

# Semantic Operations

The pandas integration provides several semantic operations through the `.semantic` accessor. Each operation is designed to handle specific types of transformations and analyses using LLMs.

All semantic operations return a new DataFrame that preserves the original columns and adds new columns based on the output schema. For example, if your original DataFrame has a column `text` and you use `map` with an `output={"schema": {"sentiment": "str", "keywords": "list[str]"}}`, the resulting DataFrame will have three columns: `text`, `sentiment`, and `keywords`. This makes it easy to chain operations and maintain data lineage.

## Map Operation

Apply semantic mapping to each row using a language model.

Documentation: <https://ucbepic.github.io/docetl/operators/map/>

### Parameters:

Name	Type	Description	Default
<code>prompt</code>	<code>str</code>	Jinja template string for generating prompts. Use <code>{{input.column_name}}</code> to reference input columns.	<i>required</i>
<code>output</code>	<code>dict[str, Any]</code>	Dictionary containing output configuration with keys: - "schema": Dictionary defining the expected output structure and types. Example: <code>{"entities": "list[str]", "sentiment": "str"}</code> - "mode": Optional output mode. Either "tools" (default) or "structured_output". "structured_output" uses native JSON schema mode for supported models. - "n": Optional number of	<code>None</code>

Name	Type	Description	Default
		outputs to generate for each input (synthetic data generation)	
<code>output_schema</code>	<code>dict[str, Any]</code>	DEPRECATED. Use 'output' parameter instead. Dictionary defining the expected output structure for backward compatibility.	<code>None</code>
<code>**kwargs</code>		Additional configuration options: - model: LLM model to use (default: from config) - batch_prompt: Template for processing multiple documents in a single prompt - max_batch_size: Maximum number of documents to process in a single batch - optimize: Flag to enable operation optimization (default: True) - recursively_optimize: Flag to enable recursive optimization (default: False) - sample: Number of samples to use for the operation - tools: List of tool definitions for LLM use - validate: List of Python expressions to validate output - num_retries_on_validate_failure: Number of retry attempts (default: 0) - gleaning: Configuration for LLM-based refinement - drop_keys: List of keys to drop from input - timeout: Timeout for each LLM call in seconds (default: 120) - max_retries_per_timeout: Maximum retries per timeout (default: 2) - litellm_completion_kwargs: Additional parameters for LiteLLM - skip_on_error: Skip operation if LLM returns error (default: False) - bypass_cache: Bypass cache for this operation (default: False) - n: Number of outputs to generate for	<code>{}</code>

Name	Type	Description	Default
		each input (synthetic data generation)	

Returns:

Type	Description
DataFrame	pd.DataFrame: A new DataFrame containing the transformed data with columns matching the output schema.

Examples:

```
>>> # Extract entities and sentiment
>>> df.semantic.map(
...     prompt="Analyze this text: {{input.text}}",
...     output={
...         "schema": {
...             "entities": "list[str]",
...             "sentiment": "str"
...         }
...     },
...     validate=["len(output['entities']) <= 5"],
...     num_retries_on_validate_failure=2
... )

>>> # Generate synthetic data with multiple variations per input
>>> df.semantic.map(
...     prompt="Create a headline for: {{input.topic}}",
...     output={"schema": {"headline": "str"}, "n": 5}
... )

>>> # Use structured output mode for better JSON schema support
>>> df.semantic.map(
...     prompt="Extract structured data: {{input.text}}",
...     output={
...         "schema": {"name": "str", "age": "int", "tags": "list[str]"},
...         "mode": "structured_output"
...     }
... )
```

**Source code in** `docetl/apis/pd_accessors.py`

```

172 def map(
173     self,
174     prompt: str,
175     output: dict[str, Any] = None,
176     *,
177     output_schema: dict[str, Any] = None,
178     **kwargs,
179 ) -> pd.DataFrame:
180     """
181     Apply semantic mapping to each row using a language model.
182
183     Documentation: https://ucbepic.github.io/docetl/operators/map/
184
185     Args:
186         prompt: Jinja template string for generating prompts. Use
187         {{input.column_name}}
188             to reference input columns.
189         output: Dictionary containing output configuration with keys:
190             - "schema": Dictionary defining the expected output
191             structure and types.
192             Example: {"entities": "list[str]", "sentiment":
193             "str"}
194             - "mode": Optional output mode. Either "tools" (default)
195             or "structured_output".
196             "structured_output" uses native JSON schema mode
197             for supported models.
198             - "n": Optional number of outputs to generate for each
199             input (synthetic data generation)
200         output_schema: DEPRECATED. Use 'output' parameter instead.
201             Dictionary defining the expected output structure
202             for backward compatibility.
203         **kwargs: Additional configuration options:
204             - model: LLM model to use (default: from config)
205             - batch_prompt: Template for processing multiple documents in
206             a single prompt
207             - max_batch_size: Maximum number of documents to process in a
208             single batch
209             - optimize: Flag to enable operation optimization (default:
210             True)
211             - recursively_optimize: Flag to enable recursive optimization
212             (default: False)
213             - sample: Number of samples to use for the operation
214             - tools: List of tool definitions for LLM use
215             - validate: List of Python expressions to validate output
216             - num_retries_on_validate_failure: Number of retry attempts
217             (default: 0)
218             - gleaning: Configuration for LLM-based refinement
219             - drop_keys: List of keys to drop from input
220             - timeout: Timeout for each LLM call in seconds (default:
221             120)
222             - max_retries_per_timeout: Maximum retries per timeout
223             (default: 2)
224             - litellm_completion_kwargs: Additional parameters for
225             LiteLLM
226             - skip_on_error: Skip operation if LLM returns error
227             (default: False)
228             - bypass_cache: Bypass cache for this operation (default:

```

```

229 False)
230         - n: Number of outputs to generate for each input (synthetic
231 data generation)
232
233 Returns:
234     pd.DataFrame: A new DataFrame containing the transformed data
235 with columns
236
237         matching the output schema.
238
239 Examples:
240     >>> # Extract entities and sentiment
241     >>> df.semantic.map(
242     ...     prompt="Analyze this text: {{input.text}}",
243     ...     output={
244     ...         "schema": {
245     ...             "entities": "list[str]",
246     ...             "sentiment": "str"
247     ...         },
248     ...     validate=["len(output['entities']) <= 5"],
249     ...     num_retries_on_validate_failure=2
250     ... )
251
252     >>> # Generate synthetic data with multiple variations per input
253     >>> df.semantic.map(
254     ...     prompt="Create a headline for: {{input.topic}}",
255     ...     output={"schema": {"headline": "str"}, "n": 5}
256     ... )
257
258     >>> # Use structured output mode for better JSON schema support
259     >>> df.semantic.map(
260     ...     prompt="Extract structured data: {{input.text}}",
261     ...     output={
262     ...         "schema": {"name": "str", "age": "int", "tags":
263 "list[str]"},
264     ...         "mode": "structured_output"
265     ...     }
266     ... )
267
268     """
269     # Convert DataFrame to list of dicts for DocETL
270     input_data = self._df.to_dict("records")
271
272     # Handle backward compatibility: if output_schema is provided,
273     convert it to output format
274     if output_schema is not None and output is None:
275         output = {"schema": output_schema}
276         if "n" in kwargs:
277             output["n"] = kwargs.pop("n")
278     elif output is None and output_schema is None:
279         raise ValueError("Either 'output' or 'output_schema' must be
280 provided")
281     elif output is not None and output_schema is not None:
282         raise ValueError(
283             "Cannot provide both 'output' and 'output_schema' parameters"
284         )
285
286     # Validate output parameter
287     if not isinstance(output, dict):
288         raise ValueError("output must be a dictionary")
289
290     if "schema" not in output:

```

```

290         raise ValueError("output must contain a 'schema' key")
291
292     # Validate output mode if provided
293     output_mode = output.get("mode", "tools")
294     if output_mode not in ["tools", "structured_output"]:
295         raise ValueError(
296             f"output mode must be 'tools' or 'structured_output', got
297             '{output_mode}'"
298         )
299
300     # Create map operation config
301     map_config = {
302         "type": "map",
303         "name": f"semantic_map_{len(self._history)}",
304         "prompt": prompt,
305         "output": output,
306         **kwargs,
307     }
308
309     # Create and execute map operation
310     map_op = MapOperation(
311         runner=self.runner,
312         config=map_config,
313         default_model=self.runner.config["default_model"],
314         max_threads=self.runner.max_threads,
315         console=self.runner.console,
316         status=self.runner.status,
317     )
318     results, cost = map_op.execute(input_data)
319
320     return self._record_operation(results, "map", map_config, cost)

```

### Example usage:

```

# Basic map operation
df.semantic.map(
    prompt="Extract sentiment and key points from: {{input.text}}",
    output={
        "schema": {
            "sentiment": "str",
            "key_points": "list[str]"
        }
    },
    validate=["len(output['key_points']) <= 5"],
    num_retries_on_validate_failure=2
)

# Using structured output mode for better JSON schema support
df.semantic.map(
    prompt="Extract detailed information from: {{input.text}}",
    output={
        "schema": {
            "company": "str",
            "product": "str",
            "features": "list[str]"
        },
        "mode": "structured_output"
    }
)

```

```
)

# Backward compatible syntax (still supported)
df.semantic.map(
    prompt="Extract sentiment from: {{input.text}}",
    output_schema={"sentiment": "str"}
)
```

## Filter Operation

Filter DataFrame rows based on semantic conditions.

Documentation: <https://ucbepic.github.io/docetl/operators/filter/>

### Parameters:

Name	Type	Description	Default
<code>prompt</code>	<code>str</code>	Jinja template string for generating prompts	<i>required</i>
<code>output</code>	<code>dict[str, Any]   None</code>	Output configuration with keys: - "schema": Dictionary defining the expected output structure and types For filtering, this must be a single boolean field (e.g., {"keep": "bool"}) - "mode": Optional output mode. Either "tools" (default) or "structured_output"	<code>None</code>
<code>output_schema</code>	<code>dict[str, Any]   None</code>	DEPRECATED. Use 'output' parameter instead. Backward-compatible schema dict.	<code>None</code>
<code>**kwargs</code>		Additional configuration options: - model: LLM model to use - validate: List of validation expressions - num_retries_on_validate_failure: Number of retries - timeout: Timeout in seconds (default: 120) - max_retries_per_timeout: Max retries per timeout (default: 2) - skip_on_error: Skip rows on LLM error (default: False) -	<code>{}</code>

Name	Type	Description	Default
		bypass_cache: Bypass cache for this operation (default: False)	

**Returns:**

Type	Description
DataFrame	pd.DataFrame: Filtered DataFrame containing only rows where the model returned True

**Examples:**

```
>>> # Simple filtering
>>> df.semantic.filter(
...     prompt="Is this about technology? {{input.text}}"
... )
```

```
>>> # Custom output schema
>>> df.semantic.filter(
...     prompt="Analyze if this is relevant: {{input.text}}",
...     output={"schema": {"keep": "bool", "reason": "str"}}
... )
```



Source code in `docetl/apis/pd_accessors.py`

```

676 def filter(
677     self,
678     prompt: str,
679     *,
680     output: dict[str, Any] | None = None,
681     output_schema: dict[str, Any] | None = None,
682     **kwargs,
683 ) -> pd.DataFrame:
684     """
685     Filter DataFrame rows based on semantic conditions.
686
687     Documentation: https://ucbepic.github.io/docetl/operators/filter/
688
689     Args:
690         prompt: Jinja template string for generating prompts
691         output: Output configuration with keys:
692             - "schema": Dictionary defining the expected output structure
693 and types
694             For filtering, this must be a single boolean field (e.g.,
695 {"keep": "bool"})
696             - "mode": Optional output mode. Either "tools" (default) or
697 "structured_output"
698             output_schema: DEPRECATED. Use 'output' parameter instead.
699 Backward-compatible schema dict.
700             **kwargs: Additional configuration options:
701                 - model: LLM model to use
702                 - validate: List of validation expressions
703                 - num_retries_on_validate_failure: Number of retries
704                 - timeout: Timeout in seconds (default: 120)
705                 - max_retries_per_timeout: Max retries per timeout (default:
706 2)
707                 - skip_on_error: Skip rows on LLM error (default: False)
708                 - bypass_cache: Bypass cache for this operation (default:
709 False)
710
711     Returns:
712         pd.DataFrame: Filtered DataFrame containing only rows where the
713 model
714                 returned True
715
716     Examples:
717         >>> # Simple filtering
718         >>> df.semantic.filter(
719             ...     prompt="Is this about technology? {{input.text}}"
720             ... )
721
722         >>> # Custom output schema
723         >>> df.semantic.filter(
724             ...     prompt="Analyze if this is relevant: {{input.text}}",
725             ...     output={"schema": {"keep": "bool", "reason": "str"}}
726             ... )
727     """
728     # Convert DataFrame to list of dicts
729     input_data = self.df.to_dict("records")
730
731     # Backward compatibility and defaults
732     if output is None and output_schema is not None:

```

```

733         output = {"schema": output_schema}
734     if output is None and output_schema is None:
735         output = {"schema": {"keep": "bool"}}
736
737     # Validate output
738     if not isinstance(output, dict):
739         raise ValueError("output must be a dictionary")
740     if "schema" not in output:
741         raise ValueError("output must contain a 'schema' key")
742     output_mode = output.get("mode", "tools")
743     if output_mode not in ["tools", "structured_output"]:
744         raise ValueError(
745             f"output mode must be 'tools' or 'structured_output', got
746             '{output_mode}'"
747         )
748
749     # Create map operation config for filtering
750     filter_config = {
751         "type": "map",
752         "name": f"semantic_filter_{len(self._history)}",
753         "prompt": prompt,
754         "output": output,
755         **kwargs,
756     }
757
758     # Create and execute filter operation
759     filter_op = FilterOperation(
760         runner=self.runner,
761         config=filter_config,
762         default_model=self.runner.config["default_model"],
763         max_threads=self.runner.max_threads,
764         console=self.runner.console,
765         status=self.runner.status,
766     )
767     results, cost = filter_op.execute(input_data)
768
769     return self._record_operation(results, "filter", filter_config, cost)

```

### Example usage:

```

# Simple filtering
df.semantic.filter(
    prompt="Is this text about technology? {{input.text}}"
)

# Custom output with reasons
df.semantic.filter(
    prompt="Analyze if this is relevant: {{input.text}}",
    output={
        "schema": {
            "keep": "bool",
            "reason": "str"
        }
    }
)

```

## Merge Operation (Experimental)

**Note:** The merge operation is an experimental feature based on our equijoin operator. It provides a pandas-like interface for semantic record matching and deduplication. When `fuzzy=True`, it automatically invokes optimization to improve performance while maintaining accuracy.

Semantically merge two DataFrames based on flexible matching criteria.

Documentation: <https://ucbepic.github.io/docetl/operators/equijoin/>

When `fuzzy=True` and no blocking parameters are provided, this method automatically invokes the JoinOptimizer to generate efficient blocking conditions. The optimizer will suggest blocking thresholds and conditions to improve performance while maintaining the target recall. The optimized configuration will be displayed for future reuse.

### Parameters:

Name	Type	Description	Default
<code>right</code>	<code>DataFrame</code>	Right DataFrame to merge with	<i>required</i>
<code>comparison_prompt</code>	<code>str</code>	Prompt template for comparing records	<i>required</i>
<code>fuzzy</code>	<code>bool</code>	Whether to use fuzzy matching with optimization (default: False)	<code>False</code>
<code>**kwargs</code>		Additional configuration options: - <code>model</code> : LLM model to use - <code>blocking_threshold</code> : Threshold for blocking optimization - <code>blocking_conditions</code> : Custom blocking conditions - <code>target_recall</code> : Target recall for optimization (default: 0.95) - <code>estimated_selectivity</code> : Estimated match rate - <code>validate</code> : List of validation expressions - <code>num_retries_on_validate_failure</code> : Number of retries - <code>timeout</code> :	<code>{}</code>

Name	Type	Description	Default
		Timeout in seconds (default: 120) - max_retries_per_timeout: Max retries per timeout (default: 2)	

**Returns:**

Type	Description
<code>DataFrame</code>	pd.DataFrame: Merged DataFrame containing matched records

**Examples:**

```
>>> # Simple merge
>>> merged_df = df1.semantic.merge(
...     df2,
...     comparison_prompt="Are these records about the same entity? {{input1}}
vs {{input2}}"
... )
```

```
>>> # Fuzzy merge with automatic optimization
>>> merged_df = df1.semantic.merge(
...     df2,
...     comparison_prompt="Compare: {{input1}} vs {{input2}}",
...     fuzzy=True,
...     target_recall=0.9
... )
```

```
>>> # Fuzzy merge with manual blocking parameters
>>> merged_df = df1.semantic.merge(
...     df2,
...     comparison_prompt="Compare: {{input1}} vs {{input2}}",
...     fuzzy=False,
...     blocking_threshold=0.8,
...     blocking_conditions=["input1.category == input2.category"]
... )
```

Source code in `docetl/apis/pd_accessors.py`

```

299 def merge(
300     self,
301     right: pd.DataFrame,
302     comparison_prompt: str,
303     *,
304     fuzzy: bool = False,
305     **kwargs,
306 ) -> pd.DataFrame:
307     """
308     Semantically merge two DataFrames based on flexible matching
309     criteria.
310
311     Documentation: https://ucbepic.github.io/docetl/operators/equijoin/
312
313     When fuzzy=True and no blocking parameters are provided, this method
314     automatically
315     invokes the JoinOptimizer to generate efficient blocking conditions.
316     The optimizer
317     will suggest blocking thresholds and conditions to improve
318     performance while
319     maintaining the target recall. The optimized configuration will be
320     displayed
321     for future reuse.
322
323     Args:
324         right: Right DataFrame to merge with
325         comparison_prompt: Prompt template for comparing records
326         fuzzy: Whether to use fuzzy matching with optimization (default:
327         False)
328         **kwargs: Additional configuration options:
329             - model: LLM model to use
330             - blocking_threshold: Threshold for blocking optimization
331             - blocking_conditions: Custom blocking conditions
332             - target_recall: Target recall for optimization (default:
333             0.95)
334             - estimated_selectivity: Estimated match rate
335             - validate: List of validation expressions
336             - num_retries_on_validate_failure: Number of retries
337             - timeout: Timeout in seconds (default: 120)
338             - max_retries_per_timeout: Max retries per timeout (default:
339             2)
340
341     Returns:
342         pd.DataFrame: Merged DataFrame containing matched records
343
344     Examples:
345         >>> # Simple merge
346         >>> merged_df = df1.semantic.merge(
347             ...     df2,
348             ...     comparison_prompt="Are these records about the same
349             entity? {{input1}} vs {{input2}}"
350             ... )
351
352         >>> # Fuzzy merge with automatic optimization
353         >>> merged_df = df1.semantic.merge(
354             ...     df2,
355             ...     comparison_prompt="Compare: {{input1}} vs {{input2}}",

