Running the Optimizer

Optimizer Stability

The optimization process can be unstable, as well as resource-intensive (we've seen it take up to 10 minutes to optimize a single operation, spending up to ~$50 in API costs for end-to-end pipelines). We recommend optimizing one operation at a time and retrying if necessary, as results may vary between runs. This approach also allows you to confidently verify that each optimized operation is performing as expected before moving on to the next.

See the API for more details on how to resume the optimizer from a failed run, by rerunning `docetl build pipeline.yaml --resume` (with the `--resume` flag).

Also, you can use gpt-4o-mini for cheaper optimizations (rather than the default gpt-4o), which you can do via `docetl build pipeline.yaml --model=gpt-4o-mini`.

To optimize your pipeline, start with your initial configuration and follow these steps:

1. Set `optimize: True` for the operation you want to optimize (start with the first operation, if you're not sure which one).
2. Run the optimizer using the command `docetl build pipeline.yaml`. This will generate an optimized version in `pipeline_opt.yaml`.
3. Review the optimized operation in `pipeline_opt.yaml`. If you're satisfied with the changes, copy the optimized operation back into your original `pipeline.yaml`.
4. Move on to the next LLM-powered operation and repeat steps 1-3.
5. Once all operations are optimized, your `pipeline.yaml` will contain the fully optimized pipeline.

When optimizing a resolve operation, the optimizer will also set blocking configurations and thresholds, saving you from manual configuration.

Feeling Ambitious?

You can run the optimizer on your entire pipeline by setting `optimize: True` for each operation you want to optimize. But sometimes the agent fails to find a better plan, and you'll need to manually intervene. We are exploring human-in-the-loop optimization, where the optimizer can ask for human feedback to improve its plans.

Example: Optimizing a Medical Transcripts Pipeline

Let's walk through optimizing a pipeline for extracting medication information from medical transcripts. We'll start with an initial pipeline and optimize it step by step.

Initial Pipeline

```yaml
datasets:
  transcripts:
    path: medical_transcripts.json
    type: file

default_model: gpt-4o-mini

operations:
  - name: extract_medications
    type: map
    optimize: true
    output:
      schema:
        medication: list[str]
    prompt: |
      Analyze the transcript: {{ input.src }}
      List all medications mentioned.

  - name: unnest_medications
    type: unnest
    unnest_key: medication

  - name: summarize_prescriptions
    type: reduce
    optimize: true
    reduce_key:
      - medication
    output:
      schema:
        side_effects: str
        uses: str
    prompt: |
      Summarize side effects and uses of {{ reduce_key }} from:
      {% for value in inputs %}
      Transcript {{ loop.index }}: {{ value.src }}
      {% endfor %}

pipeline:
  output:
    path: medication_summaries.json
    type: file
  steps:
    - input: transcripts
      name: medical_info_extraction
      operations:
        - extract_medications
```

```yaml
          - unnest_medications
          - summarize_prescriptions
```

Optimization Steps

First, we'll optimize the `extract_medications` operation. Set `optimize: True` for this operation and run the optimizer. Review the changes and integrate them into your pipeline.

Then, optimize the `summarize_prescriptions` operation by setting `optimize: True` and running `docetl build pipeline.yaml` again. The optimizer may suggest adding a resolve operation at this point, and will automatically configure blocking and thresholds. After completing all steps, your optimized pipeline might look like this:

