Airflow at Burns & McDonnell

Orchestration from 0 to 100

Bonnie Why

Bonnie Why

is just your average person with way too many interests and not enough time.

Bonnie Why

is just your average person with way too many interests and not enough time.

And that’s my dog

AGENDA

1. Data at Burns & McDonnell
2. Ingestion in one day (or less)
3. Airflow is awesome

Slide Deck
and
Resources

↓

ABOUT BURNS & MCDONNELL

STRENGTH	**14,000+** PROFESSIONALS
DEPTH	**75+** OFFICES WORLDWIDE
EXCELLENCE	**#7** TOP 500 DESIGN FIRMS *Engineering News-Record*
COMMITMENT	**100%** EMPLOYEE-OWNED

IVANPAH SOLAR THERMAL POWER FACILITY

NRG ENERGY | NIPTON, CA

LUNAR PRODUCTION & OPERATION CENTER

INTUITIVE MACHINES| HOUSTON, TX

STARTING LINE

DATA AT BURNS & MCDONNELL

HARD TO MAINTAIN

- Multitude of varied source systems
- Multitude of differing tools and processes
- Teams working in silos

HARD TO MAINTAIN

HARD TO TRUST

- Duplicated effort, duplicated data
- Lack of discoverability
- Unclear as to what the data means

HARD TO CHANGE

- Brittle, interconnected systems
- Unsure of who is using the data, and how often
- Little testing, end users find the bugs

HARD TO CHANGE

FINISH LINE

DATA AT BURNS & MCDONNELL

SCALEABLE

- Scale with the business, as well as the data
- Centralized for uniform access
- Maintainable to keep up with high demand

SCALEABLE

RELIABLE

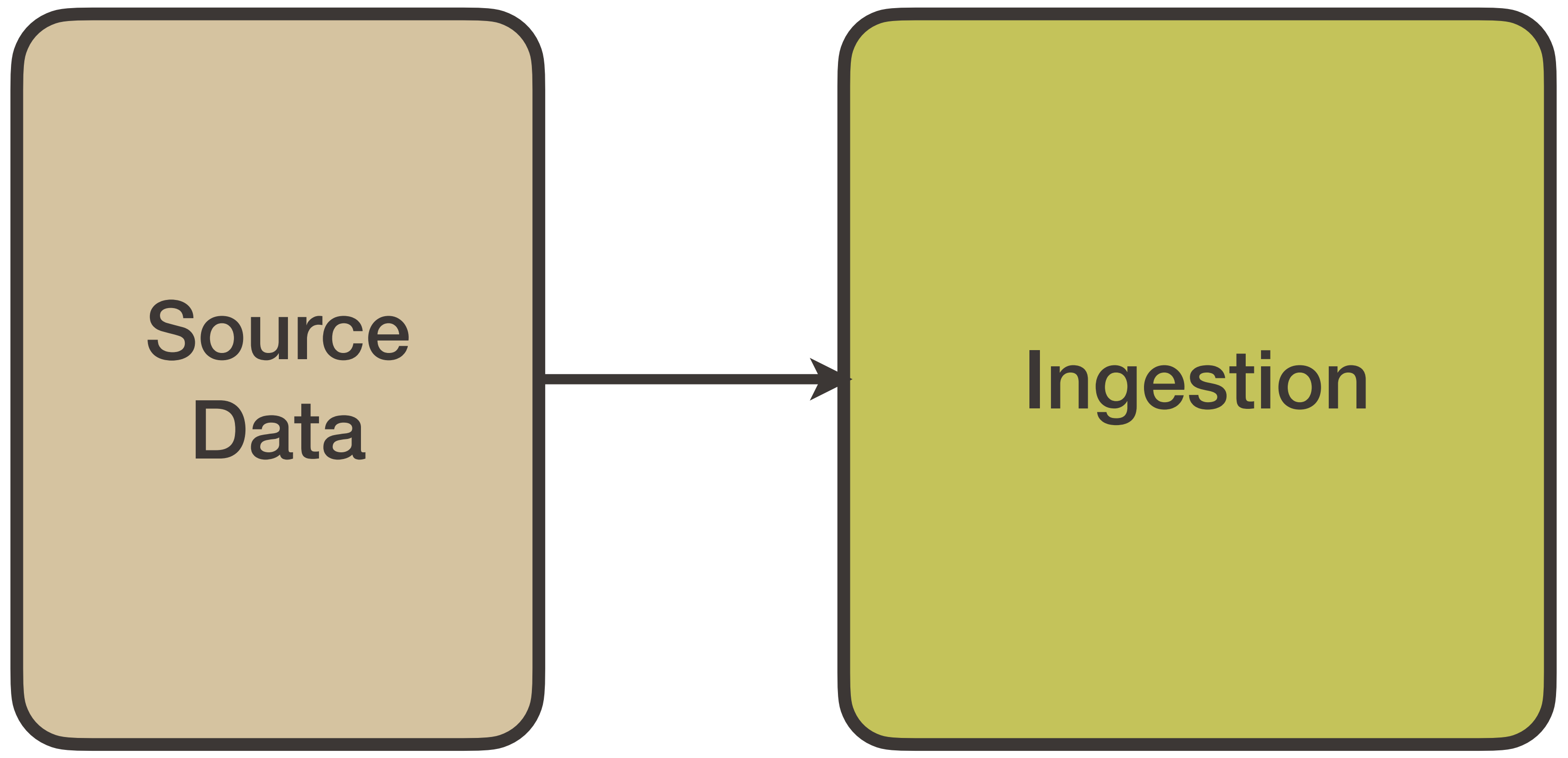
- Company-wide understanding, shared language
- Using the “right” data
- Obvious, secure, and accurate data lineage

RELIABLE

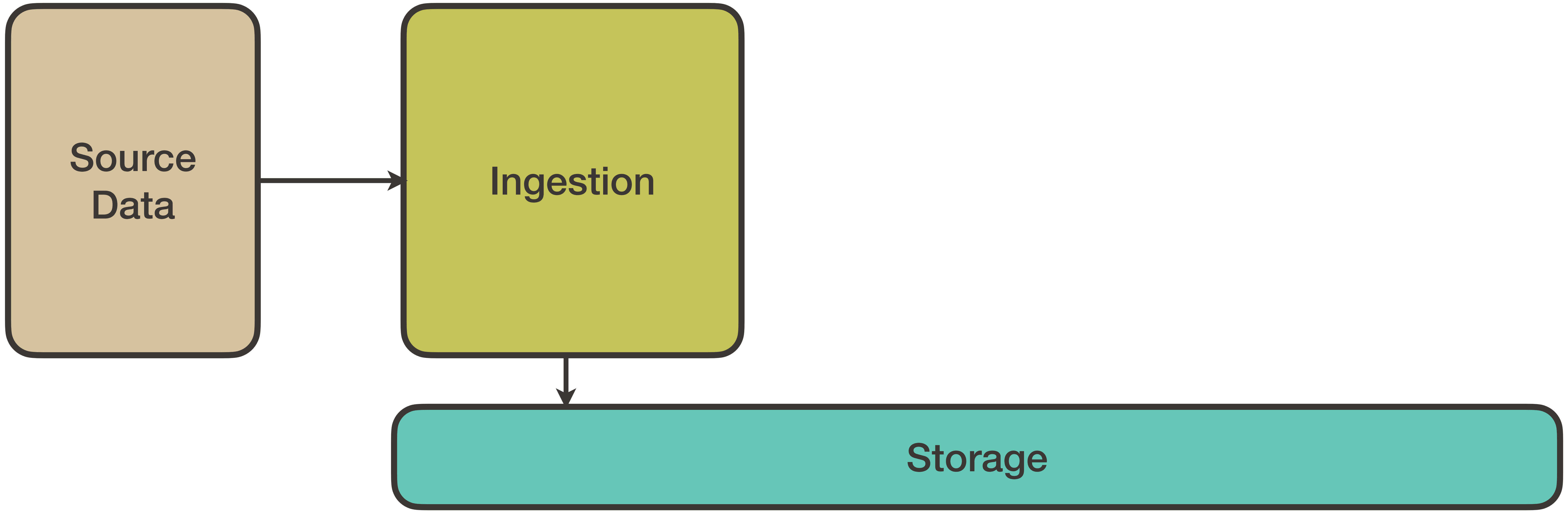
EVOLVABLE

- Composable data management strategy
- Grow with emerging technologies
- Support any new use cases our business needs

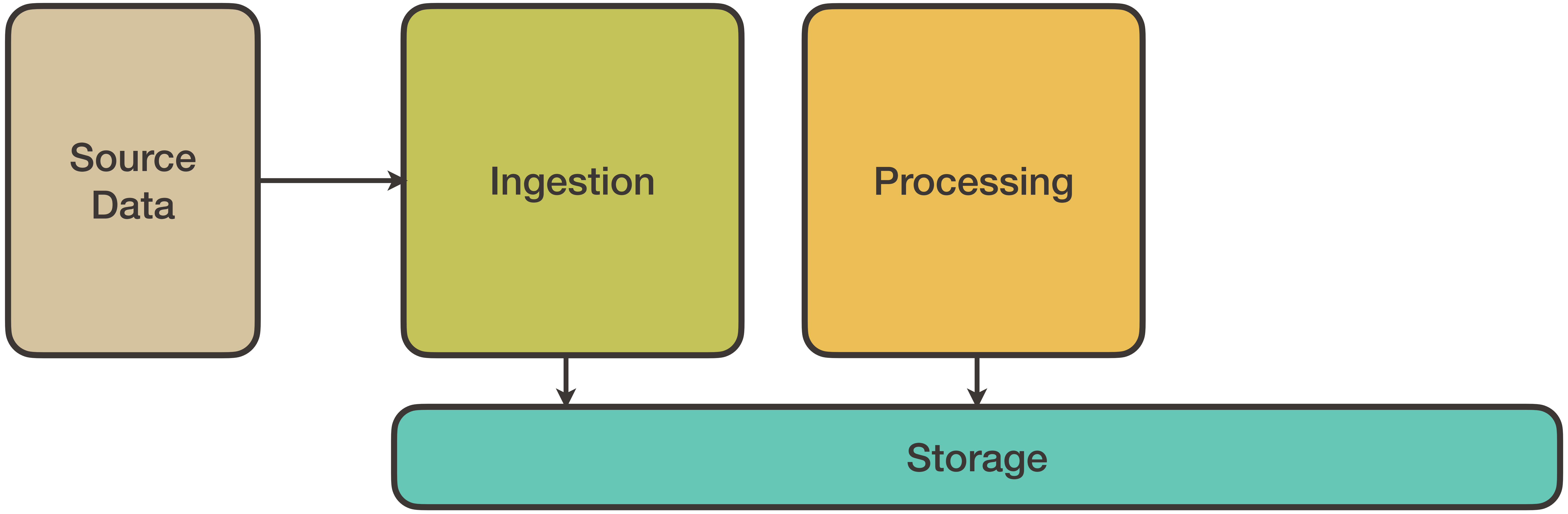
EVOLVABLE



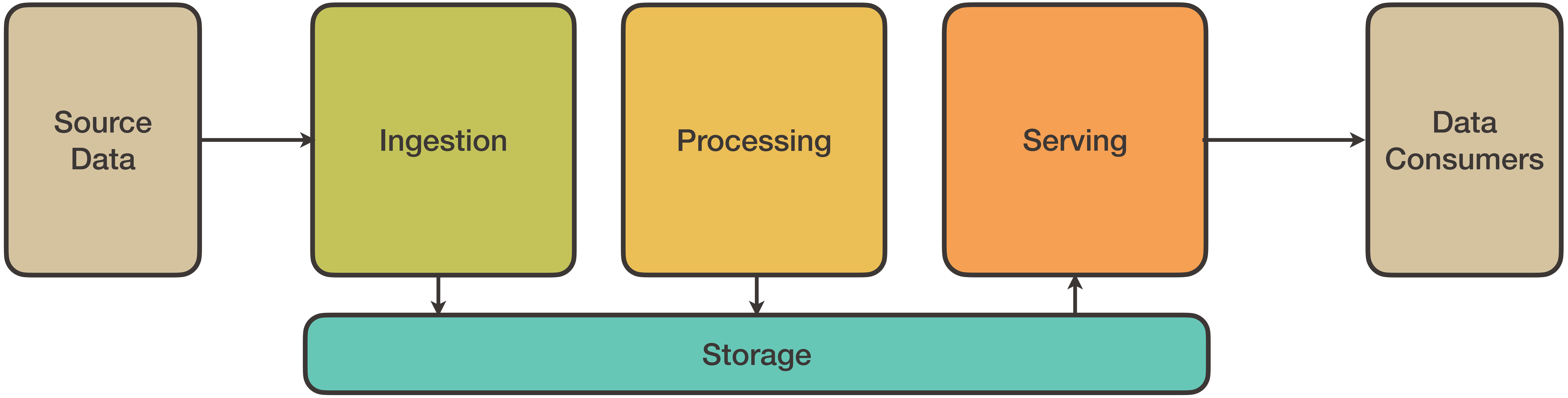
Adapted from “Designing Cloud Data Platforms” by
Daniel Zburivsky, Lynda Partner



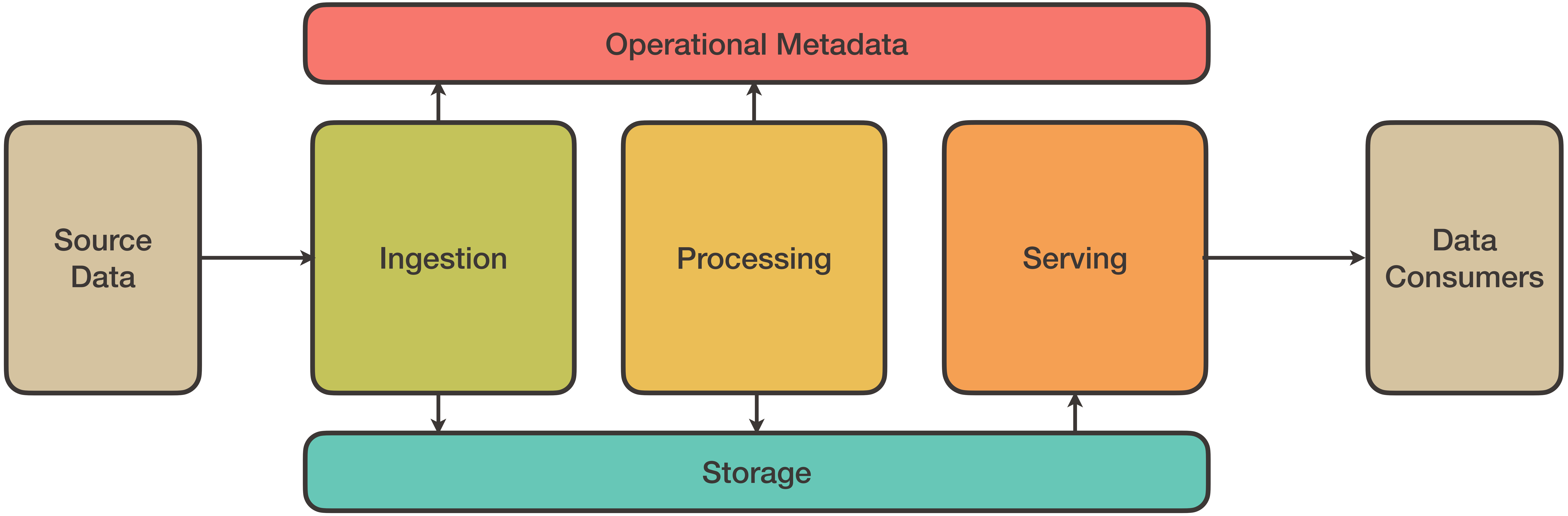
Adapted from “Designing Cloud Data Platforms” by Daniel Zburivsky, Lynda Partner



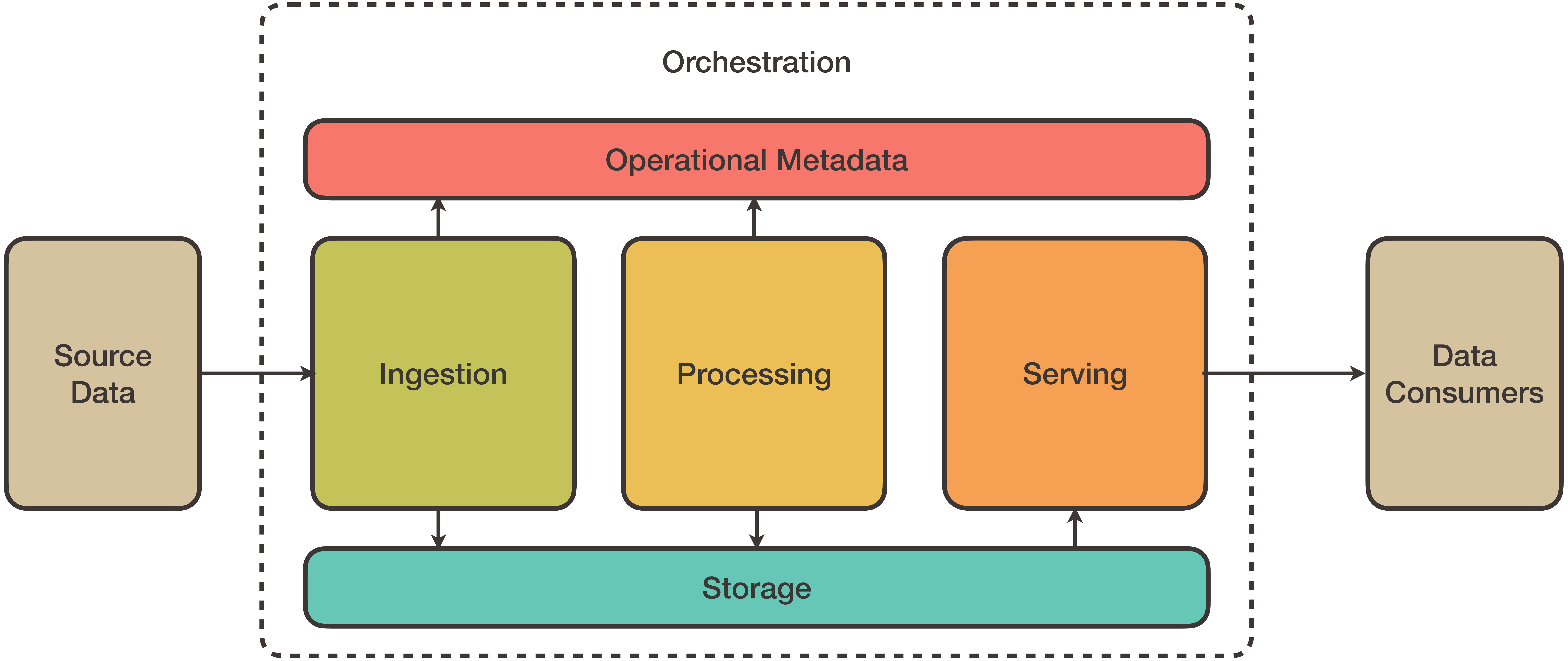
Adapted from "Designing Cloud Data Platforms" by Daniel Zburivsky, Lynda Partner



Adapted from “Designing Cloud Data Platforms” by
Daniel Zburivsky, Lynda Partner



Adapted from "Designing Cloud Data Platforms" by Daniel Zburivsky, Lynda Partner



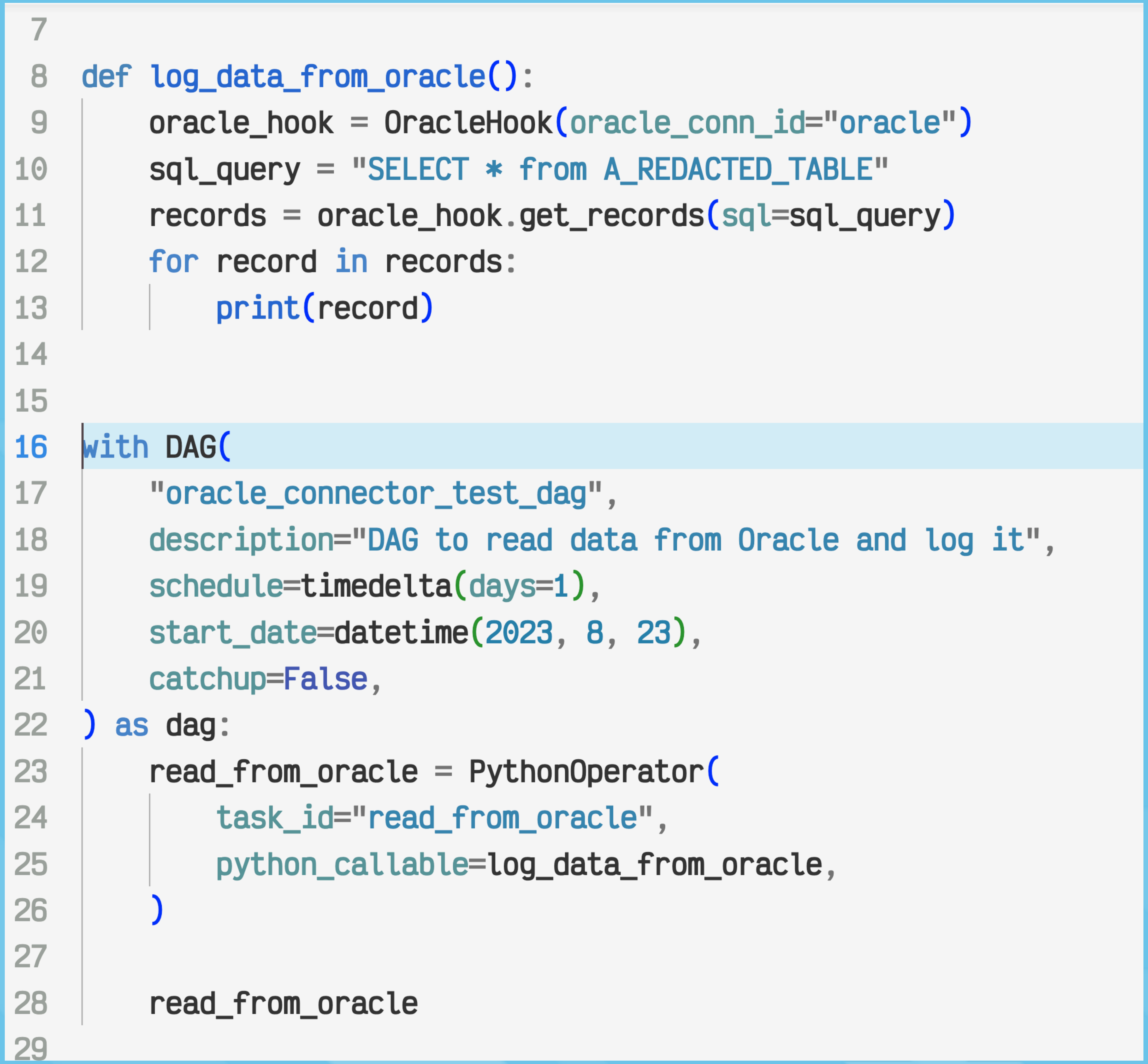
Adapted from "Designing Cloud Data Platforms" by Daniel Zburivsky, Lynda Partner

ON YOUR MARK

You have one day.