```

```

356         ...     fuzzy=True,
357         ...     target_recall=0.9
358         ... )
359
360     >>> # Fuzzy merge with manual blocking parameters
361     >>> merged_df = df1.semantic.merge(
362         ...     df2,
363         ...     comparison_prompt="Compare: {{input1}} vs {{input2}}",
364         ...     fuzzy=False,
365         ...     blocking_threshold=0.8,
366         ...     blocking_conditions=["input1.category ==
367 input2.category"]
368         ... )
369     """
370     # Convert DataFrames to lists of dicts
371     left_data = self._df.to_dict("records")
372     right_data = right.to_dict("records")
373
374     # Create equijoin operation config
375     join_config = {
376         "type": "equijoin",
377         "name": f"semantic_merge_{len(self._history)}",
378         "comparison_prompt": comparison_prompt,
379         **kwargs,
380     }
381
382     # If fuzzy matching and no blocking params provided, use
383     JoinOptimizer
384     if (
385         fuzzy
386         and not kwargs.get("blocking_threshold")
387         and not kwargs.get("blocking_conditions")
388     ):
389         join_optimizer = JoinOptimizer(
390             self.runner,
391             join_config,
392             target_recall=kwargs.get("target_recall", 0.95),
393             estimated_selectivity=kwargs.get("estimated_selectivity",
394 None),
395         )
396         optimized_config, optimizer_cost, _ =
397         join_optimizer.optimize_equijoin(
398             left_data, right_data, skip_map_gen=True,
399             skip_containment_gen=True
400         )
401
402         # Print optimized config for reuse
403         self.runner.console.log(
404             Panel.fit(
405                 "[bold cyan]Optimized Configuration for Merge[/bold
406 cyan]\n"
407                 "[yellow]Consider adding these parameters to avoid re-
408 running optimization:[/yellow]\n\n"
409                 f"blocking_threshold:
410 {optimized_config.get('blocking_threshold')}\n"
411                 f"blocking_keys:
412 {optimized_config.get('blocking_keys')}\n"
413                 f"blocking_conditions:
414 {optimized_config.get('blocking_conditions', [])}\n",
415                 title="Optimization Results",
416             )

```

```

416         )
417         join_config = optimized_config
         optimizer_cost_value = optimizer_cost
     else:
         optimizer_cost_value = 0.0

     # Create and execute equijoin operation
     join_op = EquijoinOperation(
         runner=self.runner,
         config=join_config,
         default_model=self.runner.config["default_model"],
         max_threads=self.runner.max_threads,
         console=self.runner.console,
         status=self.runner.status,
     )
     results, cost = join_op.execute(left_data, right_data)

     return self._record_operation(
         results, "equijoin", join_config, cost + optimizer_cost_value
     )

```

Example usage:

```

# Simple merge
merged_df = df1.semantic.merge(
    df2,
    comparison_prompt="Are these records about the same entity? {{input1}} vs {{input2}}"
)

# Fuzzy merge with optimization
merged_df = df1.semantic.merge(
    df2,
    comparison_prompt="Compare: {{input1}} vs {{input2}}",
    fuzzy=True,
    target_recall=0.9
)

```

## Aggregate Operation

Semantically aggregate data with optional fuzzy matching.

Documentation: - Resolve Operation: <https://ucbepic.github.io/docetl/operators/resolve/>

- Reduce Operation: <https://ucbepic.github.io/docetl/operators/reduce/>

When fuzzy=True and no blocking parameters are provided in resolve\_kwargs, this method automatically invokes the JoinOptimizer to generate efficient blocking conditions for the resolve phase. The optimizer will suggest blocking thresholds and conditions to improve performance while maintaining the target recall. The optimized configuration will be displayed for future reuse.

The resolve phase is skipped if: - fuzzy=False - reduce\_keys=["\_all"] - input data has 5 or fewer rows

### Parameters:

Name	Type	Description	Default
<code>reduce_prompt</code>	<code>str</code>	Prompt template for the reduction phase	<i>required</i>
<code>output</code>	<code>dict[str, Any]   None</code>	Output configuration with keys: - "schema": Dictionary defining the expected output structure and types - "mode": Optional output mode. Either "tools" (default) or "structured_output"	<code>None</code>
<code>output_schema</code>	<code>dict[str, Any]   None</code>	DEPRECATED. Use 'output' parameter instead. Backward-compatible schema dict	<code>None</code>
<code>fuzzy</code>	<code>bool</code>	Whether to use fuzzy matching for resolution (default: False)	<code>False</code>
<code>comparison_prompt</code>	<code>str   None</code>	Prompt template for comparing records during resolution	<code>None</code>
<code>resolution_prompt</code>	<code>str   None</code>	Prompt template for resolving conflicts	<code>None</code>
<code>resolution_output</code>	<code>dict[str, Any]   None</code>	Output configuration for resolution (new API with schema key)	<code>None</code>
<code>resolution_output_schema</code>	<code>dict[str, Any]   None</code>	Schema for resolution output (deprecated, use <code>resolution_output</code> )	<code>None</code>
<code>reduce_keys</code>	<code>str   list[str]</code>	Keys to group by for reduction (default: ["_all"])	<code>['_all']</code>



Name	Type	Description	Default
<code>resolve_kwargs</code>	<code>dict[str, Any]</code>	Additional kwargs for resolve operation: - model: LLM model to use - blocking_threshold: Threshold for blocking optimization - blocking_conditions: Custom blocking conditions - target_recall: Target recall for optimization (default: 0.95) - estimated_selectivity: Estimated match rate - validate: List of validation expressions - num_retries_on_validate_failure: Number of retries - timeout: Timeout in seconds (default: 120) - max_retries_per_timeout: Max retries per timeout (default: 2)	<code>{}</code>
<code>reduce_kwargs</code>	<code>dict[str, Any]</code>	Additional kwargs for reduce operation: - model: LLM model to use - validate: List of validation expressions - num_retries_on_validate_failure: Number of retries - timeout: Timeout in seconds (default: 120) - max_retries_per_timeout: Max retries per timeout (default: 2)	<code>{}</code>

**Returns:**

Type	Description
<code>DataFrame</code>	<code>pd.DataFrame</code> : Aggregated DataFrame with columns matching <code>output['schema']</code>

**Examples:**

```
>>> # Simple aggregation
>>> df.semantic.agg(
...     reduce_prompt="Summarize these items: {{input.text}}",
...     output={"schema": {"summary": "str"}}
... )
```

```
>>> # Fuzzy matching with automatic optimization
>>> df.semantic.agg(
...     reduce_prompt="Combine these items: {{input.text}}",
...     output={"schema": {"combined": "str"}},
...     fuzzy=True,
...     comparison_prompt="Are these items similar: {{input1.text}} vs {{input2.text}}",
...     resolution_prompt="Resolve conflicts between: {{items}}",
...     resolution_output={"schema": {"resolved": "str"}}
... )
```

```
>>> # Fuzzy matching with manual blocking parameters
>>> df.semantic.agg(
...     reduce_prompt="Combine these items: {{input.text}}",
...     output={"schema": {"combined": "str"}},
...     fuzzy=False,
...     comparison_prompt="Compare items: {{input1.text}} vs {{input2.text}}",
...     resolve_kwargs={
...         "blocking_threshold": 0.8,
...         "blocking_conditions": ["input1.category == input2.category"]
...     }
... )
```