Optimized Pipeline

```yaml
datasets:
  transcripts:
    path: medical_transcripts.json
    type: file

default_model: gpt-4o-mini

operations:
  - name: extract_medications
    type: map
    output:
      schema:
        medication: list[str]
    prompt: |
      Analyze the transcript: {{ input.src }}
      List all medications mentioned.
    gleaning:
      num_rounds: 1
      validation_prompt: |
        Evaluate the extraction for completeness and accuracy:
        1. Are all medications, dosages, and symptoms from the transcript
included?
        2. Is the extracted information correct and relevant?

  - name: unnest_medications
    type: unnest
    unnest_key: medication

  - name: resolve_medications
    type: resolve
    blocking_keys:
      - medication
    blocking_threshold: 0.7
    comparison_prompt: |
      Compare medications:
      1: {{ input1.medication }}
```

```yaml
        2: {{ input2.medication }}
        Are these the same or closely related?
      resolution_prompt: |
        Standardize the name for:
        {% for entry in inputs %}
        - {{ entry.medication }}
        {% endfor %}

    - name: summarize_prescriptions
      type: reduce
      reduce_key:
        - medication
      output:
        schema:
          side_effects: str
          uses: str
      prompt: |
        Summarize side effects and uses of {{ reduce_key }} from:
        {% for value in inputs %}
        Transcript {{ loop.index }}: {{ value.src }}
        {% endfor %}
      fold_batch_size: 10
      fold_prompt: |
        Update the existing summary of side effects and uses for {{ reduce_key
}} based on the following additional transcripts:
        {% for value in inputs %}
        Transcript {{ loop.index }}: {{ value.src }}
        {% endfor %}

        Existing summary:
        Side effects: {{ output.side_effects }}
        Uses: {{ output.uses }}

        Provide an updated and comprehensive summary, incorporating both the
existing information and any new insights from the additional transcripts.

pipeline:
  output:
    path: medication_summaries.json
    type: file
  steps:
    - input: transcripts
      name: medical_info_extraction
      operations:
        - extract_medications
        - unnest_medications
        - resolve_medications
        - summarize_prescriptions
```

This optimized pipeline now includes improved prompts, a resolve operation, and additional output fields for more comprehensive medication information extraction.

Feedback Welcome

We're continually improving the optimizer. Your feedback on its performance and usability is invaluable. Please share your experiences and suggestions!

Optimizer API

`docetl.cli.build(yaml_file=typer.Argument(..., help='Path to the YAML file containing the pipeline configuration'), max_threads=typer.Option(None, help='Maximum number of threads to use for running operations'), resume=typer.Option(False, help='Resume optimization from a previous build that may have failed'), save_path=typer.Option(None, help='Path to save the optimized pipeline configuration'))`

Build and optimize the configuration specified in the YAML file. Any arguments passed here will override the values in the YAML file.

Parameters:

Name	Type	Description	Default
`yaml_file`	`Path`	Path to the YAML file containing the pipeline configuration.	`Argument(..., help='Path to the YAML file containing the pipeline configuration')`
`max_threads`	`int | None`	Maximum number of threads to use for running operations.	`Option(None, help='Maximum number of threads to use for running operations')`
`model`	`str`	Model to use for optimization. Defaults to "gpt-4o".	*required*
`resume`	`bool`	Whether to resume optimization from a previous run. Defaults to False.	`Option(False, help='Resume optimization from a`

Name	Type	Description	Default
			`previous build that may have failed')`
`save_path`	`Path`	Path to save the optimized pipeline configuration.	`Option(None, help='Path to save the optimized pipeline configuration')`

Source code in `docetl/cli.py`

```python
@app.command()
def build(
    yaml_file: Path = typer.Argument(
        ..., help="Path to the YAML file containing the pipeline
configuration"
    ),
    max_threads: int | None = typer.Option(
        None, help="Maximum number of threads to use for running
operations"
    ),
    resume: bool = typer.Option(
        False, help="Resume optimization from a previous build that may
have failed"
    ),
    save_path: Path = typer.Option(
        None, help="Path to save the optimized pipeline configuration"
    ),
):
    """
    Build and optimize the configuration specified in the YAML file.
    Any arguments passed here will override the values in the YAML file.

    Args:
        yaml_file (Path): Path to the YAML file containing the pipeline
configuration.
        max_threads (int | None): Maximum number of threads to use for
running operations.
        model (str): Model to use for optimization. Defaults to "gpt-4o".
        resume (bool): Whether to resume optimization from a previous run.
Defaults to False.
        save_path (Path): Path to save the optimized pipeline
configuration.
    """
    # Get the current working directory (where the user called the
command)
    cwd = os.getcwd()

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

    runner = DSLRunner.from_yaml(str(yaml_file), max_threads=max_threads)
    runner.optimize(
        save=True,
        return_pipeline=False,
        resume=resume,
        save_path=save_path,
    )
```