STARTING AT ZERO

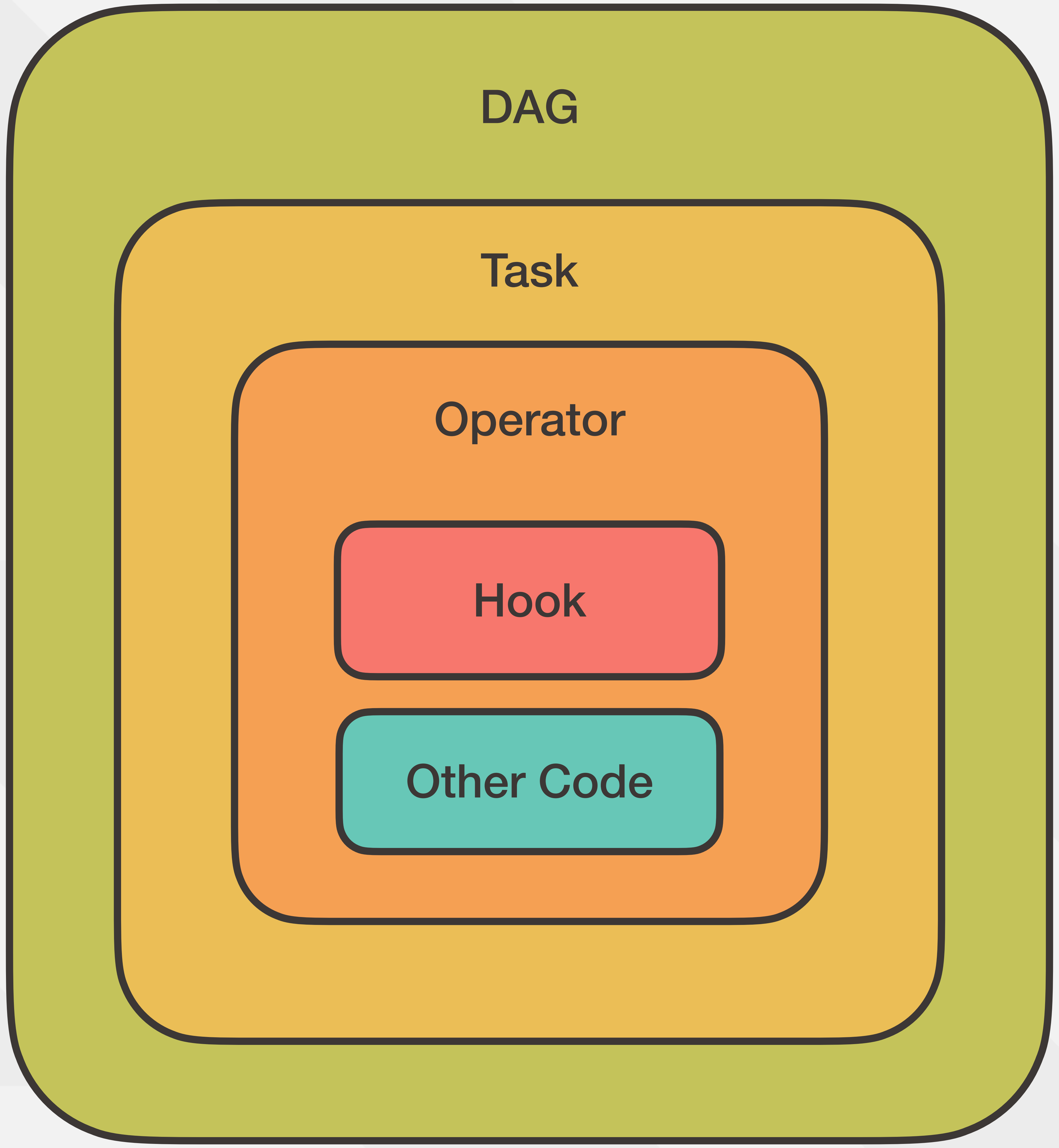
GETTING STARTED

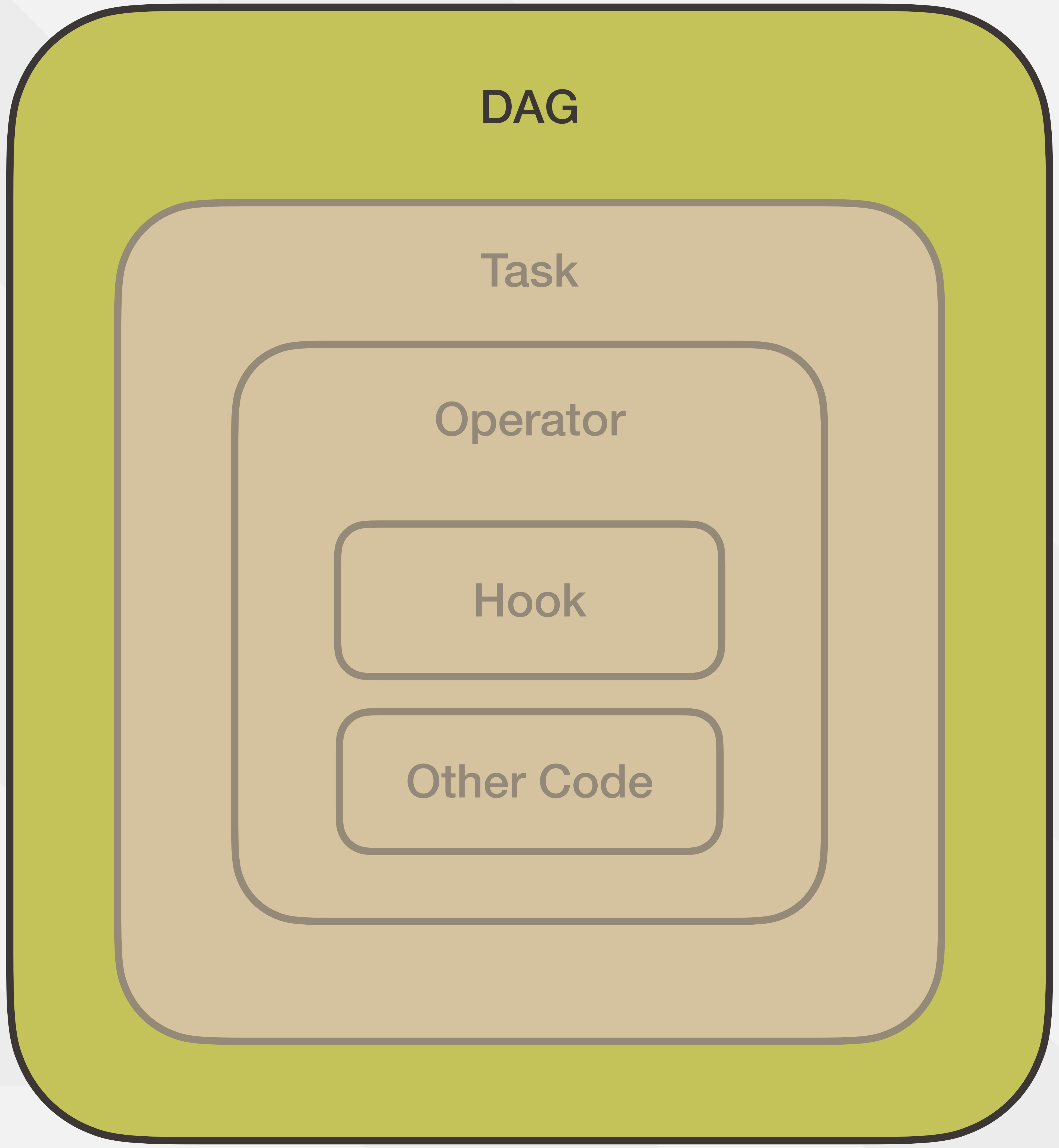


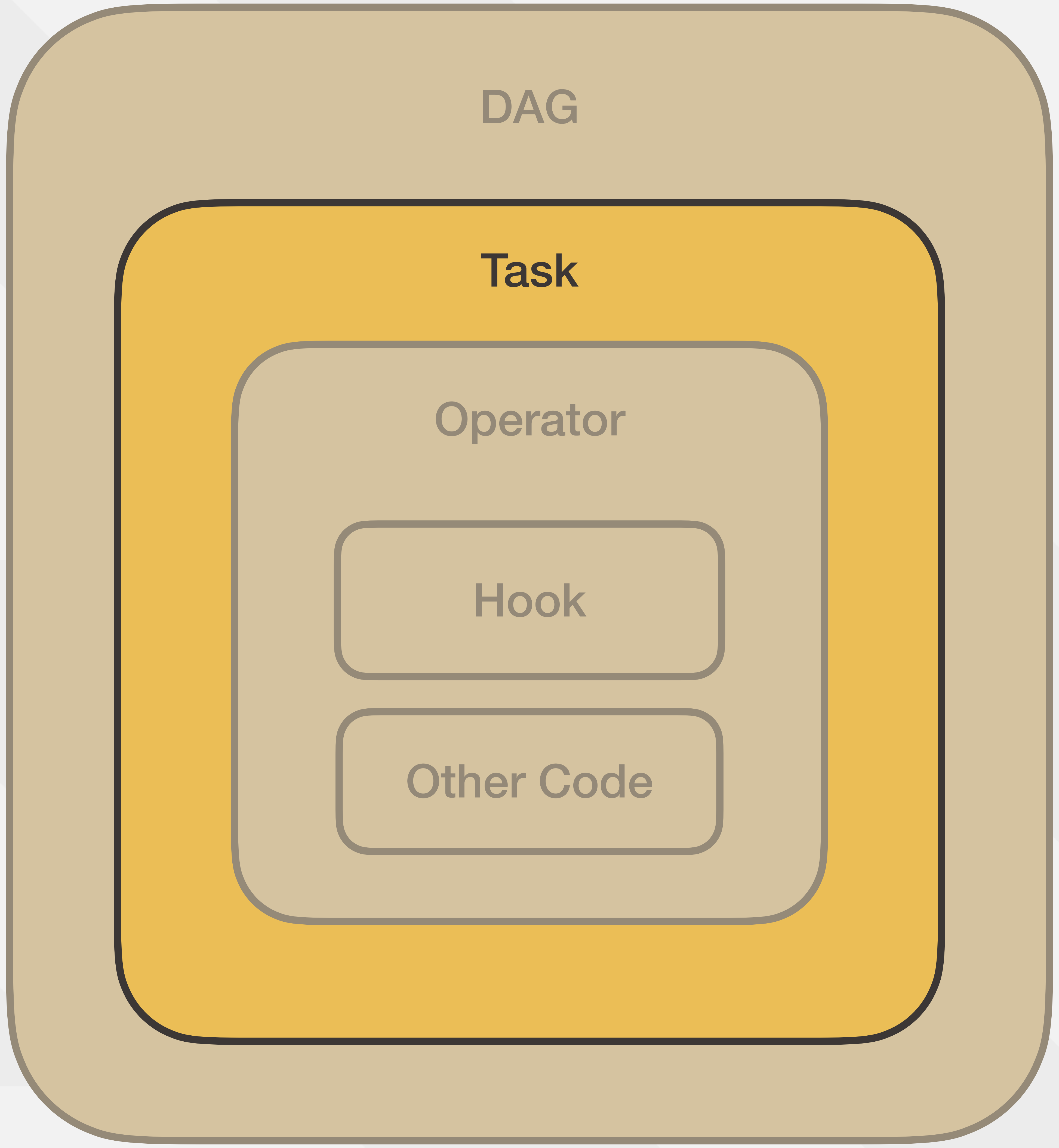
```python

def log_data_from_oracle():
    oracle_hook = OracleHook(oracle_conn_id="oracle")
    sql_query = "SELECT * from A_REDACTED_TABLE"
    records = oracle_hook.get_records(sql=sql_query)
    for record in records:
        print(record)


with DAG(
    "oracle_connector_test_dag",
    description="DAG to read data from Oracle and log it",
    schedule=timedelta(days=1),
    start_date=datetime(2023, 8, 23),
    catchup=False,
) as dag:
    read_from_oracle = PythonOperator(
        task_id="read_from_oracle",
        python_callable=log_data_from_oracle,
    )

    read_from_oracle

```

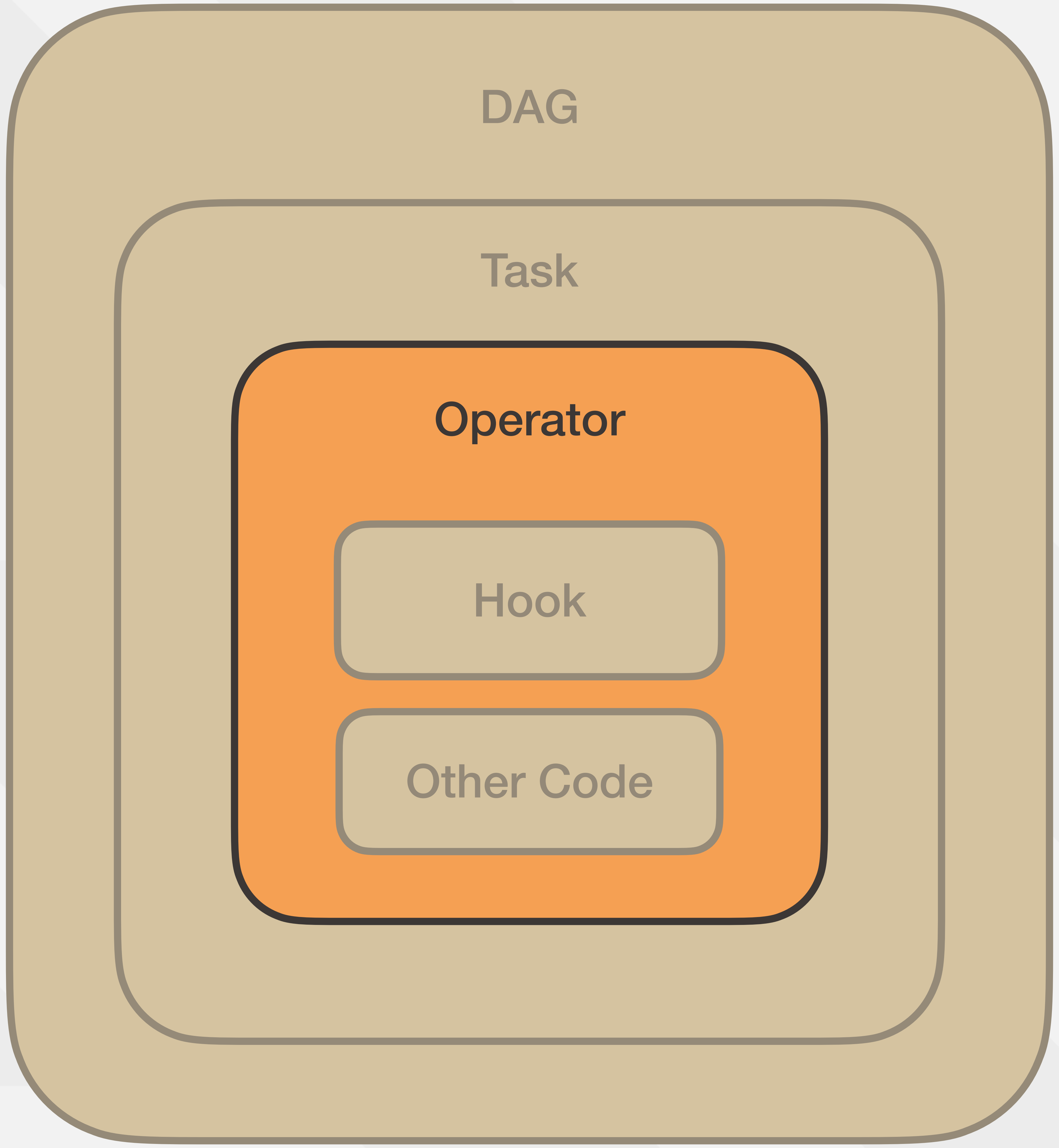
GETTING STARTED

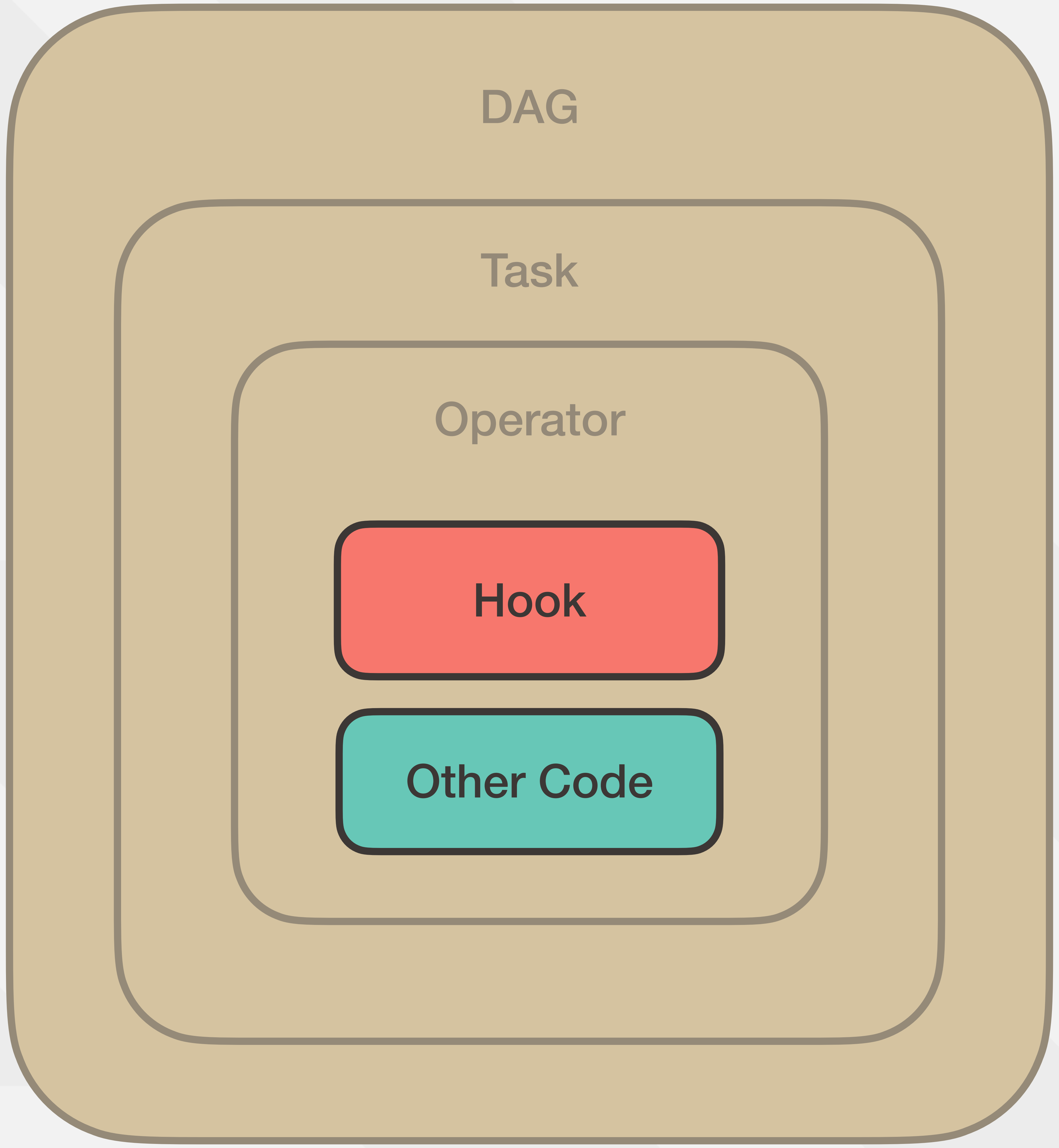
GETTING STARTED


DAG
Task
Operator
Hook
Other Code


DAG
Task
Operator
Hook
Other Code


DAG
Task
Operator
Hook
Other Code


DAG
Task
Operator
Hook
Other Code


DAG
Task
Operator
Hook
Other Code

DAG

Task

Operator

Hook

Other Code

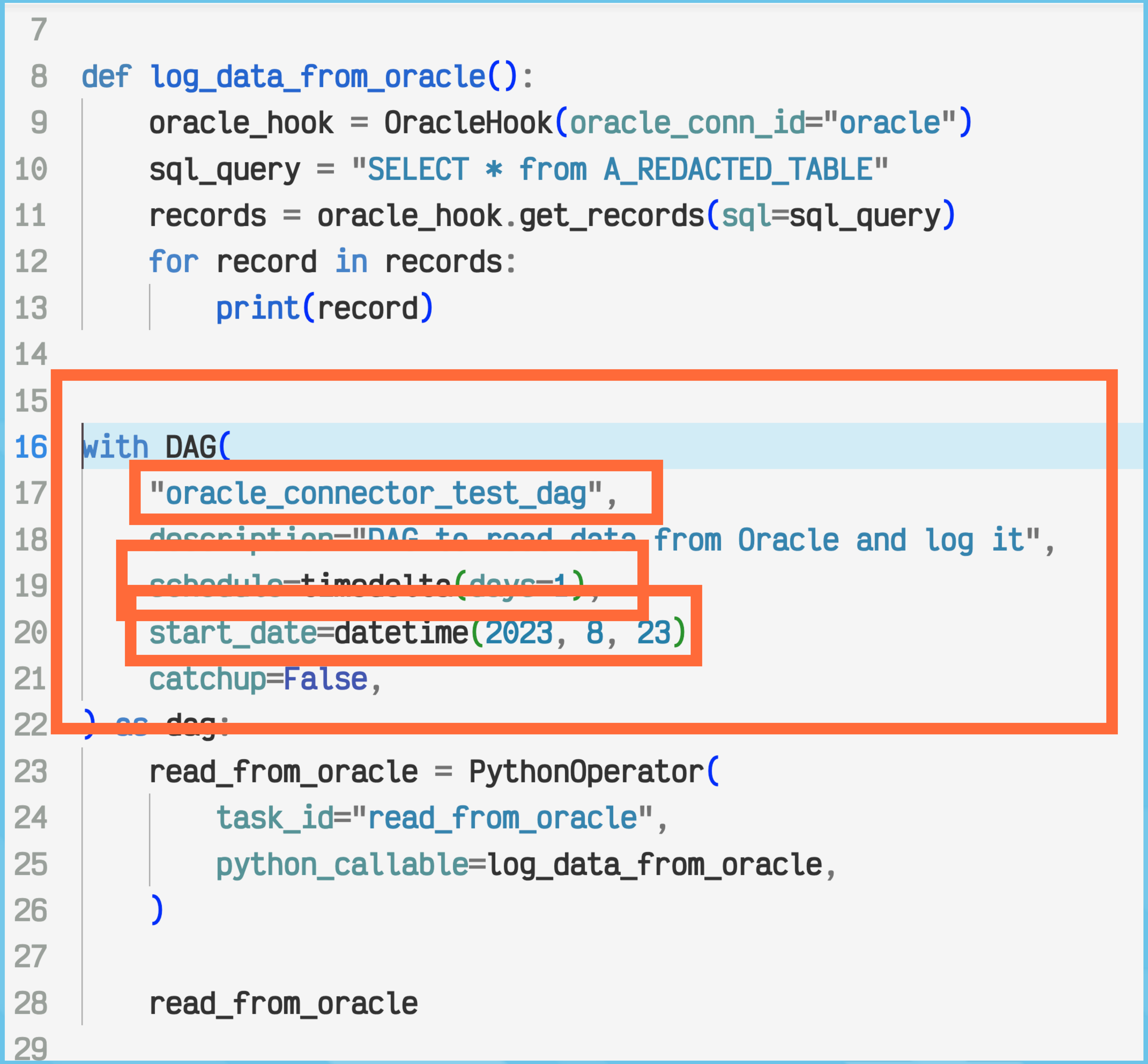
DAG

Task

Operator

Hook

Other Code



```python

def log_data_from_oracle():
    oracle_hook = OracleHook(oracle_conn_id="oracle")
    sql_query = "SELECT * from A_REDACTED_TABLE"
    records = oracle_hook.get_records(sql=sql_query)
    for record in records:
        print(record)


with DAG(
    "oracle_connector_test_dag",
    description="DAG to read data from Oracle and log it",
    schedule=timedelta(days=1),
    start_date=datetime(2023, 8, 23)
    catchup=False,
) as dag:
    read_from_oracle = PythonOperator(
        task_id="read_from_oracle",
        python_callable=log_data_from_oracle,
    )

    read_from_oracle