Source code in `docetl/apis/pd_accessors.py`

```

419     def agg(
420         self,
421         *,
422         # Reduction phase params (required)
423         reduce_prompt: str,
424         output: dict[str, Any] | None = None,
425         output_schema: dict[str, Any] | None = None,
426         # Resolution and reduce phase params (optional)
427         fuzzy: bool = False,
428         comparison_prompt: str | None = None,
429         resolution_prompt: str | None = None,
430         resolution_output: dict[str, Any] | None = None,
431         resolution_output_schema: dict[str, Any] | None = None,
432         reduce_keys: str | list[str] = ["_all"],
433         resolve_kwargs: dict[str, Any] = {},
434         reduce_kwargs: dict[str, Any] = {},
435     ) -> pd.DataFrame:
436         """
437         Semantically aggregate data with optional fuzzy matching.
438
439         Documentation:
440         - Resolve Operation:
441         https://ucbepic.github.io/docetl/operators/resolve/
442         - Reduce Operation:
443         https://ucbepic.github.io/docetl/operators/reduce/
444
445         When fuzzy=True and no blocking parameters are provided in
446         resolve_kwargs,
447         this method automatically invokes the JoinOptimizer to generate
448         efficient
449         blocking conditions for the resolve phase. The optimizer will
450         suggest
451         blocking thresholds and conditions to improve performance while
452         maintaining
453         the target recall. The optimized configuration will be displayed
454         for future reuse.
455
456         The resolve phase is skipped if:
457         - fuzzy=False
458         - reduce_keys=["_all"]
459         - input data has 5 or fewer rows
460
461         Args:
462             reduce_prompt: Prompt template for the reduction phase
463             output: Output configuration with keys:
464                 - "schema": Dictionary defining the expected output
465                 structure and types
466                 - "mode": Optional output mode. Either "tools" (default)
467                 or "structured_output"
468             output_schema: DEPRECATED. Use 'output' parameter instead.
469             Backward-compatible schema dict
470             fuzzy: Whether to use fuzzy matching for resolution (default:
471             False)
472             comparison_prompt: Prompt template for comparing records
473             during resolution
474             resolution_prompt: Prompt template for resolving conflicts
475             resolution_output: Output configuration for resolution (new

```

```

476 API with schema key)
477     resolution_output_schema: Schema for resolution output
478 (deprecated, use resolution_output)
479     reduce_keys: Keys to group by for reduction (default:
480 ["_all"])
481     resolve_kwargs: Additional kwargs for resolve operation:
482         - model: LLM model to use
483         - blocking_threshold: Threshold for blocking optimization
484         - blocking_conditions: Custom blocking conditions
485         - target_recall: Target recall for optimization (default:
486 0.95)
487         - estimated_selectivity: Estimated match rate
488         - validate: List of validation expressions
489         - num_retries_on_validate_failure: Number of retries
490         - timeout: Timeout in seconds (default: 120)
491         - max_retries_per_timeout: Max retries per timeout
492 (default: 2)
493     reduce_kwargs: Additional kwargs for reduce operation:
494         - model: LLM model to use
495         - validate: List of validation expressions
496         - num_retries_on_validate_failure: Number of retries
497         - timeout: Timeout in seconds (default: 120)
498         - max_retries_per_timeout: Max retries per timeout
499 (default: 2)
500
501     Returns:
502     pd.DataFrame: Aggregated DataFrame with columns matching
503 output['schema']
504
505     Examples:
506     >>> # Simple aggregation
507     >>> df.semantic.agg(
508         ...     reduce_prompt="Summarize these items:
509 {{input.text}}",
510         ...     output={"schema": {"summary": "str"}},
511         ... )
512
513     >>> # Fuzzy matching with automatic optimization
514     >>> df.semantic.agg(
515         ...     reduce_prompt="Combine these items: {{input.text}}",
516         ...     output={"schema": {"combined": "str"}},
517         ...     fuzzy=True,
518         ...     comparison_prompt="Are these items similar:
519 {{input1.text}} vs {{input2.text}}",
520         ...     resolution_prompt="Resolve conflicts between:
521 {{items}}",
522         ...     resolution_output={"schema": {"resolved": "str"}}
523         ... )
524
525     >>> # Fuzzy matching with manual blocking parameters
526     >>> df.semantic.agg(
527         ...     reduce_prompt="Combine these items: {{input.text}}",
528         ...     output={"schema": {"combined": "str"}},
529         ...     fuzzy=False,
530         ...     comparison_prompt="Compare items: {{input1.text}} vs
531 {{input2.text}}",
532         ...     resolve_kwargs={
533         ...         "blocking_threshold": 0.8,
534         ...         "blocking_conditions": ["input1.category ==
535 input2.category"]
536         ...     }

```

```

537         ... )
538         """
539         # Convert DataFrame to list of dicts
540         input_data = self._df.to_dict("records")
541
542         # Handle backward compatibility: if output_schema is provided,
543         convert it to output format
544         if output_schema is not None and output is None:
545             output = {"schema": output_schema}
546         elif output is None and output_schema is None:
547             raise ValueError("Either 'output' or 'output_schema' must be
548 provided")
549         elif output is not None and output_schema is not None:
550             raise ValueError(
551                 "Cannot provide both 'output' and 'output_schema'
552 parameters"
553             )
554
555         # Validate output parameter
556         if not isinstance(output, dict):
557             raise ValueError("output must be a dictionary")
558         if "schema" not in output:
559             raise ValueError("output must contain a 'schema' key")
560
561         # Handle backward compatibility for resolution_output_schema
562         if resolution_output_schema is not None and resolution_output is
563 None:
564             resolution_output = {"schema": resolution_output_schema}
565         elif resolution_output is not None and resolution_output_schema
566 is not None:
567             raise ValueError(
568                 "Cannot provide both 'resolution_output' and
569 'resolution_output_schema' parameters"
570             )
571
572         # Validate output mode if provided
573         output_mode = output.get("mode", "tools")
574         if output_mode not in ["tools", "structured_output"]:
575             raise ValueError(
576                 f"output mode must be 'tools' or 'structured_output', got
577 '{output_mode}'"
578             )
579
580         # Change keys to list
581         if isinstance(reduce_keys, str):
582             reduce_keys = [reduce_keys]
583
584         # Skip resolution if using _all or fuzzy is False
585         if reduce_keys == ["_all"] or not fuzzy or len(input_data) <= 5:
586             resolved_data = input_data
587         else:
588             # Synthesize comparison prompt if not provided
589             if comparison_prompt is None:
590                 # Build record template from reduce_keys
591                 record_template = ", ".join(
592                     f"{key}: {{{{ input[{0}].{key} }}}}" for key in
593 reduce_keys
594                 )
595
596             # Add context about how fields were created
597             context =

