bright_white]{escaped_type}
[/bright_white]"
                )

        # Add children
        if op.children:
            if op.is_equijoin:
                output.append(f"{indent_str}[yellow]▼ LEFT[/yellow]")
                output.append(_print_op(op.children[0], indent + 1,
step_colors))
                output.append(f"{indent_str}[yellow]▼ RIGHT[/yellow]")
                output.append(_print_op(op.children[1], indent + 1,
step_colors))
            else:
                output.append(f"{indent_str}[yellow]▼[/yellow]")
                for child in op.children:
                    output.append(_print_op(child, indent + 1,
step_colors))

        return "\n".join(output)

    # Get all step boundaries and extract unique step names
    step_boundaries = [
        op
        for name, op in self.op_container_map.items()
        if isinstance(op, StepBoundary)
    ]
    step_boundaries.sort(key=lambda x: x.name)

    # Create a color map for steps - using distinct colors
    colors = ["cyan", "magenta", "green", "yellow", "blue", "red"]
    step_names = [b.name.split("/")[0] for b in step_boundaries]
    step_colors = {
        name: colors[i % len(colors)] for i, name in
enumerate(step_names)
    }

    # Print the legend
    self.console.log("\n[bold]Pipeline Steps:[/bold]")
    for step_name, color in step_colors.items():
        self.console.log(f"[{color}]■[/{color}] {step_name}")

    # Print the full query plan starting from the last step boundary
    query_plan = _print_op(self.last_op_container,
step_colors=step_colors)
    self.console.log(Panel(query_plan, title="Query Plan", width=100))
    self.console.log()
```

`save(data)`

Save the final output of the pipeline.

Source code in `docetl/runner.py`

```
def save(self, data: list[dict]) -> None:
    """
    Save the final output of the pipeline.
    """
    self.get_output_path(require=True)

    output_config = self.config["pipeline"]["output"]
    if output_config["type"] == "file":
        # Create the directory if it doesn't exist
        if os.path.dirname(output_config["path"]):
            os.makedirs(os.path.dirname(output_config["path"]),
exist_ok=True)
        if output_config["path"].lower().endswith(".json"):
            with open(output_config["path"], "w") as file:
                json.dump(data, file, indent=2)
        else:  # CSV
            import csv

            with open(output_config["path"], "w", newline="") as file:
                writer = csv.DictWriter(file, fieldnames=data[0].keys())
                limited_data = [
                    {k: d.get(k, None) for k in data[0].keys()} for d in
data
                ]
                writer.writeheader()
                writer.writerows(limited_data)
        self.console.log(
            f"[green]✓[/green] Saved to [dim]{output_config['path']}
[/dim]\n"
        )
    else:
        raise ValueError(
            f"Unsupported output type: {output_config['type']}. Supported
types: file"
        )
```

`syntax_check()`

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

Source code in `docetl/runner.py`

```python
def syntax_check(self):
    """
    Perform a syntax check on all operations defined in the configuration.
    """
    self.console.log("[yellow]Checking operations...[/yellow]")

    # Just validate that it's a json file if specified
    self.get_output_path()
    current = self.last_op_container

    try:
        # Walk the last op container to check syntax
        op_containers = []
        if self.last_op_container:
            op_containers = [self.last_op_container]

        while op_containers:
            current = op_containers.pop(0)
            syntax_result = current.syntax_check()
            self.console.log(syntax_result, end="")
            # Add all children to the queue
            op_containers.extend(current.children)
    except Exception as e:
        raise ValueError(
            f"Syntax check failed for operation '{current.name}': {str(e)}"
        )

    self.console.log("[green]✓ All operations passed syntax 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`

```
class 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.
    """

    def __init__(
        self,
        runner: "DSLRunner",
        rewrite_agent_model: str = "gpt-4o",
        judge_agent_model: str = "gpt-4o-mini",
        litellm_kwargs: dict[str, Any] = {},
        resume: bool = False,
        timeout: int = 60,
    ):
        """
        Initialize the optimizer with a runner instance and
configuration.
        Sets up optimization parameters, caching, and cost tracking.

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

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

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