```

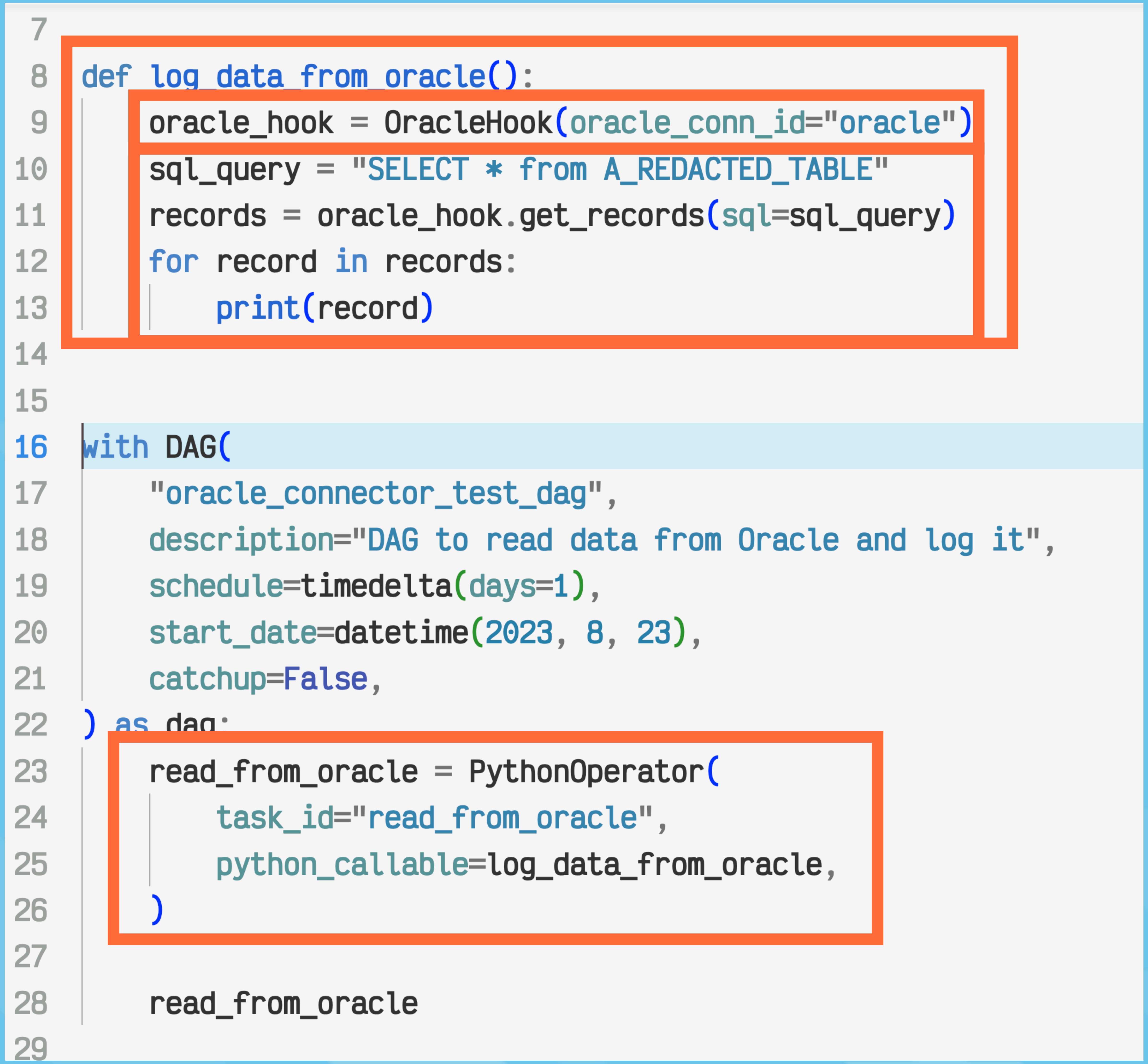
DAG

Task

Operator

Hook

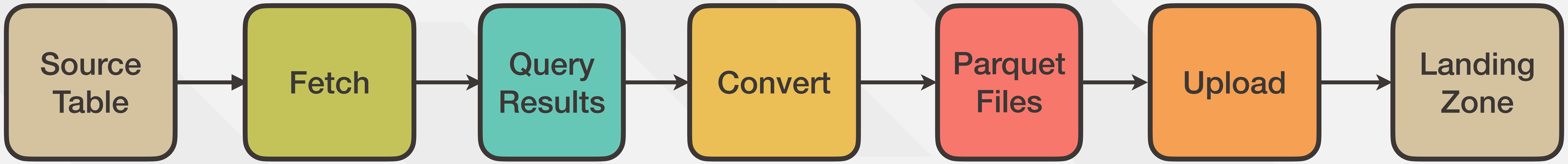
Other Code



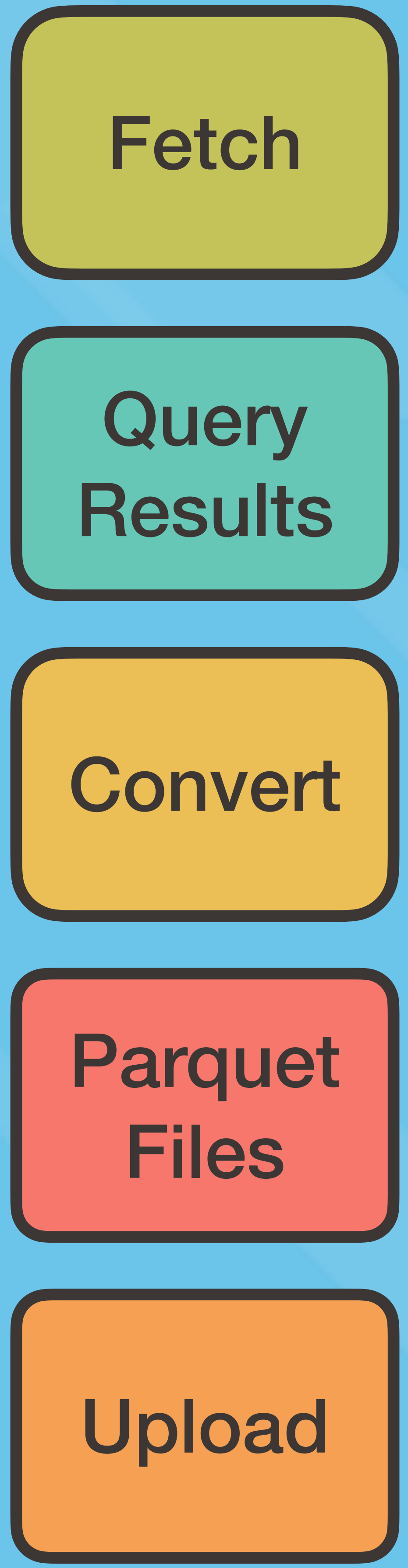
```

def log_data_from_oracle():
    oracle_hook = OracleHook(oracle_conn_id="oracle")
    sql_query = "SELECT * from A_REDACTED_TABLE"
    records = oracle_hook.get_records(sql=sql_query)
    for record in records:
        print(record)


with DAG(
    "oracle_connector_test_dag",
    description="DAG to read data from Oracle and log it",
    schedule=timedelta(days=1),
    start_date=datetime(2023, 8, 23),
    catchup=False,
) as dag:
    read_from_oracle = PythonOperator(
        task_id="read_from_oracle",
        python_callable=log_data_from_oracle,
    )

    read_from_oracle

```

GETTING STARTED

CUSTOM OPERATOR



CUSTOM OPERATOR



CUSTOM OPERATOR

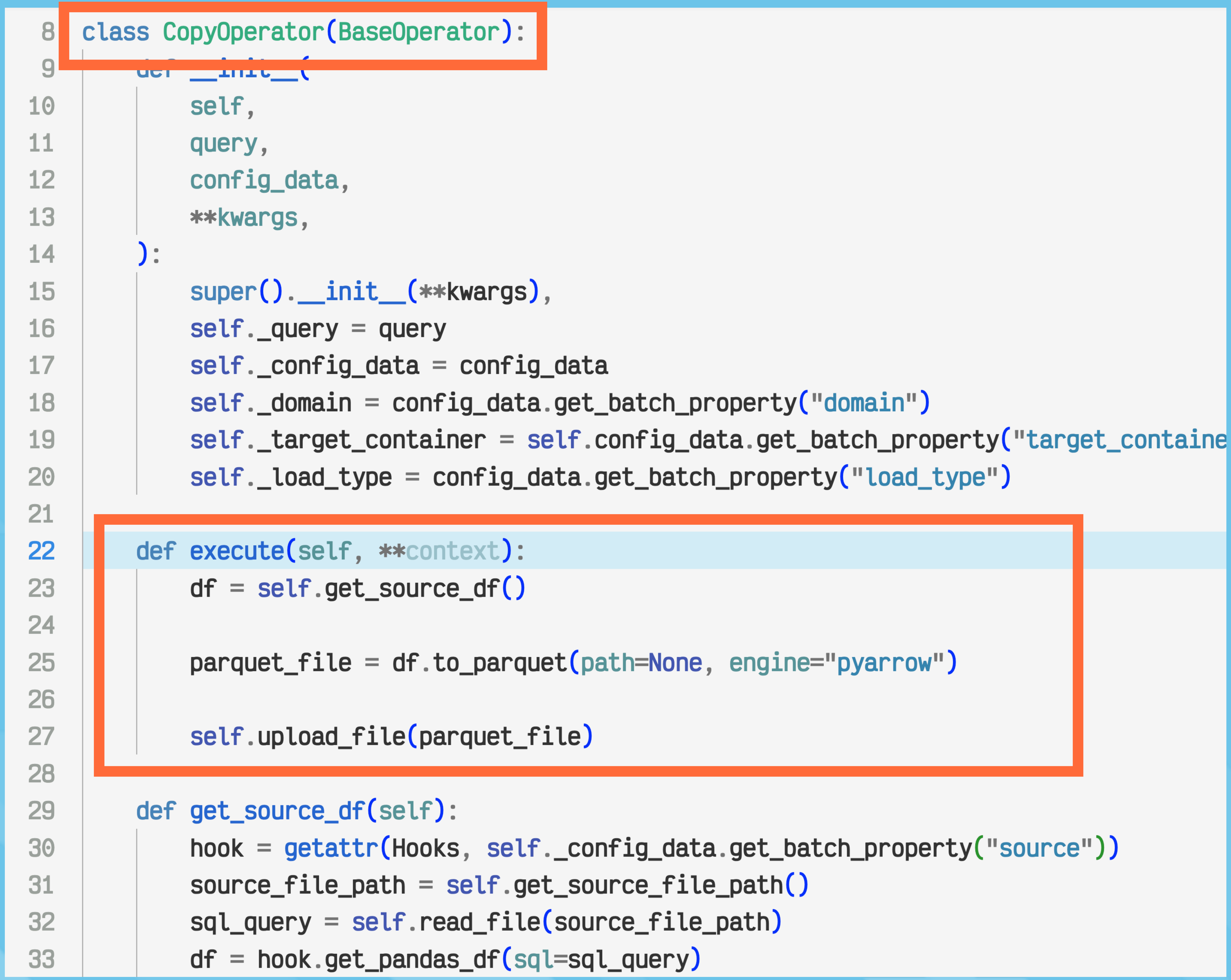
Fetch

Query Results

Convert

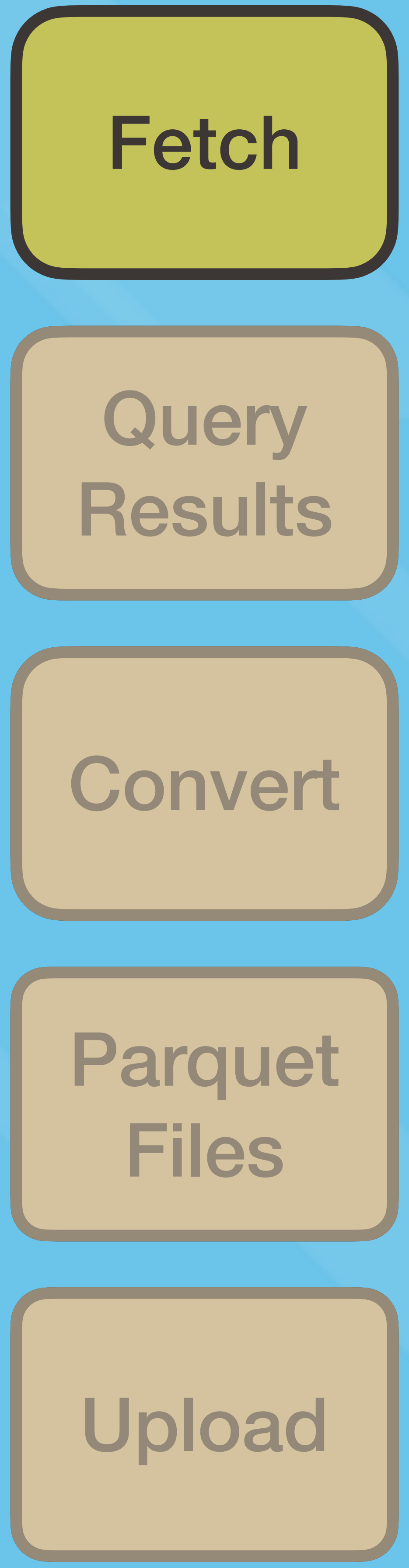
Parquet Files

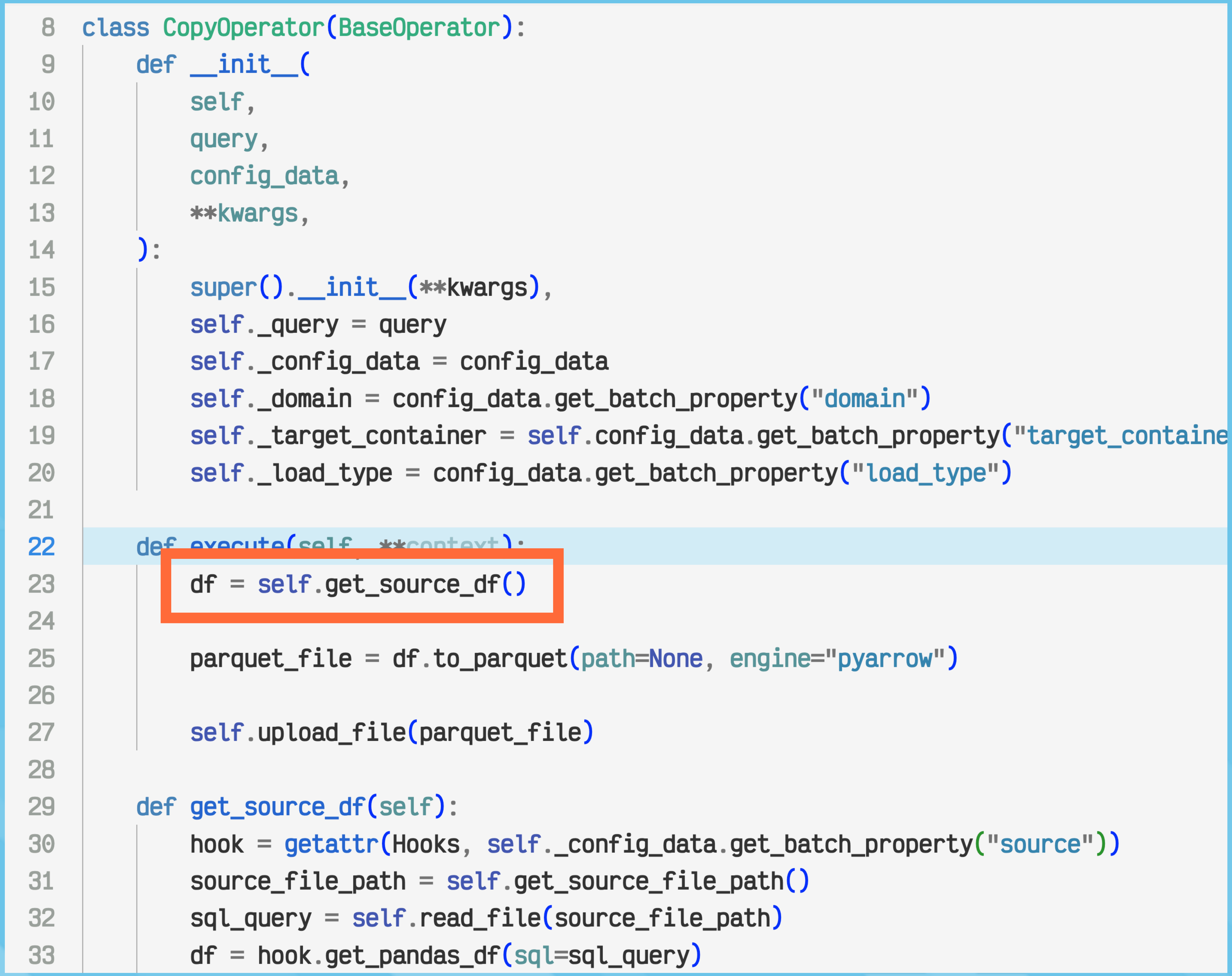
Upload



```
class CopyOperator(BaseOperator):
    def __init__(
        self,
        query,
        config_data,
        **kwargs,
    ):
        super().__init__(**kwargs),
        self._query = query
        self._config_data = config_data
        self._domain = config_data.get_batch_property("domain")
        self._target_container = self.config_data.get_batch_property("target_containe
        self._load_type = config_data.get_batch_property("load_type")

    def execute(self, **context):
        df = self.get_source_df()

        parquet_file = df.to_parquet(path=None, engine="pyarrow")

        self.upload_file(parquet_file)

    def get_source_df(self):
        hook = getattr(Hooks, self._config_data.get_batch_property("source"))
        source_file_path = self.get_source_file_path()
        sql_query = self.read_file(source_file_path)
        df = hook.get_pandas_df(sql=sql_query)
```

CUSTOM OPERATOR





```
class CopyOperator(BaseOperator):
    def __init__(
        self,
        query,
        config_data,
        **kwargs,
    ):
        super().__init__(**kwargs),
        self._query = query
        self._config_data = config_data
        self._domain = config_data.get_batch_property("domain")
        self._target_container = self.config_data.get_batch_property("target_containe
        self._load_type = config_data.get_batch_property("load_type")

    def execute(self, **context):
        df = self.get_source_df()

        parquet_file = df.to_parquet(path=None, engine="pyarrow")

        self.upload_file(parquet_file)

    def get_source_df(self):
        hook = getattr(Hooks, self._config_data.get_batch_property("source"))
        source_file_path = self.get_source_file_path()
        sql_query = self.read_file(source_file_path)
        df = hook.get_pandas_df(sql=sql_query)
```