```

```

598 self._synthesize_comparison_context(reduce_keys)
599
600         comparison_prompt = f"""Do the following two records
601 represent the same concept? Your answer should be true or false.{context}
602
603 Record 1: {record_template.replace('input0', 'input1')}
604 Record 2: {record_template.replace('input0', 'input2')}"""
605
606         # Configure resolution
607         resolve_config = {
608             "type": "resolve",
609             "name": f"semantic_resolve_{len(self._history)}",
610             "comparison_prompt": comparison_prompt,
611             **resolve_kwargs,
612         }
613
614         # Add resolution prompt and schema if provided
615         if resolution_prompt:
616             resolve_config["resolution_prompt"] = resolution_prompt
617             if resolution_output:
618                 # Use the new resolution_output format
619                 resolve_config["output"] = resolution_output
620                 if "keys" not in resolve_config["output"]:
621                     # Add keys from schema if not explicitly provided
622                     resolve_config["output"]["keys"] = list(
623                         resolution_output["schema"].keys()
624                     )
625             else:
626                 # No resolution output provided, use reduce_keys
627                 resolve_config["output"] = {"keys": reduce_keys}
628         else:
629             resolve_config["output"] = {"keys": reduce_keys}
630
631         # If blocking params not provided, use JoinOptimizer to
632         synthesize them
633         if not resolve_kwargs or (
634             "blocking_threshold" not in resolve_kwargs
635             and "blocking_conditions" not in resolve_kwargs
636         ):
637             join_optimizer = JoinOptimizer(
638                 self.runner,
639                 resolve_config,
640                 target_recall=(
641                     resolve_kwargs.get("target_recall", 0.95)
642                     if resolve_kwargs
643                     else 0.95
644                 ),
645                 estimated_selectivity=(
646                     resolve_kwargs.get("estimated_selectivity", None)
647                     if resolve_kwargs
648                     else None
649                 ),
650             )
651             optimized_config, optimizer_cost =
652             join_optimizer.optimize_resolve(
653                 input_data
654             )
655
656         # Print optimized config for reuse
657         self.runner.console.log(
658             Panel.fit(

```

```

659         "[bold cyan]Optimized Configuration for
660 Resolve[/bold cyan]\n"
661         "[yellow]Consider adding these parameters to
662 avoid re-running optimization:[/yellow]\n\n"
663         f"blocking_threshold:
664 {optimized_config.get('blocking_threshold')}\n"
665         f"blocking_keys:
666 {optimized_config.get('blocking_keys')}\n"
667         f"blocking_conditions:
668 {optimized_config.get('blocking_conditions', [])}\n",
669         title="Optimization Results",
670     )
671 )
672 else:
673     # Use provided blocking params
674     optimized_config = resolve_config.copy()
675     optimizer_cost = 0.0
676
677     # Execute resolution with optimized config
678     resolve_op = ResolveOperation(
679         runner=self.runner,
680         config=optimized_config,
681         default_model=self.runner.config["default_model"],
682         max_threads=self.runner.max_threads,
683         console=self.runner.console,
684         status=self.runner.status,
685     )
686     resolved_data, resolve_cost = resolve_op.execute(input_data)
687     _ = self._record_operation(
688         resolved_data,
689         "resolve",
690         optimized_config,
691         resolve_cost + optimizer_cost,
692     )
693
694     # Configure reduction
695     reduce_config = {
696         "type": "reduce",
697         "name": f"semantic_reduce_{len(self._history)}",
698         "reduce_key": reduce_keys,
699         "prompt": reduce_prompt,
700         "output": output,
701         **reduce_kwargs,
702     }
703
704     # Execute reduction
705     reduce_op = ReduceOperation(
706         runner=self.runner,
707         config=reduce_config,
708         default_model=self.runner.config["default_model"],
709         max_threads=self.runner.max_threads,
710         console=self.runner.console,
711         status=self.runner.status,
712     )
713     results, reduce_cost = reduce_op.execute(resolved_data)
714
715     return self._record_operation(results, "reduce", reduce_config,
716                                  reduce_cost)

```

Example usage:

```
# Simple aggregation
df.semantic.agg(
    reduce_prompt="Summarize these items: {{input.text}}",
    output={"schema": {"summary": "str"}}
)

# Fuzzy matching with custom resolution
df.semantic.agg(
    reduce_prompt="Combine these items: {{input.text}}",
    output={"schema": {"combined": "str"}},
    fuzzy=True,
    comparison_prompt="Are these items similar: {{input1.text}} vs {{input2.text}}",
    resolution_prompt="Resolve conflicts between: {{items}}",
    resolution_output={"schema": {"resolved": "str"}}
)
```

## Split Operation

Split DataFrame rows into multiple chunks based on content.

Documentation: <https://ucbepic.github.io/docetl/operators/split/>

### Args:

split\_key: The column containing content to split  
 method: Splitting method, either "token\_count" or "delimiter"  
 method\_kwargs: Dictionary containing method-specific parameters:  
 - For "token\_count": {"num\_tokens": int, "model": str (optional)}  
 - For "delimiter": {"delimiter": str, "num\_splits\_to\_group": int (optional)}  
 \*\*kwargs: Additional configuration options:  
 - model: LLM model to use for tokenization (default: from config)

### Returns:

pd.DataFrame: DataFrame with split content, including:  
 - {split\_key}\_chunk: The content of each chunk  
 - {operation\_name}\_id: Unique identifier for the original document  
 - {operation\_name}\_chunk\_num: Sequential chunk number within the document

### Examples:

```
>>> # Split by token count
>>> df.semantic.split(
...     split_key="content",
...     method="token_count",
...     method_kwargs={"num_tokens": 100}
... )

>>> # Split by delimiter
>>> df.semantic.split(
...     split_key="text",
...     method="delimiter",
...     method_kwargs={"delimiter": "
```



```
", "num_splits_to_group": 2} ... )
```