```
        """
        self.config = runner.config
        self.console = runner.console
        self.max_threads = runner.max_threads

        self.base_name = runner.base_name
        self.yaml_file_suffix = runner.yaml_file_suffix
        self.runner = runner
        self.status = runner.status

        self.optimized_config = copy.deepcopy(self.config)

        # Get the rate limits from the optimizer config
        rate_limits = self.config.get("optimizer_config",
{}).get("rate_limits", {})

        self.llm_client = LLMClient(
            runner,
            rewrite_agent_model,
            judge_agent_model,
            rate_limits,
            **litellm_kwargs,
        )
        self.timeout = timeout
        self.resume = resume
        self.captured_output = CapturedOutput()

        # Add sample cache for build operations
        self.sample_cache = {}  # Maps operation names to (output_data,
sample_size)

        home_dir = os.environ.get("DOCETL_HOME_DIR",
os.path.expanduser("~"))
        cache_dir = os.path.join(home_dir,
f".docetl/cache/{runner.yaml_file_suffix}")
        os.makedirs(cache_dir, exist_ok=True)

        # Hash the config to create a unique identifier
        config_hash =
hashlib.sha256(str(self.config).encode()).hexdigest()
        self.optimized_ops_path = f"{cache_dir}/{config_hash}.yaml"

        # Update sample size map
        self.sample_size_map = SAMPLE_SIZE_MAP
        if self.config.get("optimizer_config", {}).get("sample_sizes",
{}):
            self.sample_size_map.update(self.config["optimizer_config"]
["sample_sizes"])

        if not self.runner._from_df_accessors:
            self.print_optimizer_config()

    def print_optimizer_config(self):
        """
        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.
        """
        self.console.log(
            Panel.fit(
                "[bold cyan]Optimizer Configuration[/bold cyan]\n"
                f"[yellow]Sample Size:[/yellow] {self.sample_size_map}\n"
                f"[yellow]Max Threads:[/yellow] {self.max_threads}\n"
                f"[yellow]Rewrite Agent Model:[/yellow]
{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",
            )
        )

    def _insert_empty_resolve_operations(self):
        """
        Determines whether to insert resolve operations in the pipeline.

        For each reduce operation in the tree, checks if it has any map
operation as a descendant
        without a resolve operation in between. If found, inserts an
empty resolve operation
        right after the reduce operation.

        The method modifies the operation container tree in-place.

        Returns:
            None
        """
        if not self.runner.last_op_container:
            return

        def find_map_without_resolve(container, visited=None):
            """Helper to find first map descendant without a resolve
operation in between."""
            if visited is None:
                visited = set()

            if container.name in visited:
                return None
            visited.add(container.name)

            if not container.children:
                return None

            for child in container.children:
                if child.config["type"] == "map":
                    return child
                if child.config["type"] == "resolve":
                    continue
                map_desc = find_map_without_resolve(child, visited)
                if map_desc:
```

```
                    return map_desc
            return None

        # Walk down the operation container tree
        containers_to_check = [self.runner.last_op_container]
        while containers_to_check:
            current = containers_to_check.pop(0)

            # Skip if this is a boundary or has no children
            if isinstance(current, StepBoundary) or not current.children:
                containers_to_check.extend(current.children)
                continue

            # Get the step name from the container's name
            step_name = current.name.split("/")[0]

            # Check if current container is a reduce operation
            if current.config["type"] == "reduce" and current.config.get(
                "synthesize_resolve", True
            ):
                reduce_key = current.config.get("reduce_key", "_all")
                if isinstance(reduce_key, str):
                    reduce_key = [reduce_key]

                if "_all" not in reduce_key:
                    # Find map descendant without resolve
                    map_desc = find_map_without_resolve(current)
                    if map_desc:
                        # Synthesize an empty resolver
                        self.console.log(
                            "[yellow]Synthesizing empty resolver
operation:[/yellow]"
                        )
                        self.console.log(
                            f"  • [cyan]Reduce operation:[/cyan] [bold]
{current.name}[/bold]"
                        )
                        self.console.log(
                            f"  • [cyan]Step:[/cyan] [bold]{step_name}
[/bold]"
                        )