CUSTOM OPERATOR

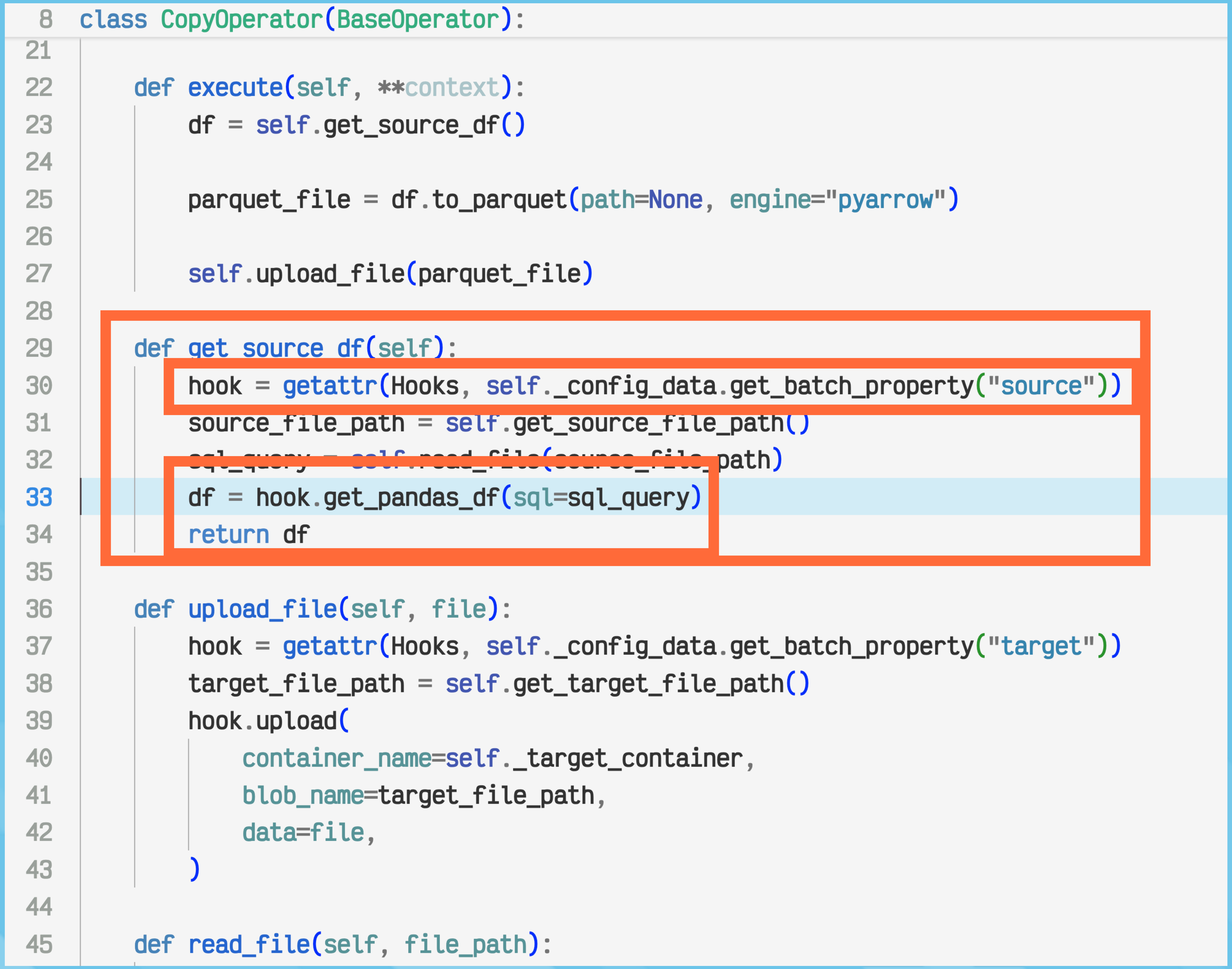
Fetch

Query Results

Convert

Parquet Files

Upload



```python
class CopyOperator(BaseOperator):

    def execute(self, **context):
        df = self.get_source_df()

        parquet_file = df.to_parquet(path=None, engine="pyarrow")

        self.upload_file(parquet_file)

    def get_source_df(self):
        hook = getattr(Hooks, self._config_data.get_batch_property("source"))
        source_file_path = self.get_source_file_path()
        sql_query = self.read_file(source_file_path)
        df = hook.get_pandas_df(sql=sql_query)
        return df

    def upload_file(self, file):
        hook = getattr(Hooks, self._config_data.get_batch_property("target"))
        target_file_path = self.get_target_file_path()
        hook.upload(
            container_name=self._target_container,
            blob_name=target_file_path,
            data=file,
        )

    def read_file(self, file_path):
```

CUSTOM OPERATOR

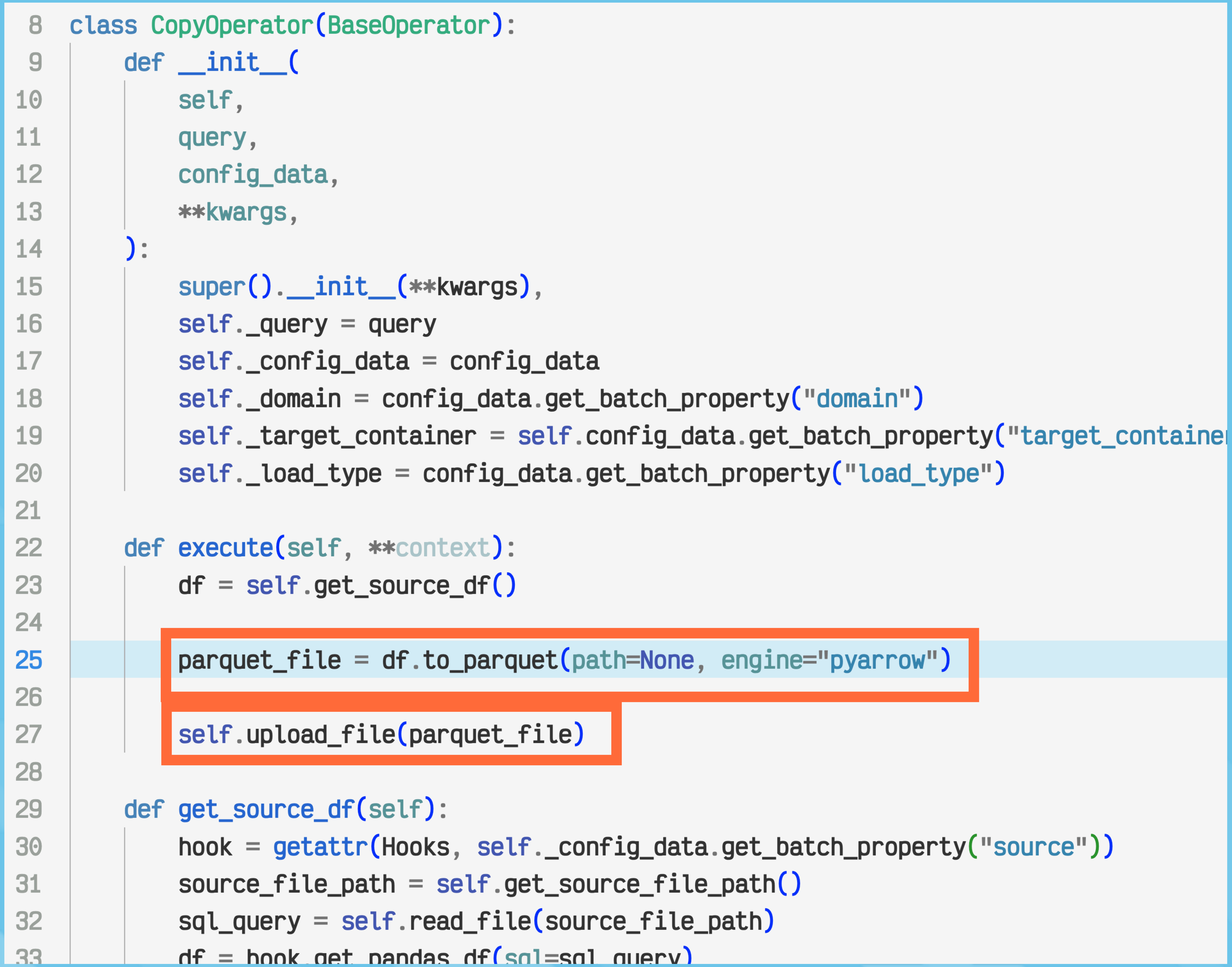
Fetch

Query Results

Convert

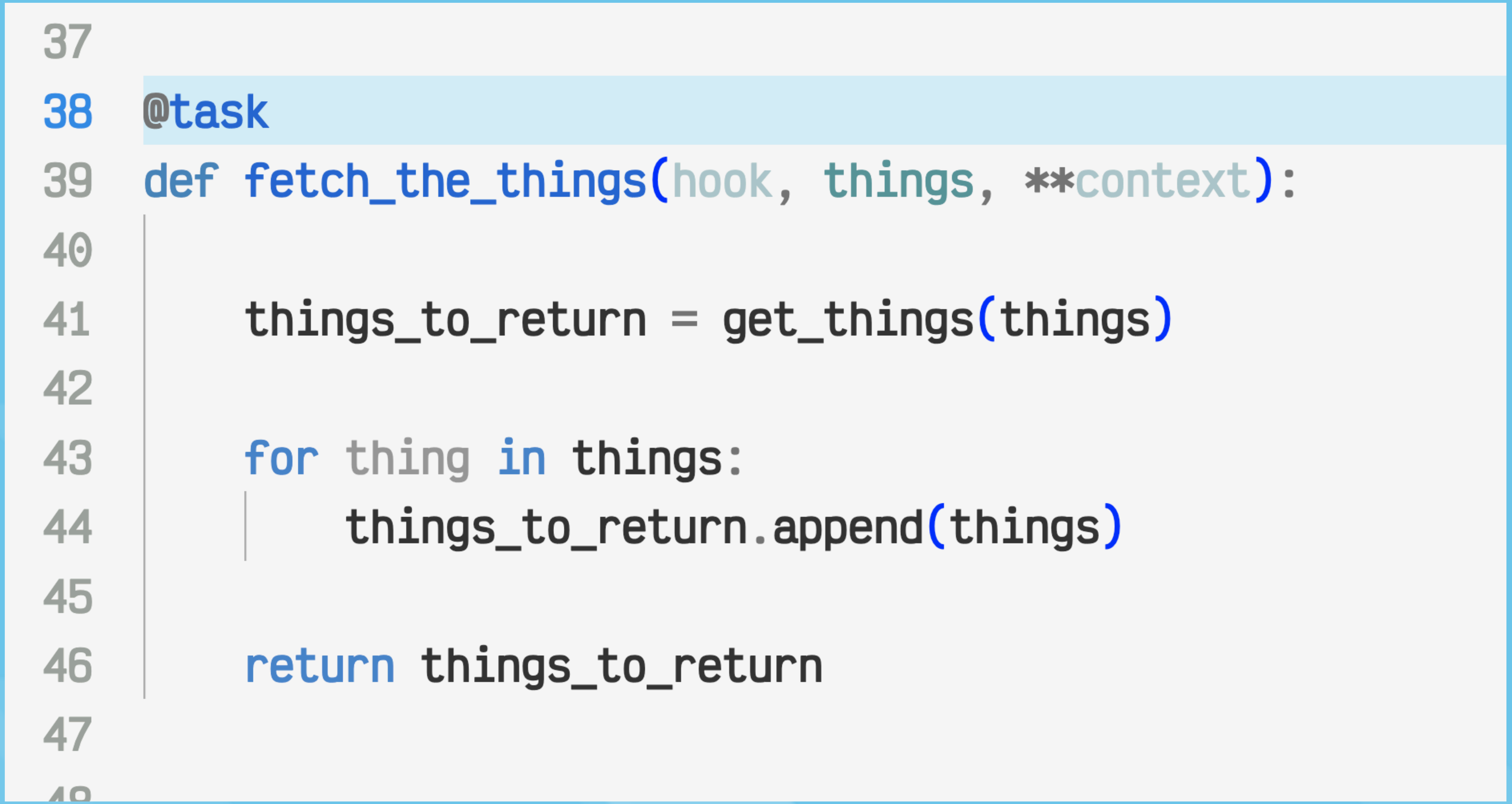
Parquet Files

Upload



```python
class CopyOperator(BaseOperator):
    def __init__(
        self,
        query,
        config_data,
        **kwargs,
    ):
        super().__init__(**kwargs),
        self._query = query
        self._config_data = config_data
        self._domain = config_data.get_batch_property("domain")
        self._target_container = self.config_data.get_batch_property("target_containe
        self._load_type = config_data.get_batch_property("load_type")

    def execute(self, **context):
        df = self.get_source_df()

        parquet_file = df.to_parquet(path=None, engine="pyarrow")

        self.upload_file(parquet_file)

    def get_source_df(self):
        hook = getattr(Hooks, self._config_data.get_batch_property("source"))
        source_file_path = self.get_source_file_path()
        sql_query = self.read_file(source_file_path)
        df = hook.get_pandas_df(sql=sql_query)
```

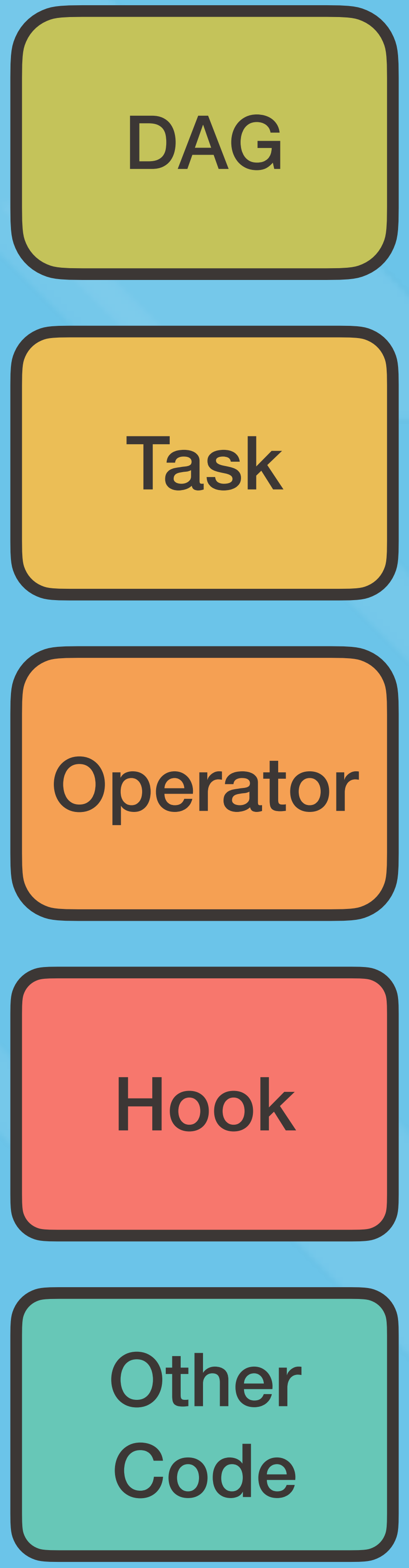
CUSTOM OPERATOR



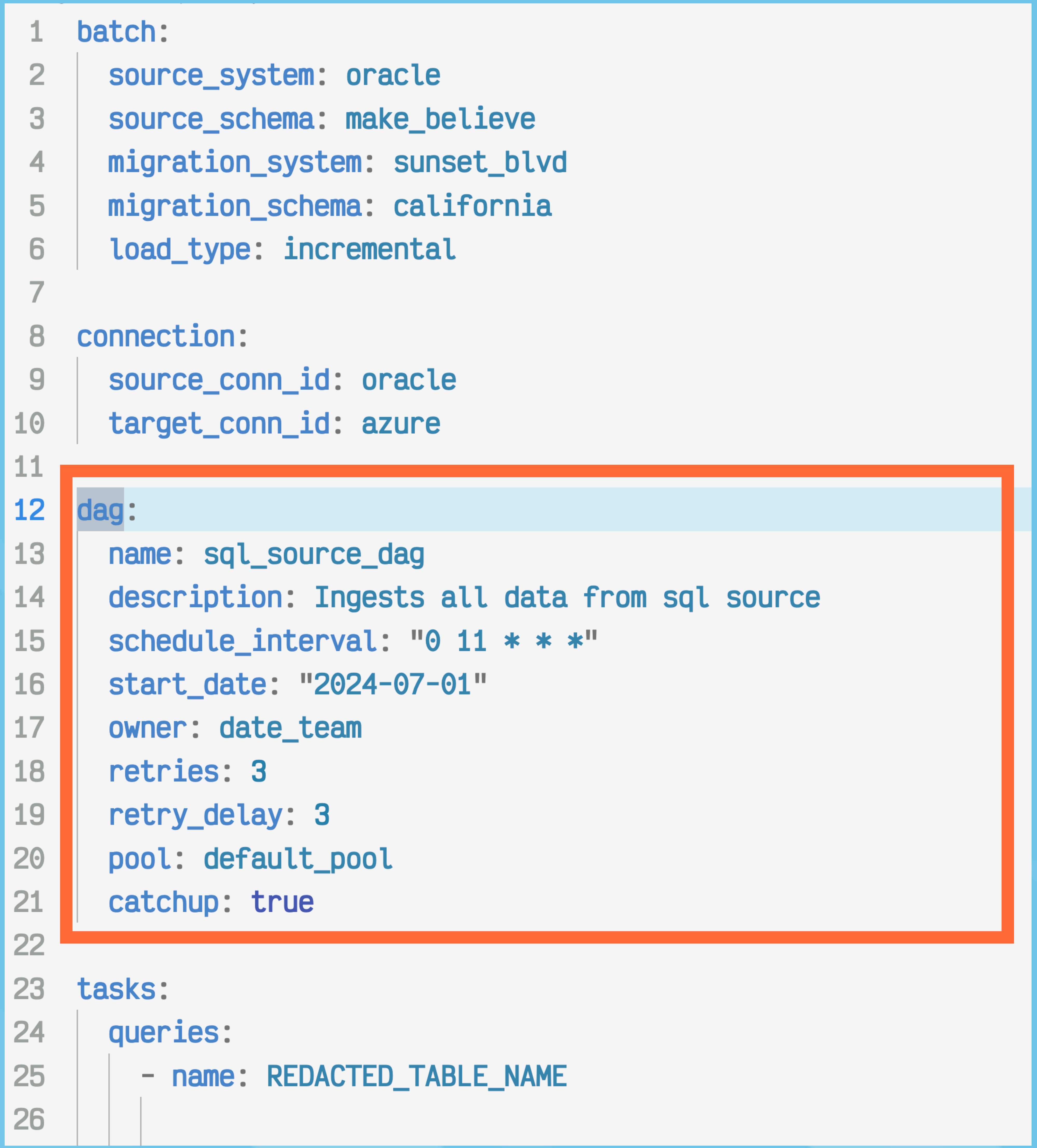
```python

@task
def fetch_the_things(hook, things, **context):

    things_to_return = get_things(things)

    for thing in things:
        things_to_return.append(things)

    return things_to_return

```

CUSTOM OPERATOR

GENERATOR PATTERN



GENERATOR PATTERN



```yaml
batch:
  source_system: oracle
  source_schema: make_believe
  migration_system: sunset_blvd
  migration_schema: california
  load_type: incremental

connection:
  source_conn_id: oracle
  target_conn_id: azure

dag:
  name: sql_source_dag
  description: Ingests all data from sql source
  schedule_interval: "0 11 * * *"
  start_date: "2024-07-01"
  owner: date_team
  retries: 3
  retry_delay: 3
  pool: default_pool
  catchup: true

tasks:
  queries:
    - name: REDACTED_TABLE_NAME

```

DAG

Task

Operator

Hook

Other Code

GENERATOR PATTERN

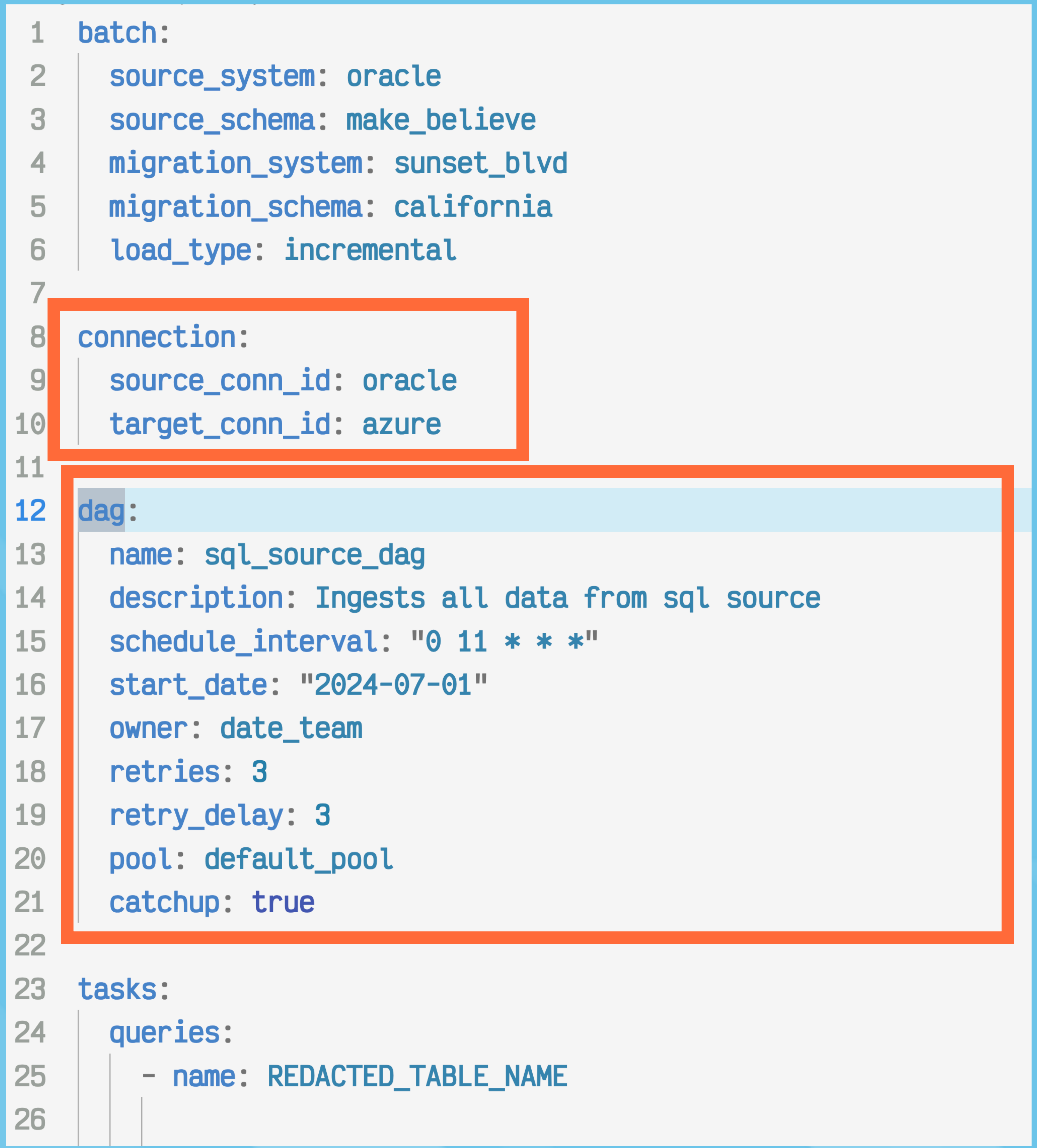
DAG

Task

Operator

Hook

Other Code



```yaml
batch:
  source_system: oracle
  source_schema: make_believe
  migration_system: sunset_blvd
  migration_schema: california
  load_type: incremental

connection:
  source_conn_id: oracle
  target_conn_id: azure

dag:
  name: sql_source_dag
  description: Ingests all data from sql source
  schedule_interval: "0 11 * * *"
  start_date: "2024-07-01"
  owner: date_team
  retries: 3
  retry_delay: 3
  pool: default_pool
  catchup: true

tasks:
  queries:
    - name: REDACTED_TABLE_NAME