Source code in `docetl/apis/pd_accessors.py`

```

763 def split(
764     self, split_key: str, method: str, method_kwargs: dict[str, Any],
765     **kwargs
766 ) -> pd.DataFrame:
767     """
768     Split DataFrame rows into multiple chunks based on content.
769
770     Documentation: https://ucbepic.github.io/docetl/operators/split/
771
772     Args:
773         split_key: The column containing content to split
774         method: Splitting method, either "token_count" or "delimiter"
775         method_kwargs: Dictionary containing method-specific parameters:
776             - For "token_count": {"num_tokens": int, "model": str
777 (optional)}
778             - For "delimiter": {"delimiter": str, "num_splits_to_group":
779 int (optional)}
780         **kwargs: Additional configuration options:
781             - model: LLM model to use for tokenization (default: from
782 config)
783
784     Returns:
785         pd.DataFrame: DataFrame with split content, including:
786             - {split_key}_chunk: The content of each chunk
787             - {operation_name}_id: Unique identifier for the original
788 document
789             - {operation_name}_chunk_num: Sequential chunk number within
790 the document
791
792     Examples:
793         >>> # Split by token count
794         >>> df.semantic.split(
795             ...     split_key="content",
796             ...     method="token_count",
797             ...     method_kwargs={"num_tokens": 100}
798             ... )
799
800         >>> # Split by delimiter
801         >>> df.semantic.split(
802             ...     split_key="text",
803             ...     method="delimiter",
804             ...     method_kwargs={"delimiter": "\n\n",
805 "num_splits_to_group": 2}
806             ... )
807     """
808     # Convert DataFrame to list of dicts
809     input_data = self._df.to_dict("records")
810
811     # Create split operation config
812     split_config = {
813         "type": "split",
814         "name": f"semantic_split_{len(self._history)}",
815         "split_key": split_key,
816         "method": method,
817         "method_kwargs": method_kwargs,
818         **kwargs,
819     }

```

```
820
821     # Create and execute split operation
822     split_op = SplitOperation(
823         runner=self.runner,
824         config=split_config,
825         default_model=self.runner.config["default_model"],
            max_threads=self.runner.max_threads,
            console=self.runner.console,
            status=self.runner.status,
        )
        results, cost = split_op.execute(input_data)

        return self._record_operation(results, "split", split_config, cost)
```

Example usage:

```
# Split by token count
df.semantic.split(
    split_key="content",
    method="token_count",
    method_kwargs={"num_tokens": 100}
)

# Split by delimiter
df.semantic.split(
    split_key="text",
    method="delimiter",
    method_kwargs={"delimiter": "\n\n", "num_splits_to_group": 2}
)
```

## Gather Operation

Gather contextual information from surrounding chunks to enhance each chunk.

Documentation: <https://ucbepic.github.io/docetl/operators/gather/>

### Parameters:

Name	Type	Description	Default
<code>content_key</code>	<code>str</code>	The column containing the main content to be enhanced	<i>required</i>
<code>doc_id_key</code>	<code>str</code>	The column containing document identifiers to group chunks	<i>required</i>
<code>order_key</code>	<code>str</code>	The column containing chunk order numbers within	<i>required</i>

Name	Type	Description	Default
		documents	
<code>peripheral_chunks</code>	<code>dict[str, Any]   None</code>	Configuration for surrounding context: - previous: {"head": {"count": int}, "tail": {"count": int}, "middle": {}} - next: {"head": {"count": int}, "tail": {"count": int}, "middle": {}}	<code>None</code>
<code>**kwargs</code>		Additional configuration options: - main_chunk_start: Start marker for main chunk (default: "--- Begin Main Chunk ---") - main_chunk_end: End marker for main chunk (default: "--- End Main Chunk ---") - doc_header_key: Column containing document headers (optional)	<code>{}</code>

**Returns:**

Type	Description
<code>DataFrame</code>	<code>pd.DataFrame</code> : DataFrame with enhanced content including: - {content_key}_rendered: The main content with surrounding context

**Examples:**

```
>>> # Basic gathering with surrounding context
>>> df.semantic.gather(
...     content_key="chunk_content",
...     doc_id_key="document_id",
...     order_key="chunk_number",
...     peripheral_chunks={
...         "previous": {"head": {"count": 2}, "tail": {"count": 1}},
...         "next": {"head": {"count": 1}, "tail": {"count": 2}}
...     }
... )
```

```
>>> # Simple gathering without peripheral chunks
>>> df.semantic.gather(
...     content_key="content",
...     doc_id_key="doc_id",
...     order_key="order"
... )
```

Source code in `docetl/apis/pd_accessors.py`

```

827 def gather(
828     self,
829     content_key: str,
830     doc_id_key: str,
831     order_key: str,
832     peripheral_chunks: dict[str, Any] | None = None,
833     **kwargs,
834 ) -> pd.DataFrame:
835     """
836     Gather contextual information from surrounding chunks to enhance each
837     chunk.
838
839     Documentation: https://ucbepic.github.io/docetl/operators/gather/
840
841     Args:
842         content_key: The column containing the main content to be
843         enhanced
844         doc_id_key: The column containing document identifiers to group
845         chunks
846         order_key: The column containing chunk order numbers within
847         documents
848         peripheral_chunks: Configuration for surrounding context:
849             - previous: {"head": {"count": int}, "tail": {"count": int},
850             "middle": {}}
851             - next: {"head": {"count": int}, "tail": {"count": int},
852             "middle": {}}
853         **kwargs: Additional configuration options:
854             - main_chunk_start: Start marker for main chunk (default: "--
855             - Begin Main Chunk ---")
856             - main_chunk_end: End marker for main chunk (default: "---
857             End Main Chunk ---")
858             - doc_header_key: Column containing document headers
859             (optional)
860
861     Returns:
862         pd.DataFrame: DataFrame with enhanced content including:
863             - {content_key}_rendered: The main content with surrounding
864             context
865
866     Examples:
867         >>> # Basic gathering with surrounding context
868         >>> df.semantic.gather(
869             ...     content_key="chunk_content",
870             ...     doc_id_key="document_id",
871             ...     order_key="chunk_number",
872             ...     peripheral_chunks={
873             ...         "previous": {"head": {"count": 2}, "tail": {"count":
874             1}},
875             ...         "next": {"head": {"count": 1}, "tail": {"count": 2}}
876             ...     }
877             ... )
878
879         >>> # Simple gathering without peripheral chunks
880         >>> df.semantic.gather(
881             ...     content_key="content",
882             ...     doc_id_key="doc_id",
883             ...     order_key="order"

```

```

884         ... )
885         """
886         # Convert DataFrame to list of dicts
887         input_data = self.df.to_dict("records")
888
889         # Create gather operation config
890         gather_config = {
891             "type": "gather",
892             "name": f"semantic_gather_{len(self._history)}",
893             "content_key": content_key,
894             "doc_id_key": doc_id_key,
895             "order_key": order_key,
896             **kwargs,
897         }
898
899         # Add peripheral_chunks config if provided
900         if peripheral_chunks is not None:
901             gather_config["peripheral_chunks"] = peripheral_chunks
902
903         # Create and execute gather operation
904         gather_op = GatherOperation(
905             runner=self.runner,
906             config=gather_config,
907             default_model=self.runner.config["default_model"],
908             max_threads=self.runner.max_threads,
909             console=self.runner.console,
910             status=self.runner.status,
911         )
912         results, cost = gather_op.execute(input_data)
913
914         return self._record_operation(results, "gather", gather_config, cost)

```

### Example usage:

```

# Basic gathering with surrounding context
df.semantic.gather(
    content_key="chunk_content",
    doc_id_key="document_id",
    order_key="chunk_number",
    peripheral_chunks={
        "previous": {"head": {"count": 2}, "tail": {"count": 1}},
        "next": {"head": {"count": 1}, "tail": {"count": 2}}
    }
)

# Simple gathering without peripheral chunks
df.semantic.gather(
    content_key="content",
    doc_id_key="doc_id",
    order_key="order"
)

```

## Unnest Operation

Unnest list-like or dictionary values into multiple rows.