                        # Create new resolve operation config
                        new_resolve_name = (

f"synthesized_resolve_{len(self.config['operations'])}"
                        )
                        new_resolve_config = {
                            "name": new_resolve_name,
                            "type": "resolve",
                            "empty": True,
                            "optimize": True,
                            "embedding_model": "text-embedding-3-small",
                            "resolution_model": self.config.get(
                                "default_model", "gpt-4o-mini"
                            ),
                            "comparison_model": self.config.get(
                                "default_model", "gpt-4o-mini"
                            ),
                            "_intermediates": {
                                "map_prompt":
```

```
map_desc.config.get("prompt"),
                                "reduce_key": reduce_key,
                            },
                        }

                        # Add to operations list

self.config["operations"].append(new_resolve_config)

                        # Create new resolve container
                        new_resolve_container = OpContainer(
                            f"{step_name}/{new_resolve_name}",
                            self.runner,
                            new_resolve_config,
                        )

                        # Insert the new container between reduce and its
children
                        new_resolve_container.children = current.children
                        for child in new_resolve_container.children:
                            child.parent = new_resolve_container
                        current.children = [new_resolve_container]
                        new_resolve_container.parent = current

                        # Add to container map
                        self.runner.op_container_map[
                            f"{step_name}/{new_resolve_name}"
                        ] = new_resolve_container

                        # Add children to the queue

containers_to_check.extend(new_resolve_container.children)

    def _add_map_prompts_to_reduce_operations(self):
        """
        Add relevant map prompts to reduce operations based on their
reduce keys.

        This method walks the operation container tree to find map
operations and their
        output schemas, then associates those with reduce operations that
use those keys.
        When a reduce operation is found, it looks through its
descendants to find the
        relevant map operations and adds their prompts.

        The method modifies the operation container tree in-place.
        """
        if not self.runner.last_op_container:
            return

        def find_map_prompts_for_keys(container, keys, visited=None):
            """Helper to find map prompts for given keys in the
container's descendants."""
            if visited is None:
                visited = set()

            if container.name in visited:
                return []
            visited.add(container.name)

```

```
            prompts = []
            if container.config["type"] == "map":
                output_schema = container.config.get("output",
{}).get("schema", {})
                if any(key in output_schema for key in keys):
                    prompts.append(container.config.get("prompt", ""))

            for child in container.children:
                prompts.extend(find_map_prompts_for_keys(child, keys,
visited))

            return prompts

        # Walk down the operation container tree
        containers_to_check = [self.runner.last_op_container]
        while containers_to_check:
            current = containers_to_check.pop(0)

            # Skip if this is a boundary or has no children
            if isinstance(current, StepBoundary) or not current.children:
                containers_to_check.extend(current.children)
                continue

            # If this is a reduce operation, find relevant map prompts
            if current.config["type"] == "reduce":
                reduce_keys = current.config.get("reduce_key", [])
                if isinstance(reduce_keys, str):
                    reduce_keys = [reduce_keys]

                # Find map prompts in descendants
                relevant_prompts = find_map_prompts_for_keys(current,
reduce_keys)

                if relevant_prompts:
                    current.config["_intermediates"] =
current.config.get(
                        "_intermediates", {}
                    )
                    current.config["_intermediates"]["last_map_prompt"] =
(
                        relevant_prompts[-1]
                    )

            # Add children to the queue
            containers_to_check.extend(current.children)

    def should_optimize(
        self, step_name: str, op_name: str
    ) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]], float]:
        """
        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.
        """
        self.console.rule("[bold cyan]Beginning Pipeline Assessment[/bold
cyan]")

        self._insert_empty_resolve_operations()

        node_of_interest = self.runner.op_container_map[f"
```

```
{step_name}/{op_name}"]