```

GENERATOR PATTERN

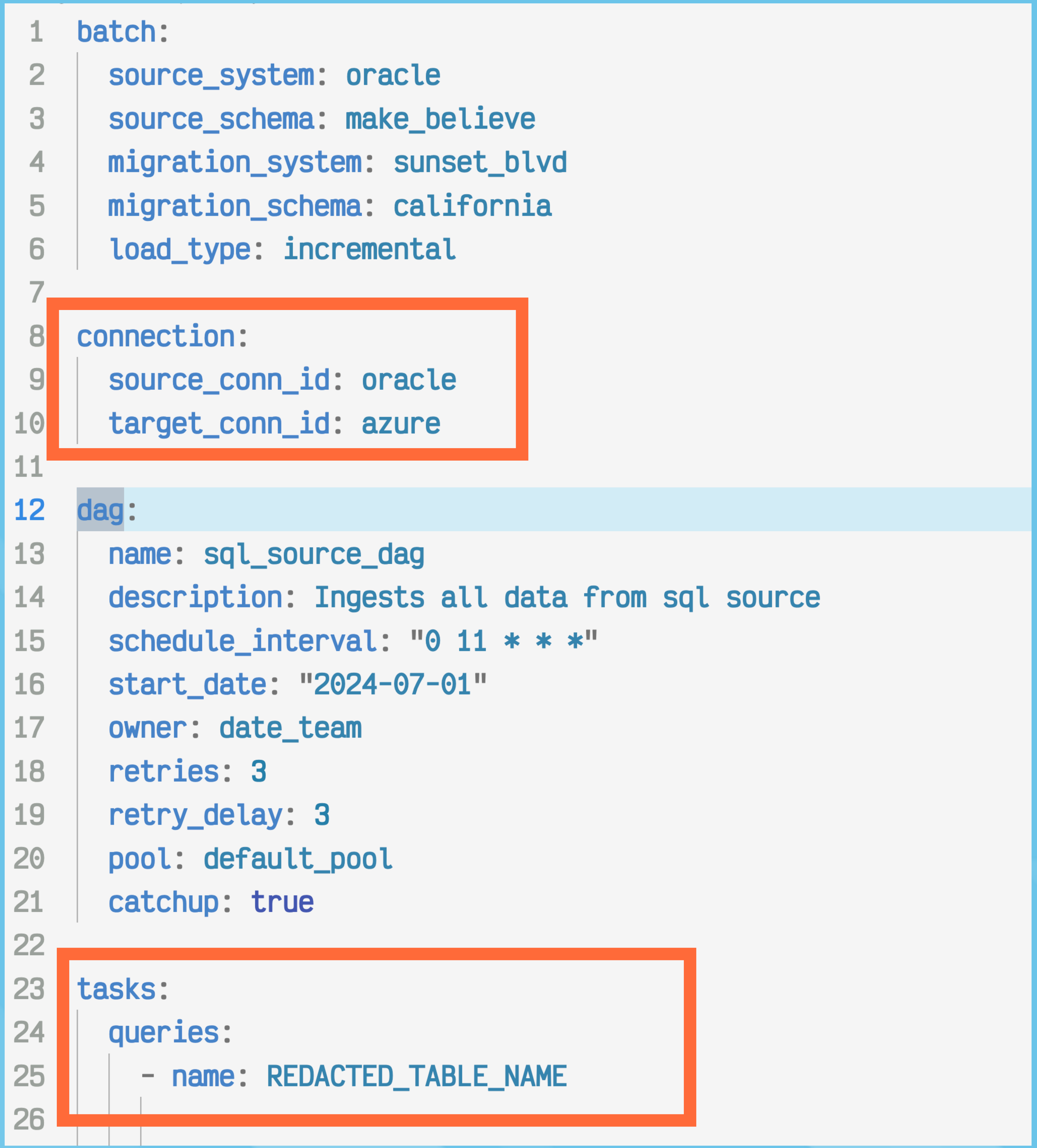
DAG

Task

Operator

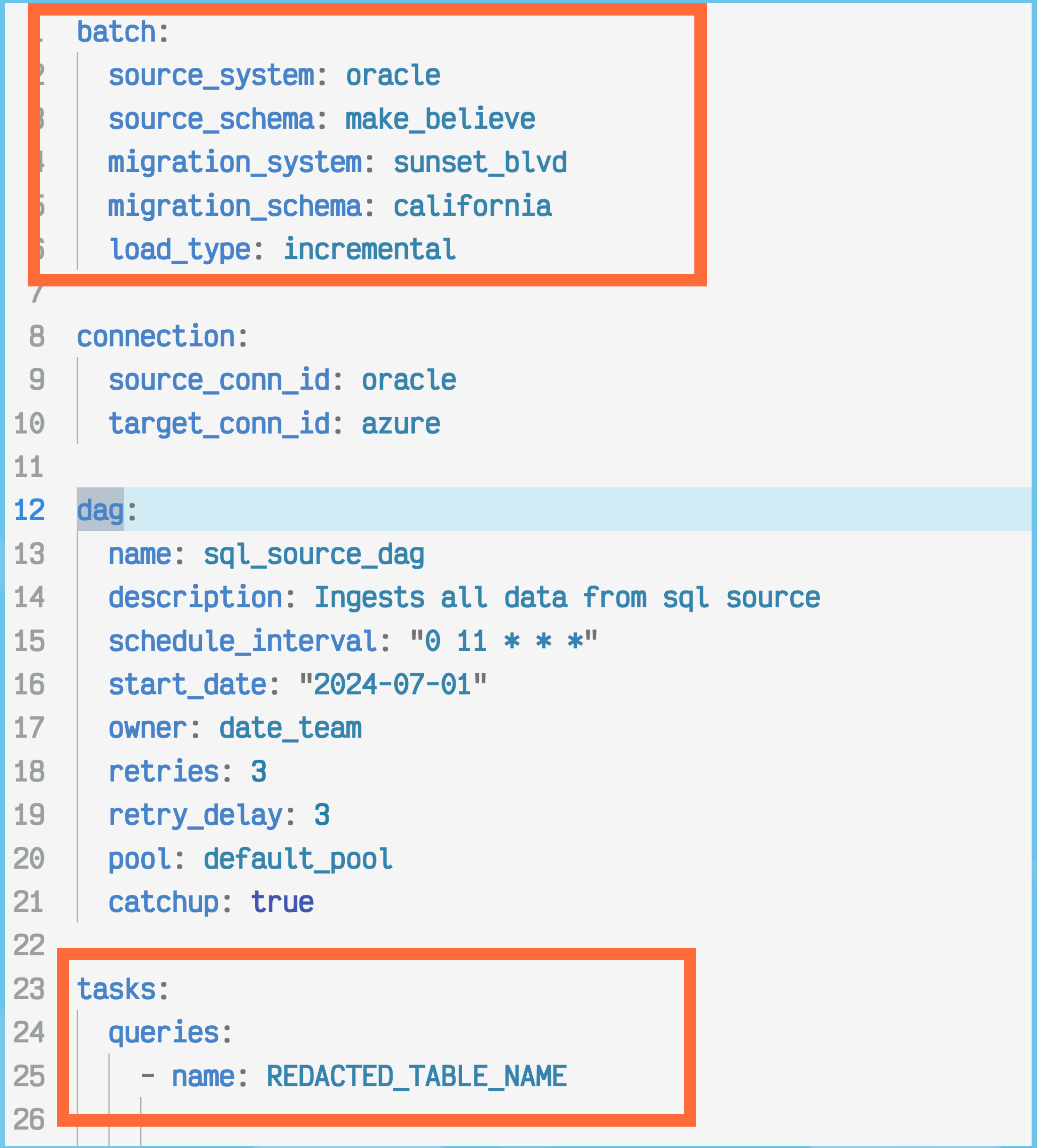
Hook

Other Code



```yaml
batch:
  source_system: oracle
  source_schema: make_believe
  migration_system: sunset_blvd
  migration_schema: california
  load_type: incremental

connection:
  source_conn_id: oracle
  target_conn_id: azure

dag:
  name: sql_source_dag
  description: Ingests all data from sql source
  schedule_interval: "0 11 * * *"
  start_date: "2024-07-01"
  owner: date_team
  retries: 3
  retry_delay: 3
  pool: default_pool
  catchup: true

tasks:
  queries:
    - name: REDACTED_TABLE_NAME

```

GENERATOR PATTERN



```yaml
batch:
  source_system: oracle
  source_schema: make_believe
  migration_system: sunset_blvd
  migration_schema: california
  load_type: incremental

connection:
  source_conn_id: oracle
  target_conn_id: azure

dag:
  name: sql_source_dag
  description: Ingests all data from sql source
  schedule_interval: "0 11 * * *"
  start_date: "2024-07-01"
  owner: date_team
  retries: 3
  retry_delay: 3
  pool: default_pool
  catchup: true

tasks:
  queries:
    - name: REDACTED_TABLE_NAME

```

GENERATOR PATTERN

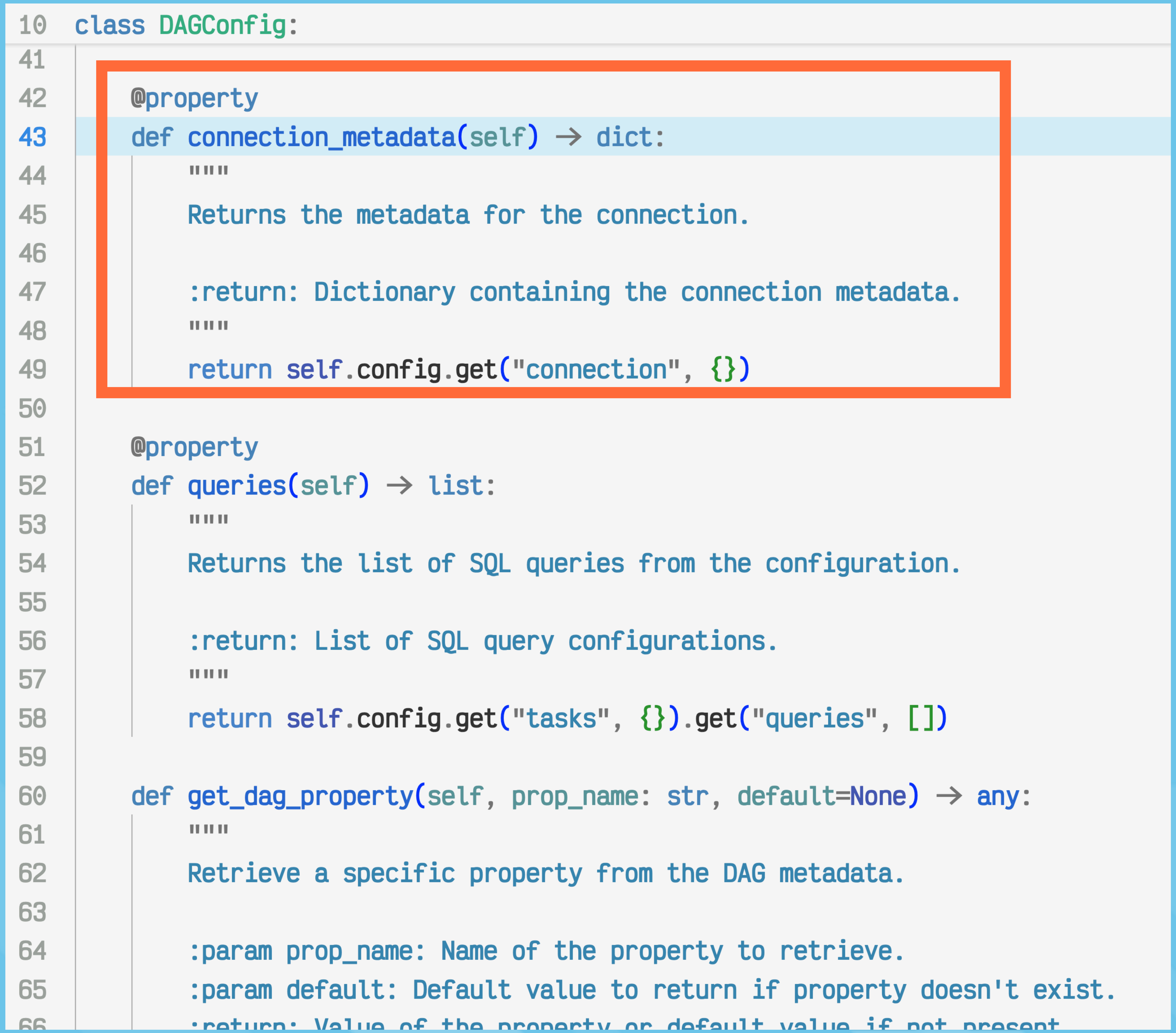
DAG

Task

Operator

Hook

Other Code



```python
class DAGConfig:

    @property
    def connection_metadata(self) → dict:
        """
        Returns the metadata for the connection.

        :return: Dictionary containing the connection metadata.
        """
        return self.config.get("connection", {})

    @property
    def queries(self) → list:
        """
        Returns the list of SQL queries from the configuration.

        :return: List of SQL query configurations.
        """
        return self.config.get("tasks", {}).get("queries", [])

    def get_dag_property(self, prop_name: str, default=None) → any:
        """
        Retrieve a specific property from the DAG metadata.

        :param prop_name: Name of the property to retrieve.
        :param default: Default value to return if property doesn't exist.
        :return: Value of the property or default value if not present
```

GENERATOR PATTERN

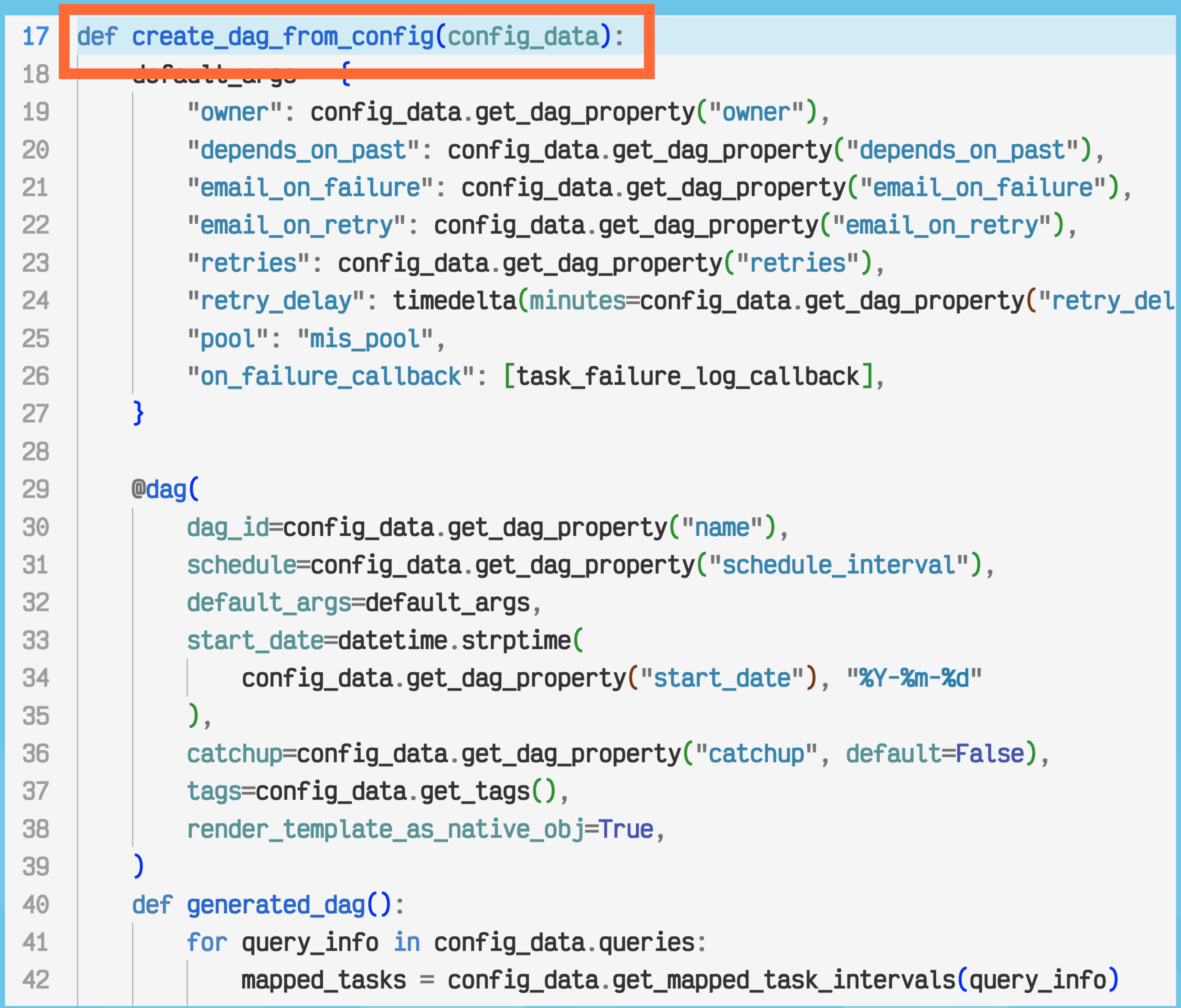
DAG

Task

Operator

Hook

Other Code



```python
def create_dag_from_config(config_data):
    default_args = {
        "owner": config_data.get_dag_property("owner"),
        "depends_on_past": config_data.get_dag_property("depends_on_past"),
        "email_on_failure": config_data.get_dag_property("email_on_failure"),
        "email_on_retry": config_data.get_dag_property("email_on_retry"),
        "retries": config_data.get_dag_property("retries"),
        "retry_delay": timedelta(minutes=config_data.get_dag_property("retry_del
        "pool": "mis_pool",
        "on_failure_callback": [task_failure_log_callback],
    }

    @dag(
        dag_id=config_data.get_dag_property("name"),
        schedule=config_data.get_dag_property("schedule_interval"),
        default_args=default_args,
        start_date=datetime.strptime(
            config_data.get_dag_property("start_date"), "%Y-%m-%d"
        ),
        catchup=config_data.get_dag_property("catchup", default=False),
        tags=config_data.get_tags(),
        render_template_as_native_obj=True,
    )
    def generated_dag():
        for query_info in config_data.queries:
            mapped_tasks = config_data.get_mapped_task_intervals(query_info)
```

GENERATOR PATTERN

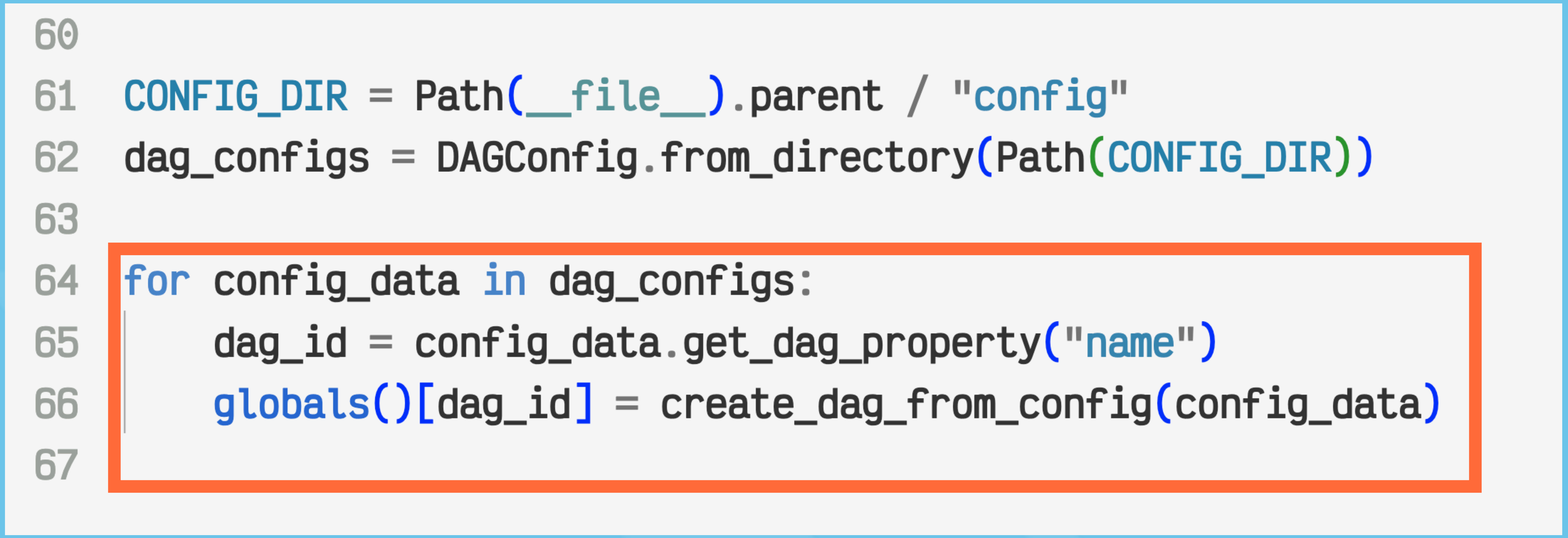
DAG

Task

Operator

Hook

Other Code



```python

CONFIG_DIR = Path(__file__).parent / "config"
dag_configs = DAGConfig.from_directory(Path(CONFIG_DIR))

for config_data in dag_configs:
    dag_id = config_data.get_dag_property("name")
    globals()[dag_id] = create_dag_from_config(config_data)