Documentation: <https://ucbepic.github.io/docetl/operators/unnest/>

### Parameters:

Name	Type	Description	Default
<code>unnest_key</code>	<code>str</code>	The column containing list-like or dictionary values to unnest	<i>required</i>
<code>keep_empty</code>	<code>bool</code>	Whether to keep rows with empty/null values (default: False)	<code>False</code>
<code>expand_fields</code>	<code>list[str]   None</code>	For dictionary values, which fields to expand (default: all)	<code>None</code>
<code>recursive</code>	<code>bool</code>	Whether to recursively unnest nested structures (default: False)	<code>False</code>
<code>depth</code>	<code>int   None</code>	Maximum depth for recursive unnesting (default: 1, or unlimited if recursive=True)	<code>None</code>
<code>**kwargs</code>		Additional configuration options	<code>{}</code>

### Returns:

Type	Description
<code>DataFrame</code>	<code>pd.DataFrame</code> : DataFrame with unnested values, where: - For lists: Each list element becomes a separate row - For dicts: Specified fields are expanded into the parent row

### Examples:

```
>>> # Unnest a list column
>>> df.semantic.unnest(
...     unnest_key="tags"
... )
# Input: [{"id": 1, "tags": ["a", "b"]}
# Output: [{"id": 1, "tags": "a"}, {"id": 1, "tags": "b"}]
```

```
>>> # Unnest a dictionary column with specific fields
>>> df.semantic.unnest(
```



```
...     unnest_key="user_info",
...     expand_fields=["name", "age"]
... )
# Input: [{"id": 1, "user_info": {"name": "Alice", "age": 30, "email":
"alice@example.com"}}]
# Output: [{"id": 1, "user_info": {...}, "name": "Alice", "age": 30}]
```

```
>>> # Recursive unnesting
>>> df.semantic.unnest(
...     unnest_key="nested_lists",
...     recursive=True,
...     depth=2
... )
```

Source code in `docetl/apis/pd_accessors.py`

```

905 def unnest(
906     self,
907     unnest_key: str,
908     keep_empty: bool = False,
909     expand_fields: list[str] | None = None,
910     recursive: bool = False,
911     depth: int | None = None,
912     **kwargs,
913 ) -> pd.DataFrame:
914     """
915     Unnest list-like or dictionary values into multiple rows.
916
917     Documentation: https://ucbepic.github.io/docetl/operators/unnest/
918
919     Args:
920         unnest_key: The column containing list-like or dictionary values
921         to unnest
922         keep_empty: Whether to keep rows with empty/null values (default:
923         False)
924         expand_fields: For dictionary values, which fields to expand
925         (default: all)
926         recursive: Whether to recursively unnest nested structures
927         (default: False)
928         depth: Maximum depth for recursive unnesting (default: 1, or
929         unlimited if recursive=True)
930         **kwargs: Additional configuration options
931
932     Returns:
933         pd.DataFrame: DataFrame with unnested values, where:
934             - For lists: Each list element becomes a separate row
935             - For dicts: Specified fields are expanded into the parent
936         row
937
938     Examples:
939         >>> # Unnest a list column
940         >>> df.semantic.unnest(
941             ...     unnest_key="tags"
942             ... )
943         # Input: [{"id": 1, "tags": ["a", "b"]}]]
944         # Output: [{"id": 1, "tags": "a"}, {"id": 1, "tags": "b"}]]
945
946         >>> # Unnest a dictionary column with specific fields
947         >>> df.semantic.unnest(
948             ...     unnest_key="user_info",
949             ...     expand_fields=["name", "age"]
950             ... )
951         # Input: [{"id": 1, "user_info": {"name": "Alice", "age": 30,
952         "email": "alice@example.com"}}]]
953         # Output: [{"id": 1, "user_info": {...}, "name": "Alice", "age":
954         30}]
955
956         >>> # Recursive unnesting
957         >>> df.semantic.unnest(
958             ...     unnest_key="nested_lists",
959             ...     recursive=True,
960             ...     depth=2
961             ... )

```

```

962     """
963     # Convert DataFrame to list of dicts
964     input_data = self._df.to_dict("records")
965
966     # Create unnest operation config
967     unnest_config = {
968         "type": "unnest",
969         "name": f"semantic_unnest_{len(self._history)}",
970         "unnest_key": unnest_key,
971         "keep_empty": keep_empty,
972         "recursive": recursive,
973         **kwargs,
974     }
975
976     # Add optional parameters if provided
977     if expand_fields is not None:
978         unnest_config["expand_fields"] = expand_fields
979     if depth is not None:
980         unnest_config["depth"] = depth
981
982     # Create and execute unnest operation
983     unnest_op = UnnestOperation(
984         runner=self.runner,
985         config=unnest_config,
986         default_model=self.runner.config["default_model"],
987         max_threads=self.runner.max_threads,
988         console=self.runner.console,
989         status=self.runner.status,
990     )
991     results, cost = unnest_op.execute(input_data)
992
993     return self._record_operation(results, "unnest", unnest_config, cost)

```

### Example usage:

```

# Unnest a list column
df.semantic.unnest(unnest_key="tags")

# Unnest a dictionary column with specific fields
df.semantic.unnest(
    unnest_key="user_info",
    expand_fields=["name", "age"]
)

# Recursive unnesting with depth control
df.semantic.unnest(
    unnest_key="nested_lists",
    recursive=True,
    depth=2
)

```

## Common Features

All operations support:

## 1. Cost Tracking

```
# After any operation  
print(f"Operation cost: ${df.semantic.total_cost}")
```

## 2. Operation History

```
# View operation history  
for op in df.semantic.history:  
    print(f"{op.op_type}: {op.output_columns}")
```

## 3. Validation Rules

```
# Add validation rules to any map or filter operation  
validate=["len(output['tags']) <= 5", "output['score'] >= 0"]
```

For more details on configuration options and best practices, refer to: - [DocETL Best Practices](#) - [Pipeline Configuration](#) - [Output Schemas](#)