        # Run the node_of_interest's children
        input_data = []
        for child in node_of_interest.children:
            input_data.append(
                child.next(
                    is_build=True,

sample_size_needed=SAMPLE_SIZE_MAP.get(child.config["type"]),
                )[0]
            )

        # Set the step
        self.captured_output.set_step(step_name)

        # Determine whether we should optimize the node_of_interest
        if (
            node_of_interest.config.get("type") == "map"
            or node_of_interest.config.get("type") == "filter"
        ):
            # Create instance of map optimizer
            map_optimizer = MapOptimizer(
                self.runner,
                self.runner._run_operation,
                is_filter=node_of_interest.config.get("type") ==
"filter",
            )
            should_optimize_output, input_data, output_data = (
                map_optimizer.should_optimize(node_of_interest.config,
input_data[0])
            )
        elif node_of_interest.config.get("type") == "reduce":
            reduce_optimizer = ReduceOptimizer(
                self.runner,
                self.runner._run_operation,
            )
            should_optimize_output, input_data, output_data = (
                reduce_optimizer.should_optimize(node_of_interest.config,
input_data[0])
            )
        elif node_of_interest.config.get("type") == "resolve":
            resolve_optimizer = JoinOptimizer(
                self.runner,
                node_of_interest.config,
                target_recall=self.config.get("optimizer_config", {})
                .get("resolve", {})
                .get("target_recall", 0.95),
            )
            _, should_optimize_output =
resolve_optimizer.should_optimize(input_data[0])

            # if should_optimize_output is empty, then we should move to
the reduce operation
            if should_optimize_output == "":
                return "", [], [], 0.0
        else:
            return "", [], [], 0.0

        # Return the string and operation cost
        return (
```

```
            should_optimize_output,
            input_data,
            output_data,
            self.runner.total_cost + self.llm_client.total_cost,
        )

    def optimize(self) -> float:
        """
        Optimizes the entire pipeline by walking the operation DAG and applying
        operation-specific optimizers where marked. Returns the total optimization cost.
        """
        self.console.rule("[bold cyan]Beginning Pipeline Rewrites[/bold cyan]")

        # If self.resume is True and there's a checkpoint, load it
        if self.resume:
            if os.path.exists(self.optimized_ops_path):
                # Load the yaml and change the runner with it
                with open(self.optimized_ops_path, "r") as f:
                    partial_optimized_config = yaml.safe_load(f)
                    self.console.log(
                        "[yellow]Loading partially optimized pipeline from checkpoint...[/yellow]"
                    )

self.runner._build_operation_graph(partial_optimized_config)
            else:
                self.console.log(
                    "[yellow]No checkpoint found, starting optimization from scratch...[/yellow]"
                )

        else:
            self._insert_empty_resolve_operations()

        # Start with the last operation container and visit each child
        self.runner.last_op_container.optimize()

        flush_cache(self.console)

        # Print the query plan
        self.console.rule("[bold cyan]Optimized Query Plan[/bold cyan]")
        self.runner.print_query_plan()

        return self.llm_client.total_cost

    def _optimize_equijoin(
        self,
        op_config: dict[str, Any],
        left_name: str,
        right_name: str,
        left_data: list[dict[str, Any]],
        right_data: list[dict[str, Any]],
        run_operation: Callable[
            [dict[str, Any], list[dict[str, Any]]], list[dict[str, Any]]
        ],
    ) -> tuple[list[dict[str, Any]], dict[str, list[dict[str, Any]]], str, str]:
        """
```