```

GENERATOR PATTERN

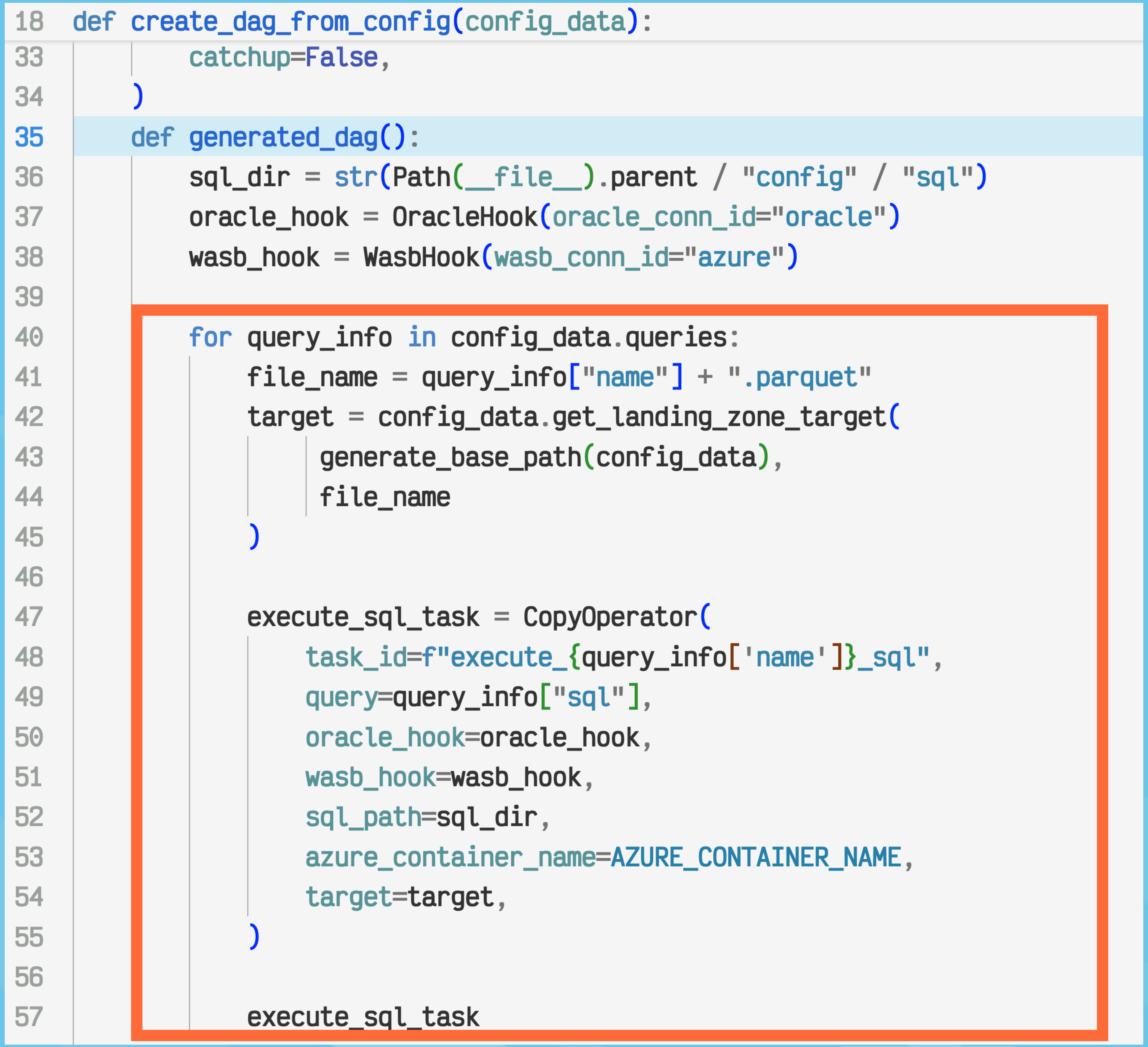
DAG

Task

Operator

Hook

Other Code



```python
def create_dag_from_config(config_data):
        catchup=False,
    )
    def generated_dag():
        sql_dir = str(Path(__file__).parent / "config" / "sql")
        oracle_hook = OracleHook(oracle_conn_id="oracle")
        wasb_hook = WasbHook(wasb_conn_id="azure")

        for query_info in config_data.queries:
            file_name = query_info["name"] + ".parquet"
            target = config_data.get_landing_zone_target(
                generate_base_path(config_data),
                file_name
            )

            execute_sql_task = CopyOperator(
                task_id=f"execute_{query_info['name']}_sql",
                query=query_info["sql"],
                oracle_hook=oracle_hook,
                wasb_hook=wasb_hook,
                sql_path=sql_dir,
                azure_container_name=AZURE_CONTAINER_NAME,
                target=target,
            )

            execute_sql_task
```

GENERATOR PATTERN

30 MPH

"LOCAL" DEVELOPMENT

deployToSandbox

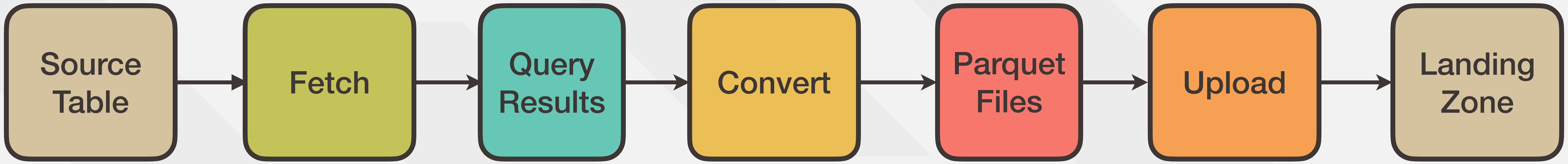
“LOCAL” DEVELOPMENT

```
git commit -m “I hope this works”

git tag -f deployToSandbox

git push -f deployToSandbox
```

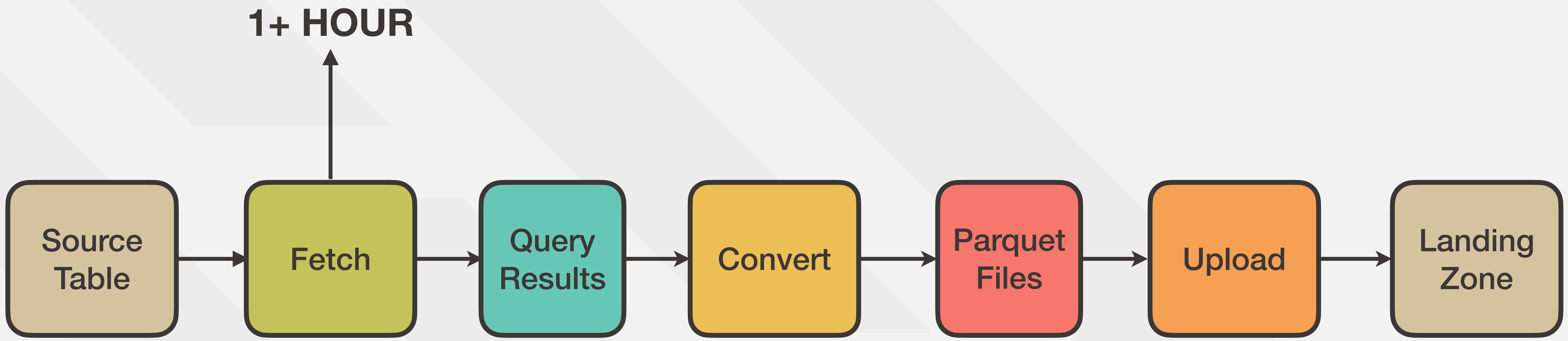
“LOCAL” DEVELOPMENT

50 MPH

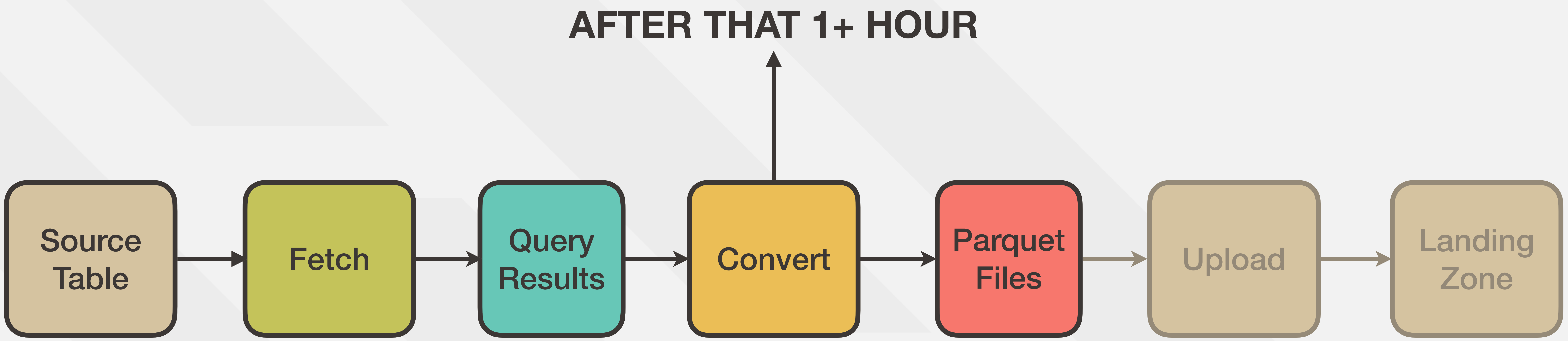
CHUNKING AND PARALLELIZATION



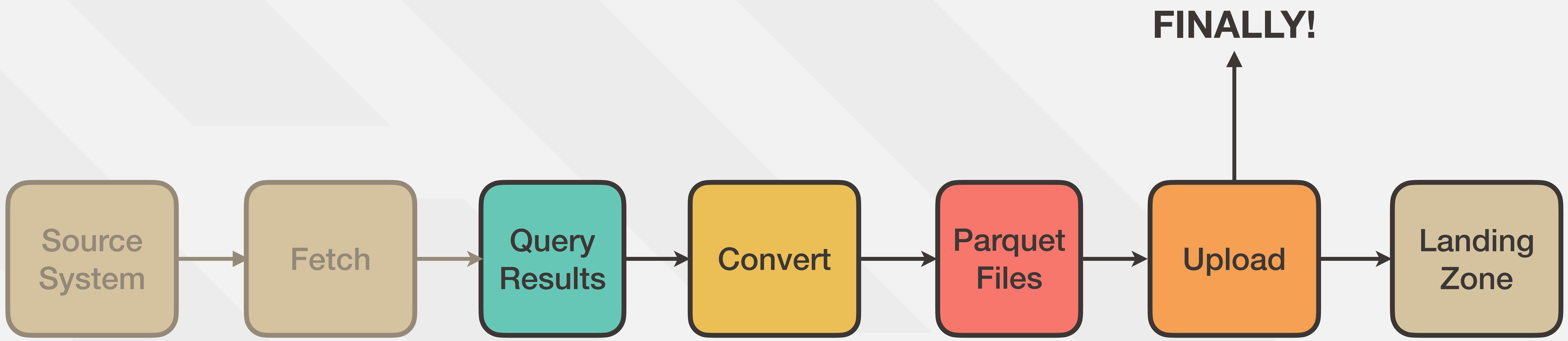
CHUNKING AND PARALLELIZATION



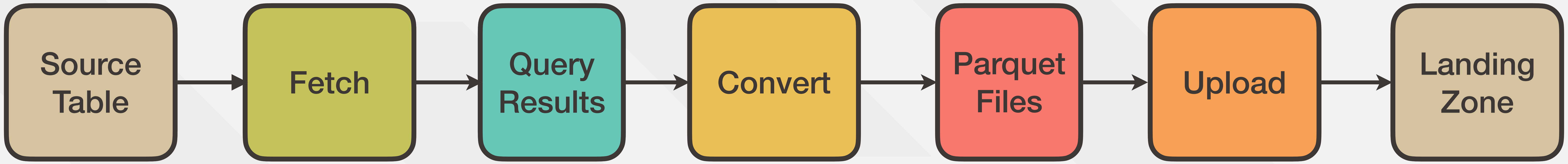
CHUNKING AND PARALLELIZATION



CHUNKING AND PARALLELIZATION



CHUNKING AND PARALLELIZATION



CHUNKING AND PARALLELIZATION

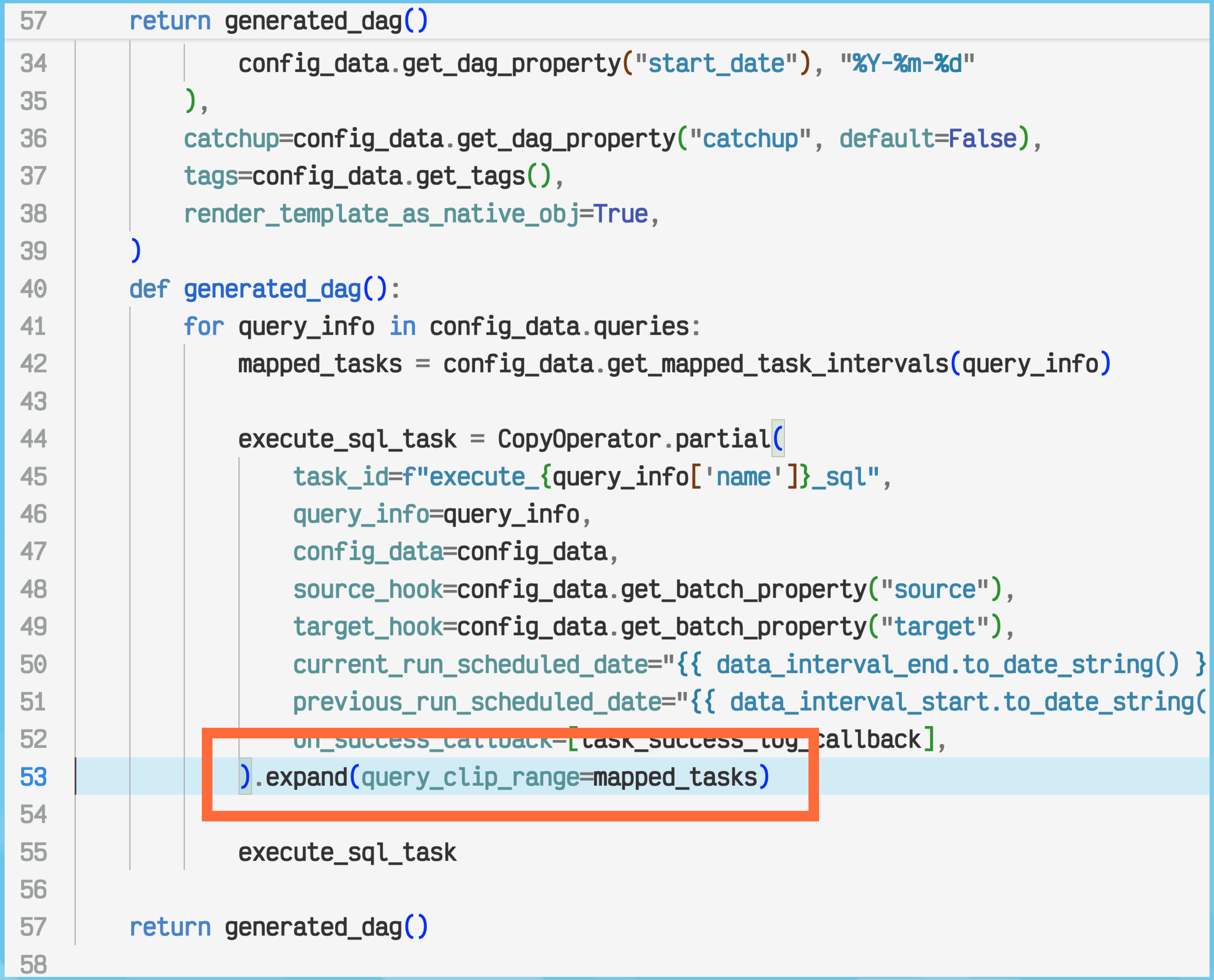
DAG

Task

Operator

Hook

Other Code



```
    return generated_dag()
            config_data.get_dag_property("start_date"), "%Y-%m-%d"
        ),
        catchup=config_data.get_dag_property("catchup", default=False),
        tags=config_data.get_tags(),
        render_template_as_native_obj=True,
    )
    def generated_dag():
        for query_info in config_data.queries:
            mapped_tasks = config_data.get_mapped_task_intervals(query_info)

            execute_sql_task = CopyOperator.partial(
                task_id=f"execute_{query_info['name']}_sql",
                query_info=query_info,
                config_data=config_data,
                source_hook=config_data.get_batch_property("source"),
                target_hook=config_data.get_batch_property("target"),
                current_run_scheduled_date="{{ data_interval_end.to_date_string() }
                previous_run_scheduled_date="{{ data_interval_start.to_date_string(
                on_success_callback=[task_success_log_callback],
            ).expand(query_clip_range=mapped_tasks)

            execute_sql_task

    return generated_dag()
```