```
        Optimizes an equijoin operation by analyzing join conditions and
potentially inserting
        map operations to improve join efficiency. Returns the optimized
configuration and updated data.
        """
        max_iterations = 2
        new_left_name = left_name
        new_right_name = right_name
        new_steps = []
        for _ in range(max_iterations):
            join_optimizer = JoinOptimizer(
                self.runner,
                op_config,
                target_recall=self.runner.config.get("optimizer_config",
{})
                .get("equijoin", {})
                .get("target_recall", 0.95),

estimated_selectivity=self.runner.config.get("optimizer_config", {})
                .get("equijoin", {})
                .get("estimated_selectivity", None),
            )
            optimized_config, cost, agent_results =
join_optimizer.optimize_equijoin(
                left_data, right_data
            )
            self.runner.total_cost += cost
            # Update the operation config with the optimized values
            op_config.update(optimized_config)

            if not agent_results.get("optimize_map", False):
                break  # Exit the loop if no more map optimizations are
necessary

            # Update the status to indicate we're optimizing a map
operation
            output_key = agent_results["output_key"]
            if self.runner.status:
                self.runner.status.update(
                    f"Optimizing map operation for {output_key}
extraction to help with the equijoin"
                )
            map_prompt = agent_results["map_prompt"]
            dataset_to_transform = (
                left_data
                if agent_results["dataset_to_transform"] == "left"
                else right_data
            )

            # Create a new step for the map operation
            map_operation = {
                "name": f"synthesized_{output_key}_extraction",
                "type": "map",
                "prompt": map_prompt,
                "model": self.config.get("default_model", "gpt-4o-mini"),
                "output": {"schema": {output_key: "string"}},
                "optimize": False,
            }

            # Optimize the map operation
            if map_operation["optimize"]:
```

```
                dataset_to_transform_sample = (
                    random.sample(dataset_to_transform,
self.sample_size_map.get("map"))
                    if self.config.get("optimizer_config", {}).get(
                        "random_sample", False
                    )
                    else dataset_to_transform[:
self.sample_size_map.get("map")]
                )
                optimized_map_operations = self._optimize_map(
                    map_operation, dataset_to_transform_sample
                )
            else:
                optimized_map_operations = [map_operation]

            new_step = {
                "name": f"synthesized_{output_key}_extraction",
                "input": (
                    left_name
                    if agent_results["dataset_to_transform"] == "left"
                    else right_name
                ),
                "operations": [mo["name"] for mo in
optimized_map_operations],
            }
            if agent_results["dataset_to_transform"] == "left":
                new_left_name = new_step["name"]
            else:
                new_right_name = new_step["name"]

            new_steps.append((new_step["name"], new_step,
optimized_map_operations))

            # Now run the optimized map operation on the entire
dataset_to_transform
            for op in optimized_map_operations:
                dataset_to_transform = run_operation(op,
dataset_to_transform)

            # Update the appropriate dataset for the next iteration
            if agent_results["dataset_to_transform"] == "left":
                left_data = dataset_to_transform
            else:
                right_data = dataset_to_transform

            if self.runner.status:
                self.runner.status.update(
                    f"Optimizing equijoin operation with {output_key}
extraction"
                )

        return op_config, new_steps, new_left_name, new_right_name

    def checkpoint_optimized_ops(self) -> None:
        """
        Generates the clean config and saves it to the
self.optimized_ops_path
        This is used to resume optimization from a previous run
        """
        clean_config = self.clean_optimized_config()
        with open(self.optimized_ops_path, "w") as f:
```

```
            yaml.safe_dump(clean_config, f, default_flow_style=False,
width=80)

    # Recursively resolve all anchors and aliases
    @staticmethod
    def resolve_anchors(data):
        """
        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.

        Args:
            data: The data structure to resolve. Can be a dictionary,
list, or any other type.

        Returns:
            The resolved data structure with all anchors and aliases
replaced by their actual values.
        """
        if isinstance(data, dict):
            return {k: Optimizer.resolve_anchors(v) for k, v in
data.items()}
        elif isinstance(data, list):
            return [Optimizer.resolve_anchors(item) for item in data]
        else:
            return data

    def clean_optimized_config(self) -> dict:
        """
        Creates a clean YAML configuration from the optimized operation
containers,
        removing internal fields and organizing operations into proper
pipeline steps.
        """
        if not self.runner.last_op_container:
            return self.config

        # Create a clean copy of the config
        datasets = {}
        for dataset_name, dataset_config in self.config.get("datasets",
{}).items():
            if dataset_config["type"] == "memory":
                dataset_config_copy = copy.deepcopy(dataset_config)
                dataset_config_copy["path"] = "in-memory data"
                datasets[dataset_name] = dataset_config_copy
            else:
                datasets[dataset_name] = dataset_config

        clean_config = {
            "datasets": datasets,
            "operations": [],
            "pipeline": self.runner.config.get(
                "pipeline", {}
            ).copy(),  # Copy entire pipeline config
        }

        # Reset steps to regenerate
        clean_config["pipeline"]["steps"] = []

```

```python
        # Keep track of operations we've seen to avoid duplicates
        seen_operations = set()

        def clean_operation(op_container: OpContainer) -> dict:
            """Remove internal fields from operation config"""
            op_config = op_container.config
            clean_op = copy.deepcopy(op_config)

            clean_op.pop("_intermediates", None)

            # If op has already been optimized, remove the recursively_optimize and optimize fields
            if op_container.is_optimized:
                for field in ["recursively_optimize", "optimize"]:
                    clean_op.pop(field, None)

            return clean_op

        def process_container(container, current_step=None):
            """Process an operation container and its dependencies"""
            # Skip step boundaries
            if isinstance(container, StepBoundary):
                if container.children:
                    return process_container(container.children[0], current_step)
                return None, None

            # Get step name from container name
            step_name = container.name.split("/")[0]

            # If this is a new step, create it
            if not current_step or current_step["name"] != step_name:
                current_step = {"name": step_name, "operations": []}
                clean_config["pipeline"]["steps"].insert(0, current_step)

            # Skip scan operations but process their dependencies
            if container.config["type"] == "scan":
                if container.children:
                    return process_container(container.children[0], current_step)
                return None, current_step

            # Handle equijoin operations
            if container.is_equijoin:
                # 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 with left and right inputs
                current_step["operations"].insert(
                    0,
                    {
                        container.config["name"]: {
                            "left": container.kwargs["left_name"],
                            "right": container.kwargs["right_name"],
                        }
                    },
                )
```

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

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

Source code in `docetl/optimizer.py`

```
def __init__(
    self,
    runner: "DSLRunner",
    rewrite_agent_model: str = "gpt-4o",
    judge_agent_model: str = "gpt-4o-mini",
    litellm_kwargs: dict[str, Any] = {},
    resume: bool = False,
    timeout: int = 60,
):
    """
    Initialize the optimizer with a runner instance and configuration.
    Sets up optimization parameters, caching, and cost tracking.

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

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

    The method also calls print_optimizer_config() to display the initial
configuration.
    """
    self.config = runner.config
    self.console = runner.console
    self.max_threads = runner.max_threads

    self.base_name = runner.base_name
    self.yaml_file_suffix = runner.yaml_file_suffix
    self.runner = runner
    self.status = runner.status

    self.optimized_config = copy.deepcopy(self.config)
```

```python

    # Get the rate limits from the optimizer config
    rate_limits = self.config.get("optimizer_config",
{}).get("rate_limits", {})

    self.llm_client = LLMClient(
        runner,
        rewrite_agent_model,
        judge_agent_model,
        rate_limits,
        **litellm_kwargs,
    )
    self.timeout = timeout
    self.resume = resume
    self.captured_output = CapturedOutput()

    # Add sample cache for build operations
    self.sample_cache = {}  # Maps operation names to (output_data,
sample_size)

    home_dir = os.environ.get("DOCETL_HOME_DIR", os.path.expanduser("~"))
    cache_dir = os.path.join(home_dir,
f".docetl/cache/{runner.yaml_file_suffix}")
    os.makedirs(cache_dir, exist_ok=True)