CHUNKING AND PARALLELIZATION

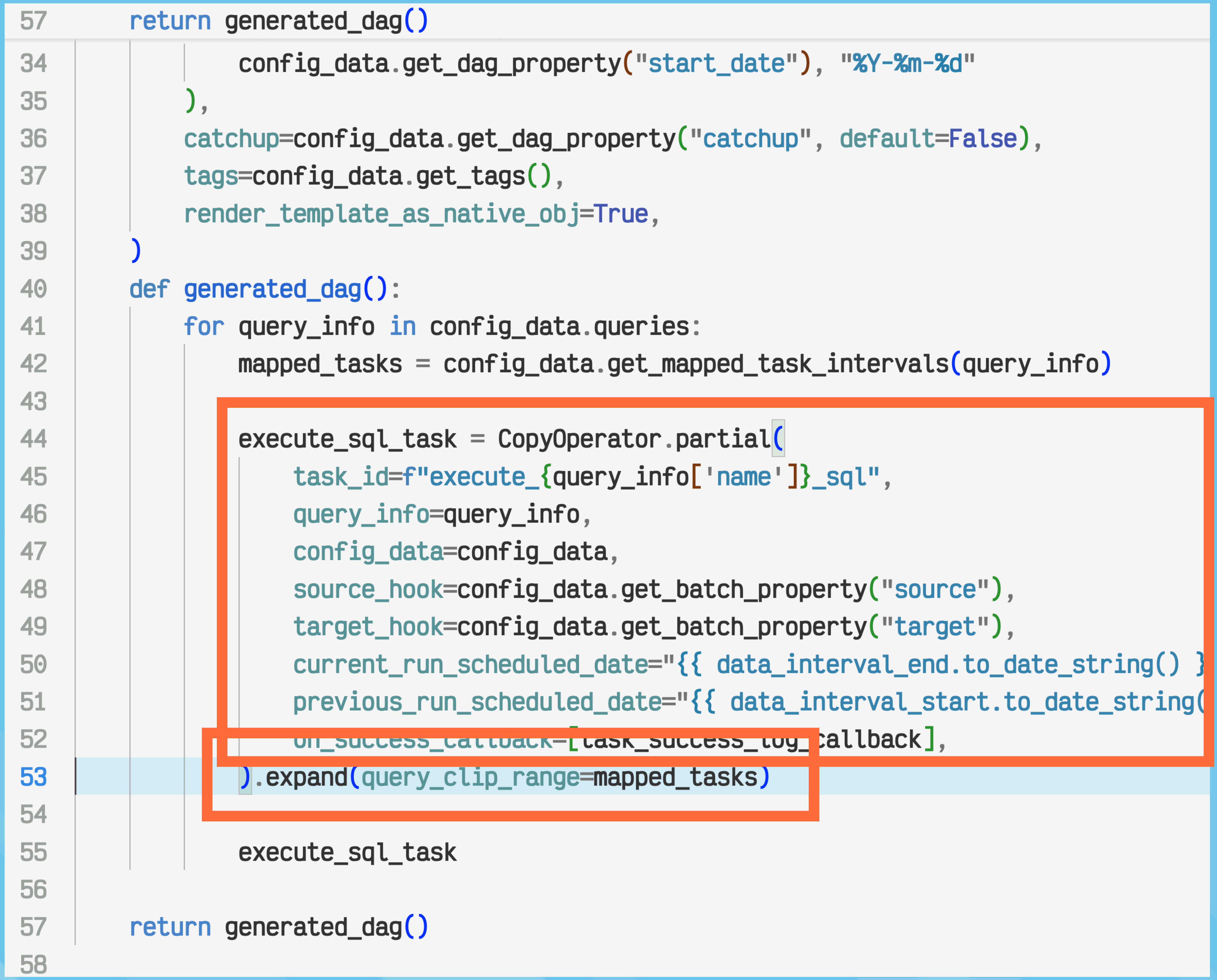
DAG

Task

Operator

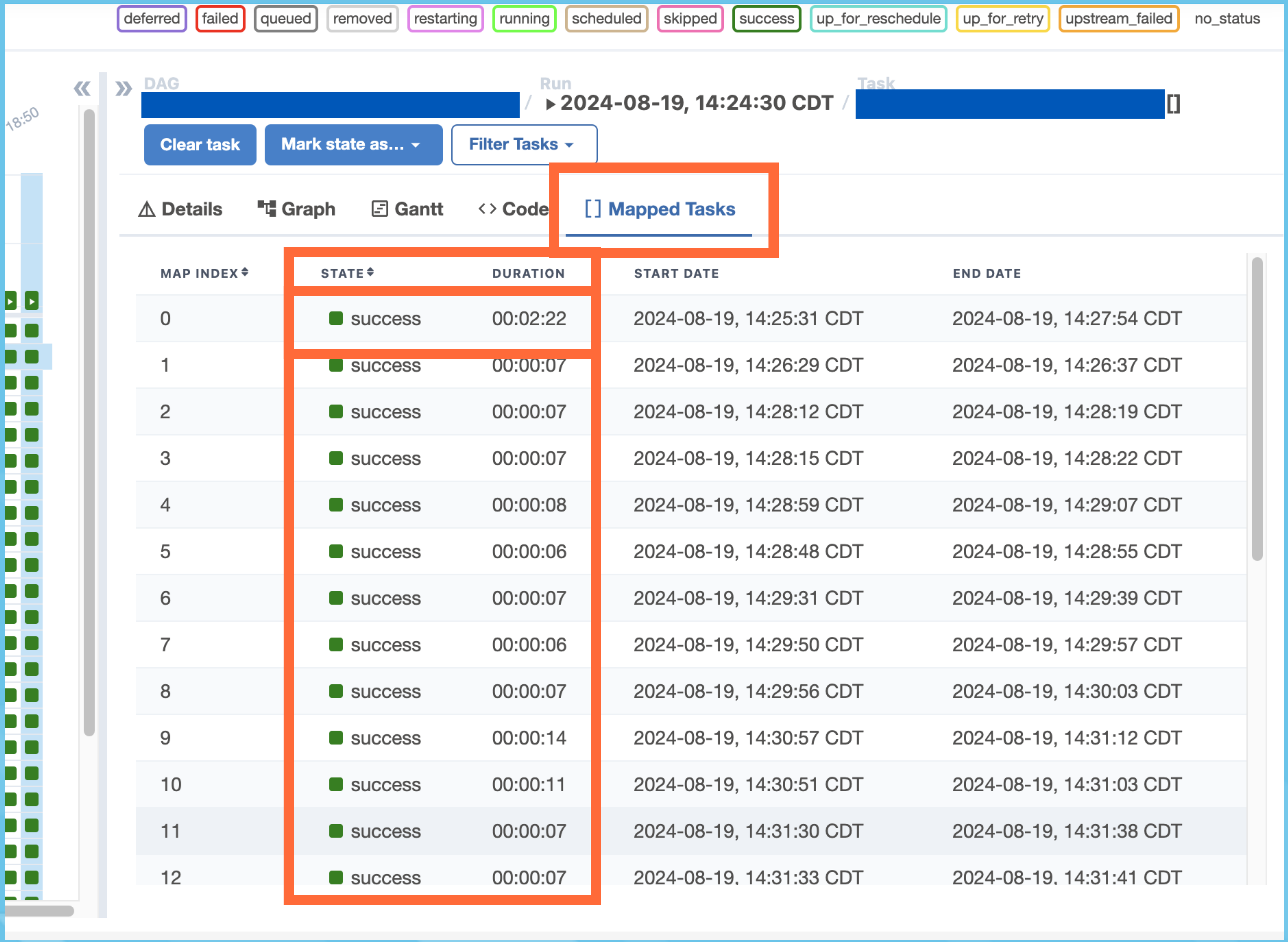
Hook

Other Code



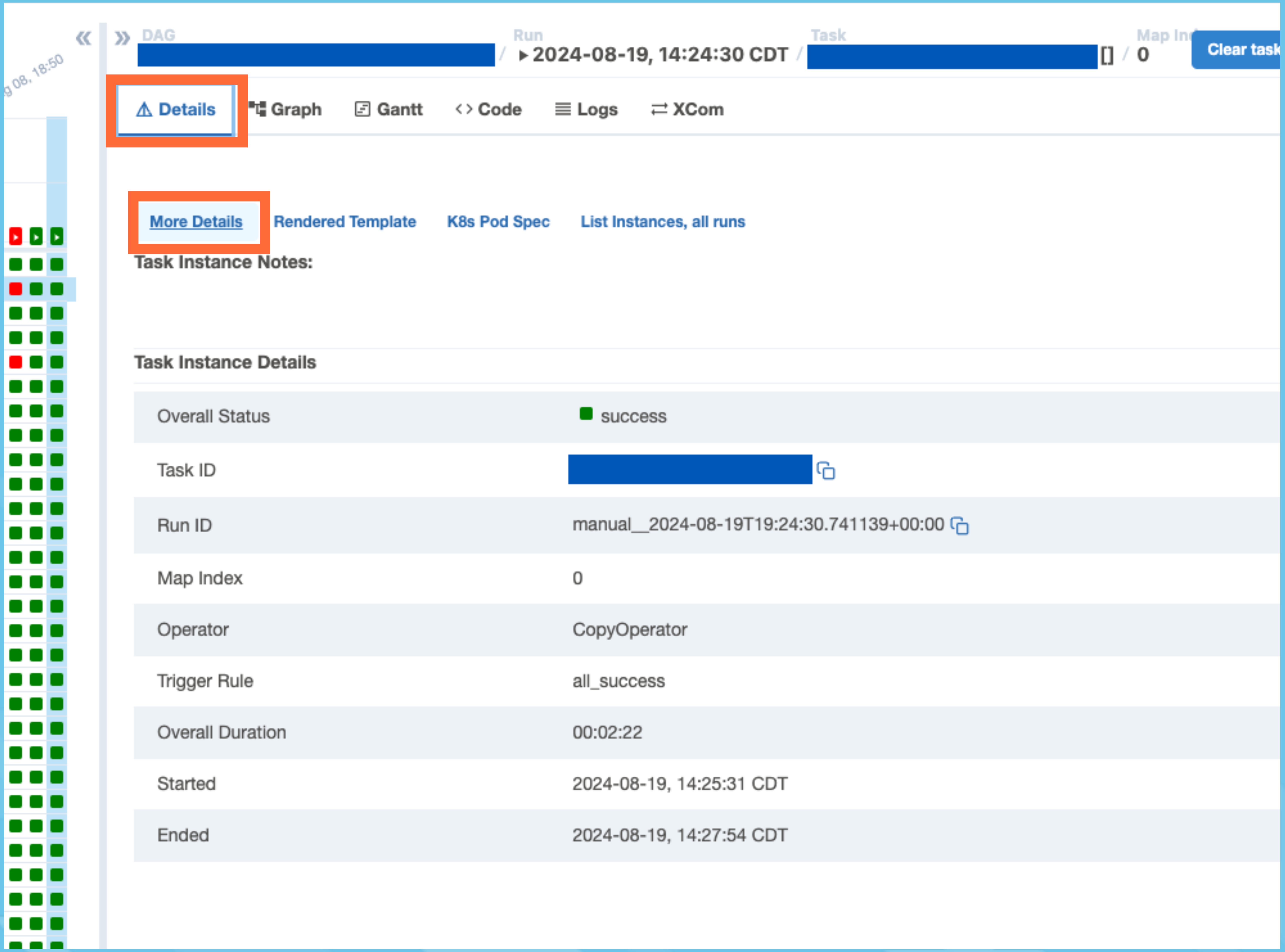
```
    return generated_dag()
            config_data.get_dag_property("start_date"), "%Y-%m-%d"
        ),
        catchup=config_data.get_dag_property("catchup", default=False),
        tags=config_data.get_tags(),
        render_template_as_native_obj=True,
    )
    def generated_dag():
        for query_info in config_data.queries:
            mapped_tasks = config_data.get_mapped_task_intervals(query_info)

            execute_sql_task = CopyOperator.partial(
                task_id=f"execute_{query_info['name']}_sql",
                query_info=query_info,
                config_data=config_data,
                source_hook=config_data.get_batch_property("source"),
                target_hook=config_data.get_batch_property("target"),
                current_run_scheduled_date="{{ data_interval_end.to_date_string() }
                previous_run_scheduled_date="{{ data_interval_start.to_date_string(
                on_success_callback=[task_success_log_callback],
            ).expand(query_clip_range=mapped_tasks)

            execute_sql_task

    return generated_dag()
```

CHUNKING AND PARALLELIZATION



MAP INDEX	STATE	DURATION	START DATE	END DATE
0	🟩 success	00:02:22	2024-08-19, 14:25:31 CDT	2024-08-19, 14:27:54 CDT
1	🟩 success	00:00:07	2024-08-19, 14:26:29 CDT	2024-08-19, 14:26:37 CDT
2	🟩 success	00:00:07	2024-08-19, 14:28:12 CDT	2024-08-19, 14:28:19 CDT
3	🟩 success	00:00:07	2024-08-19, 14:28:15 CDT	2024-08-19, 14:28:22 CDT
4	🟩 success	00:00:08	2024-08-19, 14:28:59 CDT	2024-08-19, 14:29:07 CDT
5	🟩 success	00:00:06	2024-08-19, 14:28:48 CDT	2024-08-19, 14:28:55 CDT
6	🟩 success	00:00:07	2024-08-19, 14:29:31 CDT	2024-08-19, 14:29:39 CDT
7	🟩 success	00:00:06	2024-08-19, 14:29:50 CDT	2024-08-19, 14:29:57 CDT
8	🟩 success	00:00:07	2024-08-19, 14:29:56 CDT	2024-08-19, 14:30:03 CDT
9	🟩 success	00:00:14	2024-08-19, 14:30:57 CDT	2024-08-19, 14:31:12 CDT
10	🟩 success	00:00:11	2024-08-19, 14:30:51 CDT	2024-08-19, 14:31:03 CDT
11	🟩 success	00:00:07	2024-08-19, 14:31:30 CDT	2024-08-19, 14:31:38 CDT
12	🟩 success	00:00:07	2024-08-19, 14:31:33 CDT	2024-08-19, 14:31:41 CDT

CHUNKING AND PARALLELIZATION



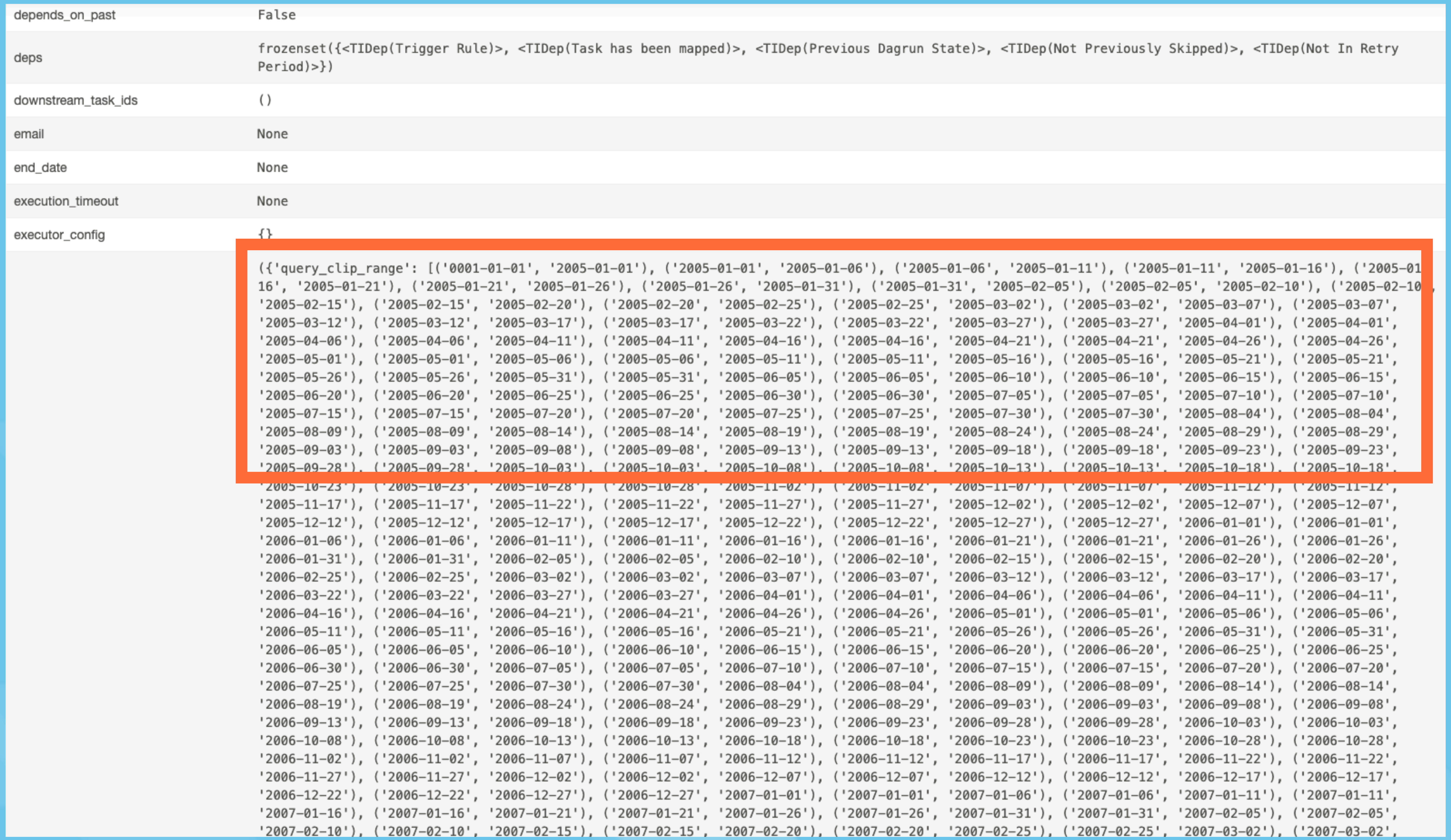
DAG [redacted] / ▸ 2024-08-19, 14:24:30 CDT / Task [redacted] [] / 0

Task Instance Notes:

Task Instance Details

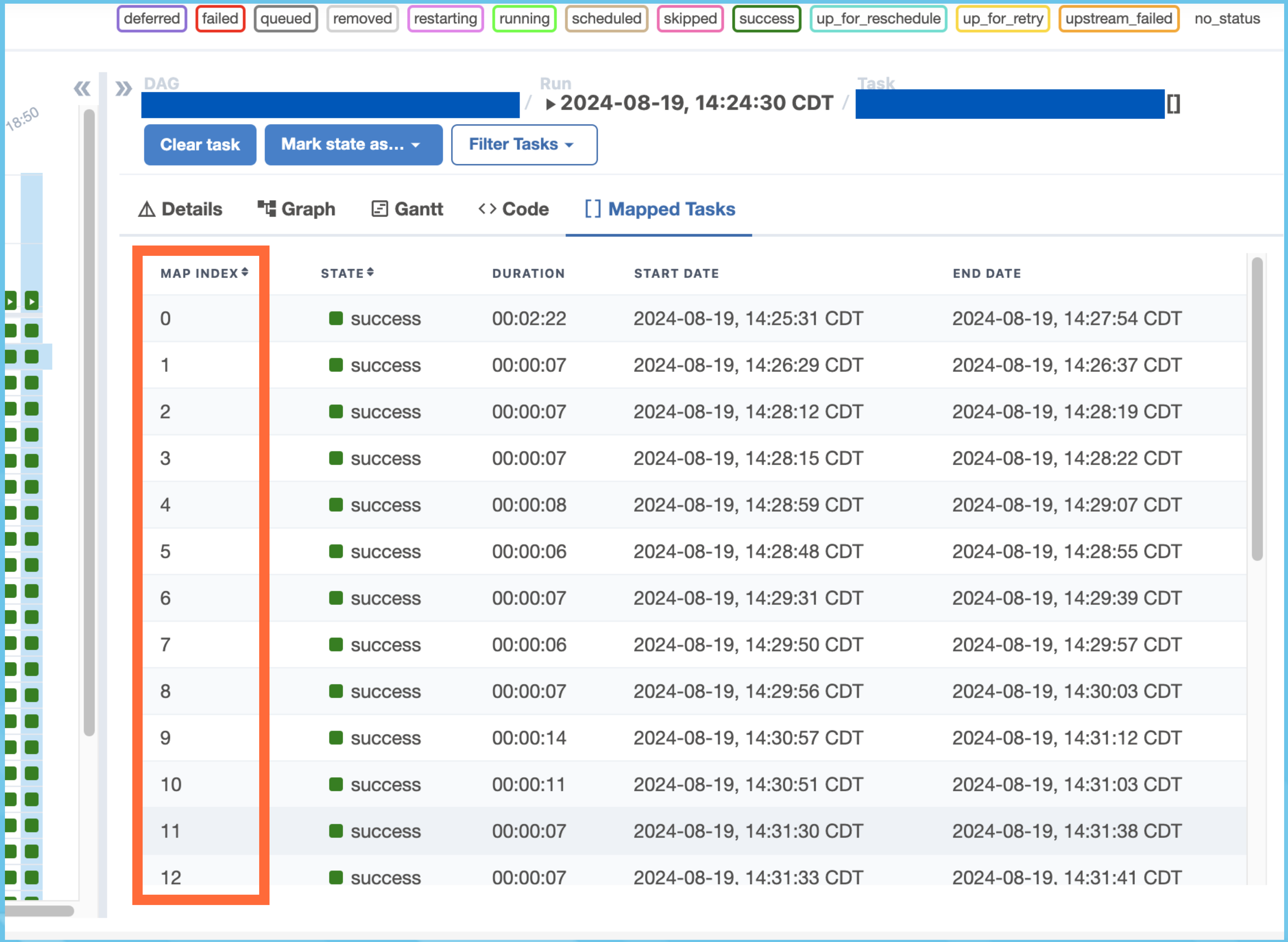
Overall Status	🟩 success
Task ID	[redacted]
Run ID	manual__2024-08-19T19:24:30.741139+00:00
Map Index	0
Operator	CopyOperator
Trigger Rule	all_success
Overall Duration	00:02:22
Started	2024-08-19, 14:25:31 CDT
Ended	2024-08-19, 14:27:54 CDT

CHUNKING AND PARALLELIZATION



```
depends_on_past       False
deps                  frozenset({<TIDep(Trigger Rule)>, <TIDep(Task has been mapped)>, <TIDep(Previous Dagrun State)>, <TIDep(Not Previously Skipped)>, <TIDep(Not In Retry
                      Period)>})
downstream_task_ids   ()
email                 None
end_date              None
execution_timeout     None
executor_config       {}
                      ({'query_clip_range': [('0001-01-01', '2005-01-01'), ('2005-01-01', '2005-01-06'), ('2005-01-06', '2005-01-11'), ('2005-01-11', '2005-01-16'), ('2005-01
                      16', '2005-01-21'), ('2005-01-21', '2005-01-26'), ('2005-01-26', '2005-01-31'), ('2005-01-31', '2005-02-05'), ('2005-02-05', '2005-02-10'), ('2005-02-10',
                      '2005-02-15'), ('2005-02-15', '2005-02-20'), ('2005-02-20', '2005-02-25'), ('2005-02-25', '2005-03-02'), ('2005-03-02', '2005-03-07'), ('2005-03-07',
                      '2005-03-12'), ('2005-03-12', '2005-03-17'), ('2005-03-17', '2005-03-22'), ('2005-03-22', '2005-03-27'), ('2005-03-27', '2005-04-01'), ('2005-04-01',
                      '2005-04-06'), ('2005-04-06', '2005-04-11'), ('2005-04-11', '2005-04-16'), ('2005-04-16', '2005-04-21'), ('2005-04-21', '2005-04-26'), ('2005-04-26',
                      '2005-05-01'), ('2005-05-01', '2005-05-06'), ('2005-05-06', '2005-05-11'), ('2005-05-11', '2005-05-16'), ('2005-05-16', '2005-05-21'), ('2005-05-21',
                      '2005-05-26'), ('2005-05-26', '2005-05-31'), ('2005-05-31', '2005-06-05'), ('2005-06-05', '2005-06-10'), ('2005-06-10', '2005-06-15'), ('2005-06-15',
                      '2005-06-20'), ('2005-06-20', '2005-06-25'), ('2005-06-25', '2005-06-30'), ('2005-06-30', '2005-07-05'), ('2005-07-05', '2005-07-10'), ('2005-07-10',
                      '2005-07-15'), ('2005-07-15', '2005-07-20'), ('2005-07-20', '2005-07-25'), ('2005-07-25', '2005-07-30'), ('2005-07-30', '2005-08-04'), ('2005-08-04',
                      '2005-08-09'), ('2005-08-09', '2005-08-14'), ('2005-08-14', '2005-08-19'), ('2005-08-19', '2005-08-24'), ('2005-08-24', '2005-08-29'), ('2005-08-29',
                      '2005-09-03'), ('2005-09-03', '2005-09-08'), ('2005-09-08', '2005-09-13'), ('2005-09-13', '2005-09-18'), ('2005-09-18', '2005-09-23'), ('2005-09-23',
                      '2005-09-28'), ('2005-09-28', '2005-10-03'), ('2005-10-03', '2005-10-08'), ('2005-10-08', '2005-10-13'), ('2005-10-13', '2005-10-18'), ('2005-10-18',
                      '2005-10-23'), ('2005-10-23', '2005-10-28'), ('2005-10-28', '2005-11-02'), ('2005-11-02', '2005-11-07'), ('2005-11-07', '2005-11-12'), ('2005-11-12',
                      '2005-11-17'), ('2005-11-17', '2005-11-22'), ('2005-11-22', '2005-11-27'), ('2005-11-27', '2005-12-02'), ('2005-12-02', '2005-12-07'), ('2005-12-07',
                      '2005-12-12'), ('2005-12-12', '2005-12-17'), ('2005-12-17', '2005-12-22'), ('2005-12-22', '2005-12-27'), ('2005-12-27', '2006-01-01'), ('2006-01-01',
                      '2006-01-06'), ('2006-01-06', '2006-01-11'), ('2006-01-11', '2006-01-16'), ('2006-01-16', '2006-01-21'), ('2006-01-21', '2006-01-26'), ('2006-01-26',
                      '2006-01-31'), ('2006-01-31', '2006-02-05'), ('2006-02-05', '2006-02-10'), ('2006-02-10', '2006-02-15'), ('2006-02-15', '2006-02-20'), ('2006-02-20',
                      '2006-02-25'), ('2006-02-25', '2006-03-02'), ('2006-03-02', '2006-03-07'), ('2006-03-07', '2006-03-12'), ('2006-03-12', '2006-03-17'), ('2006-03-17',
                      '2006-03-22'), ('2006-03-22', '2006-03-27'), ('2006-03-27', '2006-04-01'), ('2006-04-01', '2006-04-06'), ('2006-04-06', '2006-04-11'), ('2006-04-11',
                      '2006-04-16'), ('2006-04-16', '2006-04-21'), ('2006-04-21', '2006-04-26'), ('2006-04-26', '2006-05-01'), ('2006-05-01', '2006-05-06'), ('2006-05-06',
                      '2006-05-11'), ('2006-05-11', '2006-05-16'), ('2006-05-16', '2006-05-21'), ('2006-05-21', '2006-05-26'), ('2006-05-26', '2006-05-31'), ('2006-05-31',
                      '2006-06-05'), ('2006-06-05', '2006-06-10'), ('2006-06-10', '2006-06-15'), ('2006-06-15', '2006-06-20'), ('2006-06-20', '2006-06-25'), ('2006-06-25',
                      '2006-06-30'), ('2006-06-30', '2006-07-05'), ('2006-07-05', '2006-07-10'), ('2006-07-10', '2006-07-15'), ('2006-07-15', '2006-07-20'), ('2006-07-20',
                      '2006-07-25'), ('2006-07-25', '2006-07-30'), ('2006-07-30', '2006-08-04'), ('2006-08-04', '2006-08-09'), ('2006-08-09', '2006-08-14'), ('2006-08-14',
                      '2006-08-19'), ('2006-08-19', '2006-08-24'), ('2006-08-24', '2006-08-29'), ('2006-08-29', '2006-09-03'), ('2006-09-03', '2006-09-08'), ('2006-09-08',
                      '2006-09-13'), ('2006-09-13', '2006-09-18'), ('2006-09-18', '2006-09-23'), ('2006-09-23', '2006-09-28'), ('2006-09-28', '2006-10-03'), ('2006-10-03',
                      '2006-10-08'), ('2006-10-08', '2006-10-13'), ('2006-10-13', '2006-10-18'), ('2006-10-18', '2006-10-23'), ('2006-10-23', '2006-10-28'), ('2006-10-28',
                      '2006-11-02'), ('2006-11-02', '2006-11-07'), ('2006-11-07', '2006-11-12'), ('2006-11-12', '2006-11-17'), ('2006-11-17', '2006-11-22'), ('2006-11-22',
                      '2006-11-27'), ('2006-11-27', '2006-12-02'), ('2006-12-02', '2006-12-07'), ('2006-12-07', '2006-12-12'), ('2006-12-12', '2006-12-17'), ('2006-12-17',
                      '2006-12-22'), ('2006-12-22', '2006-12-27'), ('2006-12-27', '2007-01-01'), ('2007-01-01', '2007-01-06'), ('2007-01-06', '2007-01-11'), ('2007-01-11',
                      '2007-01-16'), ('2007-01-16', '2007-01-21'), ('2007-01-21', '2007-01-26'), ('2007-01-26', '2007-01-31'), ('2007-01-31', '2007-02-05'), ('2007-02-05',
                      '2007-02-10'), ('2007-02-10', '2007-02-15'), ('2007-02-15', '2007-02-20'), ('2007-02-20', '2007-02-25'), ('2007-02-25', '2007-03-02'), ('2007-03-02',
```