    # Hash the config to create a unique identifier
    config_hash = hashlib.sha256(str(self.config).encode()).hexdigest()
    self.optimized_ops_path = f"{cache_dir}/{config_hash}.yaml"

    # Update sample size map
    self.sample_size_map = SAMPLE_SIZE_MAP
    if self.config.get("optimizer_config", {}).get("sample_sizes", {}):
        self.sample_size_map.update(self.config["optimizer_config"]
["sample_sizes"])

    if not self.runner._from_df_accessors:
        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`

```python
def checkpoint_optimized_ops(self) -> None:
    """
    Generates the clean config and saves it to the
self.optimized_ops_path
    This is used to resume optimization from a previous run
    """
    clean_config = self.clean_optimized_config()
    with open(self.optimized_ops_path, "w") as f:
        yaml.safe_dump(clean_config, f, default_flow_style=False,
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`

```
def clean_optimized_config(self) -> dict:
    """
    Creates a clean YAML configuration from the optimized operation
containers,
    removing internal fields and organizing operations into proper
pipeline steps.
    """
    if not self.runner.last_op_container:
        return self.config

    # Create a clean copy of the config
    datasets = {}
    for dataset_name, dataset_config in self.config.get("datasets",
{}).items():
        if dataset_config["type"] == "memory":
            dataset_config_copy = copy.deepcopy(dataset_config)
            dataset_config_copy["path"] = "in-memory data"
            datasets[dataset_name] = dataset_config_copy
        else:
            datasets[dataset_name] = dataset_config

    clean_config = {
        "datasets": datasets,
        "operations": [],
        "pipeline": self.runner.config.get(
            "pipeline", {}
        ).copy(),  # Copy entire pipeline config
    }

    # Reset steps to regenerate
    clean_config["pipeline"]["steps"] = []

    # Keep track of operations we've seen to avoid duplicates
    seen_operations = set()

    def clean_operation(op_container: OpContainer) -> dict:
        """Remove internal fields from operation config"""
        op_config = op_container.config
        clean_op = copy.deepcopy(op_config)

        clean_op.pop("_intermediates", None)

        # If op has already been optimized, remove the
recursively_optimize and optimize fields
        if op_container.is_optimized:
            for field in ["recursively_optimize", "optimize"]:
                clean_op.pop(field, None)

        return clean_op

    def process_container(container, current_step=None):
        """Process an operation container and its dependencies"""
        # Skip step boundaries
        if isinstance(container, StepBoundary):
            if container.children:
                return process_container(container.children[0],
current_step)
```

```
            return None, None

        # Get step name from container name
        step_name = container.name.split("/")[0]

        # If this is a new step, create it
        if not current_step or current_step["name"] != step_name:
            current_step = {"name": step_name, "operations": []}
            clean_config["pipeline"]["steps"].insert(0, current_step)

        # Skip scan operations but process their dependencies
        if container.config["type"] == "scan":
            if container.children:
                return process_container(container.children[0],
current_step)
            return None, current_step

        # Handle equijoin operations
        if container.is_equijoin:
            # 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 with left and right inputs
            current_step["operations"].insert(
                0,
                {
                    container.config["name"]: {
                        "left": container.kwargs["left_name"],
                        "right": container.kwargs["right_name"],
                    }
                },
            )

            # 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
```

`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`

```
def optimize(self) -> float:
    """
    Optimizes the entire pipeline by walking the operation DAG and applying
    operation-specific optimizers where marked. Returns the total optimization cost.
    """
    self.console.rule("[bold cyan]Beginning Pipeline Rewrites[/bold cyan]")

    # If self.resume is True and there's a checkpoint, load it
    if self.resume:
        if os.path.exists(self.optimized_ops_path):
            # Load the yaml and change the runner with it
            with open(self.optimized_ops_path, "r") as f:
                partial_optimized_config = yaml.safe_load(f)
                self.console.log(
                    "[yellow]Loading partially optimized pipeline from checkpoint...[/yellow]"
                )

self.runner._build_operation_graph(partial_optimized_config)
        else:
            self.console.log(
                "[yellow]No checkpoint found, starting optimization from scratch...[/yellow]"
            )

    else:
        self._insert_empty_resolve_operations()

    # Start with the last operation container and visit each child
    self.runner.last_op_container.optimize()

    flush_cache(self.console)

    # Print the query plan
    self.console.rule("[bold cyan]Optimized Query Plan[/bold cyan]")
    self.runner.print_query_plan()

    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`

```
def print_optimizer_config(self):
    """
    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.
    """
    self.console.log(
        Panel.fit(
            "[bold cyan]Optimizer Configuration[/bold cyan]\n"
            f"[yellow]Sample Size:[/yellow] {self.sample_size_map}\n"
            f"[yellow]Max Threads:[/yellow] {self.max_threads}\n"
            f"[yellow]Rewrite Agent Model:[/yellow] {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
`data`		The data structure to resolve. Can be a dictionary, list, or any other type.	*required*

Returns:

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

Source code in `docetl/optimizer.py`

```
@staticmethod
def resolve_anchors(data):
    """
    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.

    Args:
        data: The data structure to resolve. Can be a dictionary, list,
or any other type.

    Returns:
        The resolved data structure with all anchors and aliases replaced
by their actual values.
    """
    if isinstance(data, dict):
        return {k: Optimizer.resolve_anchors(v) for k, v in data.items()}
    elif isinstance(data, list):
        return [Optimizer.resolve_anchors(item) for item in data]
    else:
        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`

```
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
{optimized_config_path}[/green italic]"
        )
```

`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`

```
def should_optimize(
    self, step_name: str, op_name: str
) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]], float]:
    """
    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.
    """
    self.console.rule("[bold cyan]Beginning Pipeline Assessment[/bold
cyan]")

    self._insert_empty_resolve_operations()

    node_of_interest = self.runner.op_container_map[f"
{step_name}/{op_name}"]

    # Run the node_of_interest's children
    input_data = []
    for child in node_of_interest.children:
        input_data.append(
            child.next(
                is_build=True,

sample_size_needed=SAMPLE_SIZE_MAP.get(child.config["type"]),
            )[0]
        )

    # Set the step
    self.captured_output.set_step(step_name)

    # Determine whether we should optimize the node_of_interest
    if (
        node_of_interest.config.get("type") == "map"
        or node_of_interest.config.get("type") == "filter"
    ):
        # Create instance of map optimizer
        map_optimizer = MapOptimizer(
            self.runner,
            self.runner._run_operation,
            is_filter=node_of_interest.config.get("type") == "filter",
        )
        should_optimize_output, input_data, output_data = (
            map_optimizer.should_optimize(node_of_interest.config,
input_data[0])
        )
    elif node_of_interest.config.get("type") == "reduce":
        reduce_optimizer = ReduceOptimizer(
            self.runner,
            self.runner._run_operation,
        )
        should_optimize_output, input_data, output_data = (
            reduce_optimizer.should_optimize(node_of_interest.config,
input_data[0])
        )
    elif node_of_interest.config.get("type") == "resolve":
        resolve_optimizer = JoinOptimizer(
```

```
            self.runner,
            node_of_interest.config,
            target_recall=self.config.get("optimizer_config", {})
            .get("resolve", {})
            .get("target_recall", 0.95),
        )
        _, should_optimize_output =
resolve_optimizer.should_optimize(input_data[0])

        # if should_optimize_output is empty, then we should move to the
reduce operation
        if should_optimize_output == "":
            return "", [], [], 0.0
    else:
        return "", [], [], 0.0

    # Return the string and operation cost
    return (
        should_optimize_output,
        input_data,
        output_data,
        self.runner.total_cost + self.llm_client.total_cost,
    )
```