CHUNKING AND PARALLELIZATION

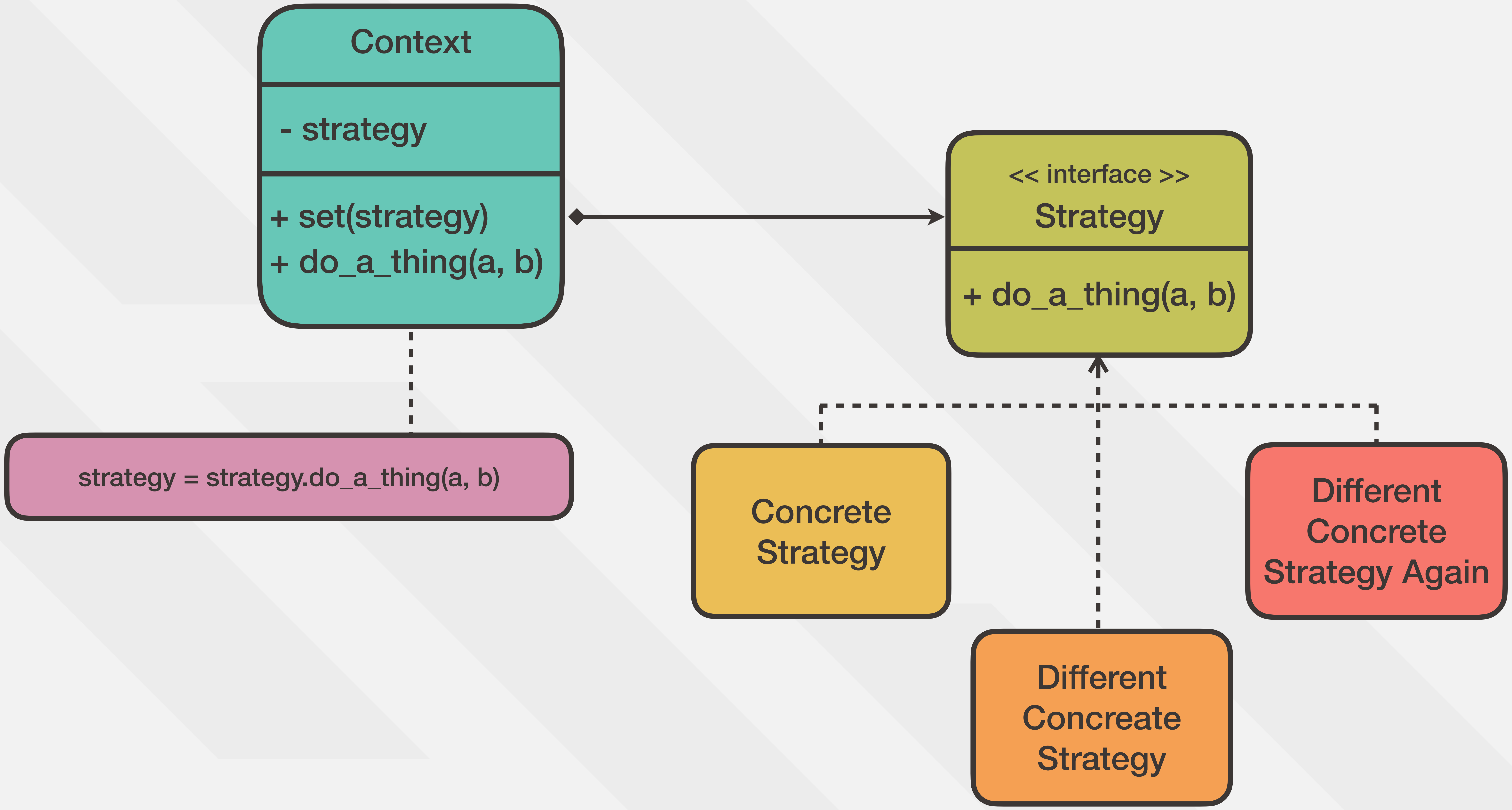


DAG [redacted] / Run ▸ 2024-08-19, 14:24:30 CDT / Task [redacted] []

MAP INDEX	STATE	DURATION	START DATE	END DATE
0	🟩 success	00:02:22	2024-08-19, 14:25:31 CDT	2024-08-19, 14:27:54 CDT
1	🟩 success	00:00:07	2024-08-19, 14:26:29 CDT	2024-08-19, 14:26:37 CDT
2	🟩 success	00:00:07	2024-08-19, 14:28:12 CDT	2024-08-19, 14:28:19 CDT
3	🟩 success	00:00:07	2024-08-19, 14:28:15 CDT	2024-08-19, 14:28:22 CDT
4	🟩 success	00:00:08	2024-08-19, 14:28:59 CDT	2024-08-19, 14:29:07 CDT
5	🟩 success	00:00:06	2024-08-19, 14:28:48 CDT	2024-08-19, 14:28:55 CDT
6	🟩 success	00:00:07	2024-08-19, 14:29:31 CDT	2024-08-19, 14:29:39 CDT
7	🟩 success	00:00:06	2024-08-19, 14:29:50 CDT	2024-08-19, 14:29:57 CDT
8	🟩 success	00:00:07	2024-08-19, 14:29:56 CDT	2024-08-19, 14:30:03 CDT
9	🟩 success	00:00:14	2024-08-19, 14:30:57 CDT	2024-08-19, 14:31:12 CDT
10	🟩 success	00:00:11	2024-08-19, 14:30:51 CDT	2024-08-19, 14:31:03 CDT
11	🟩 success	00:00:07	2024-08-19, 14:31:30 CDT	2024-08-19, 14:31:38 CDT
12	🟩 success	00:00:07	2024-08-19, 14:31:33 CDT	2024-08-19, 14:31:41 CDT

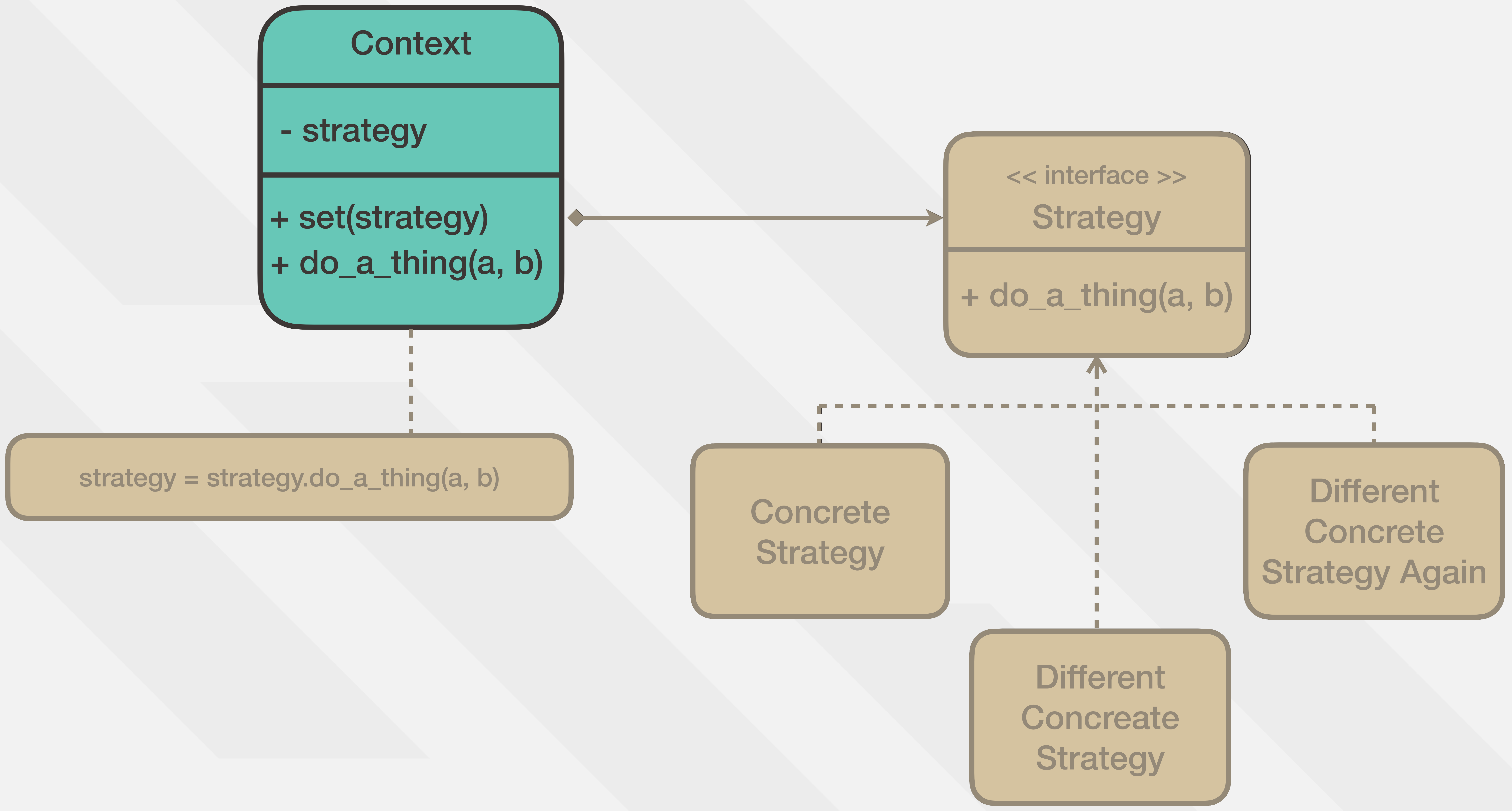
CHUNKING AND PARALLELIZATION

STRATEGY PATTERN



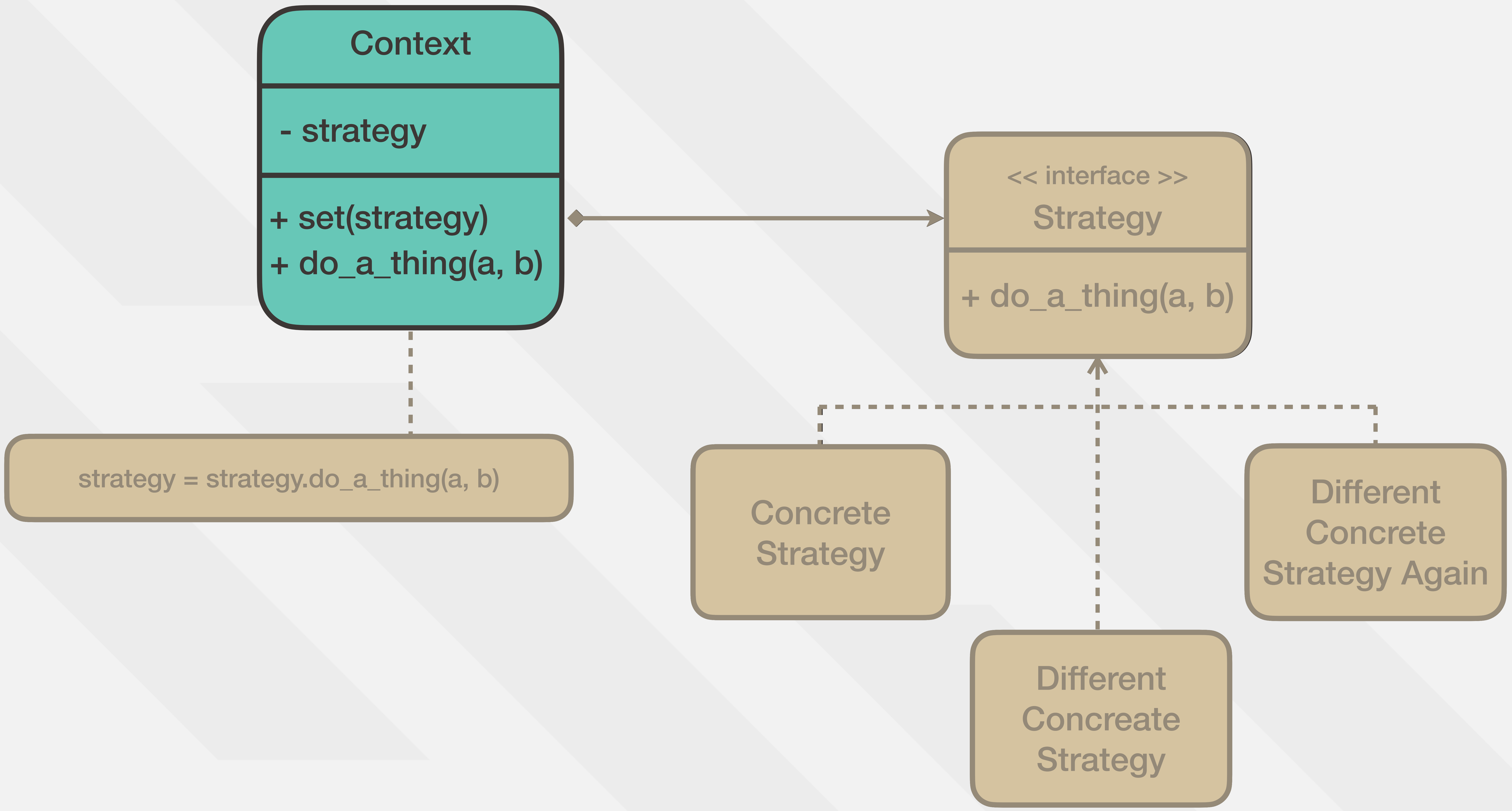
Adapted from
https://refactoring.guru/design-patterns/strategy

STRATEGY PATTERN



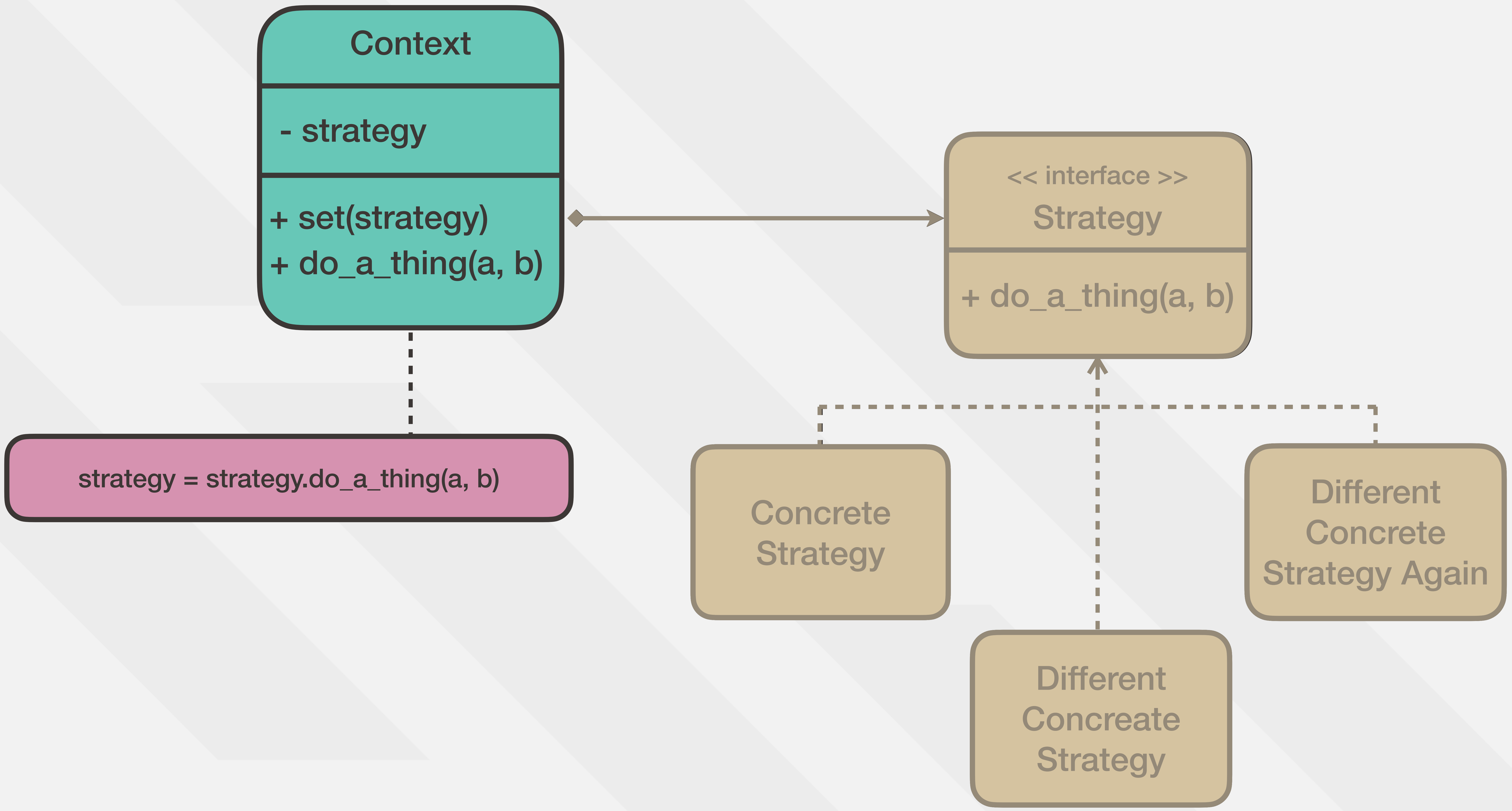
Adapted from
https://refactoring.guru/design-patterns/strategy

STRATEGY PATTERN



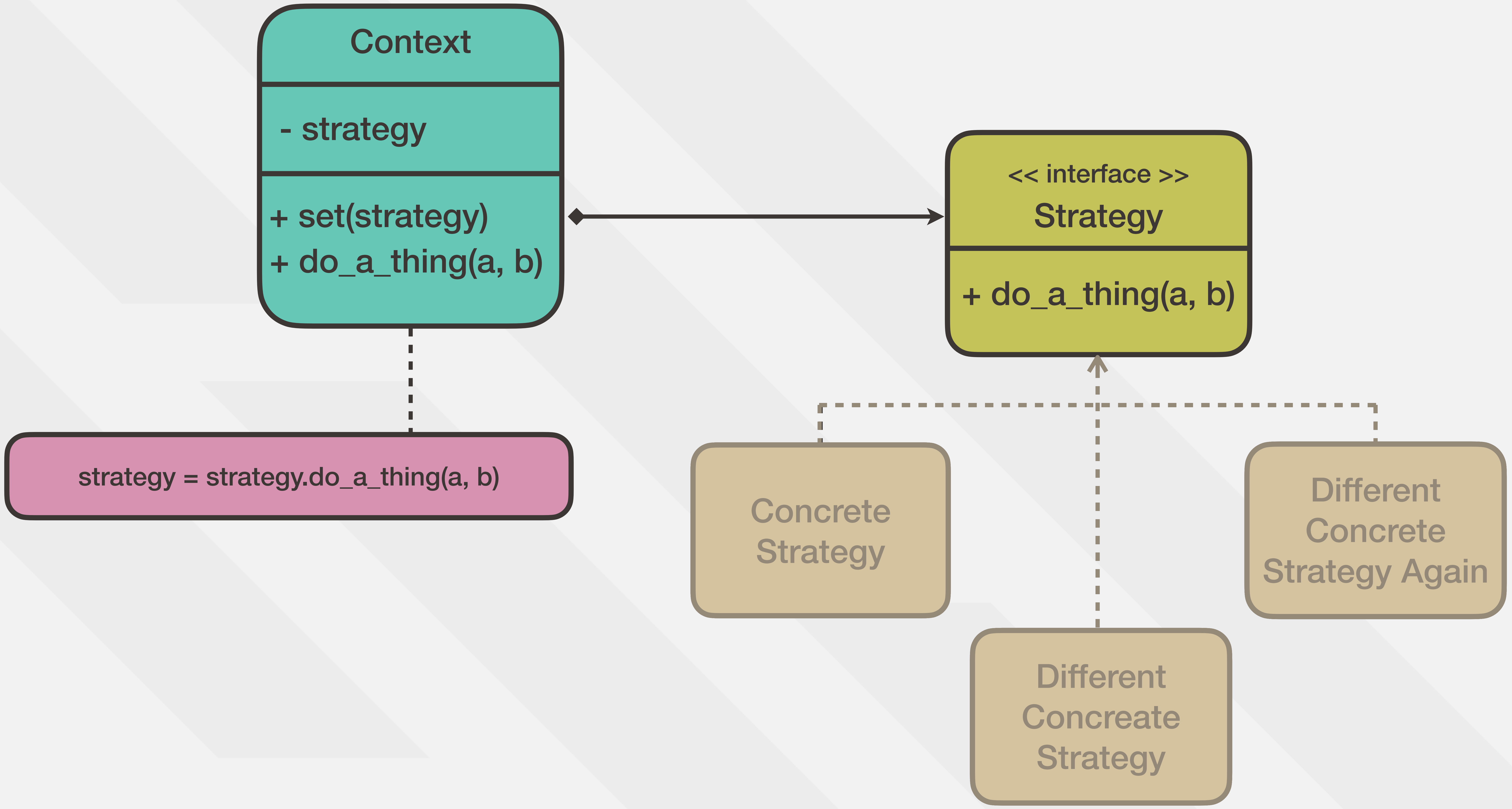
Adapted from
https://refactoring.guru/design-patterns/strategy

STRATEGY PATTERN



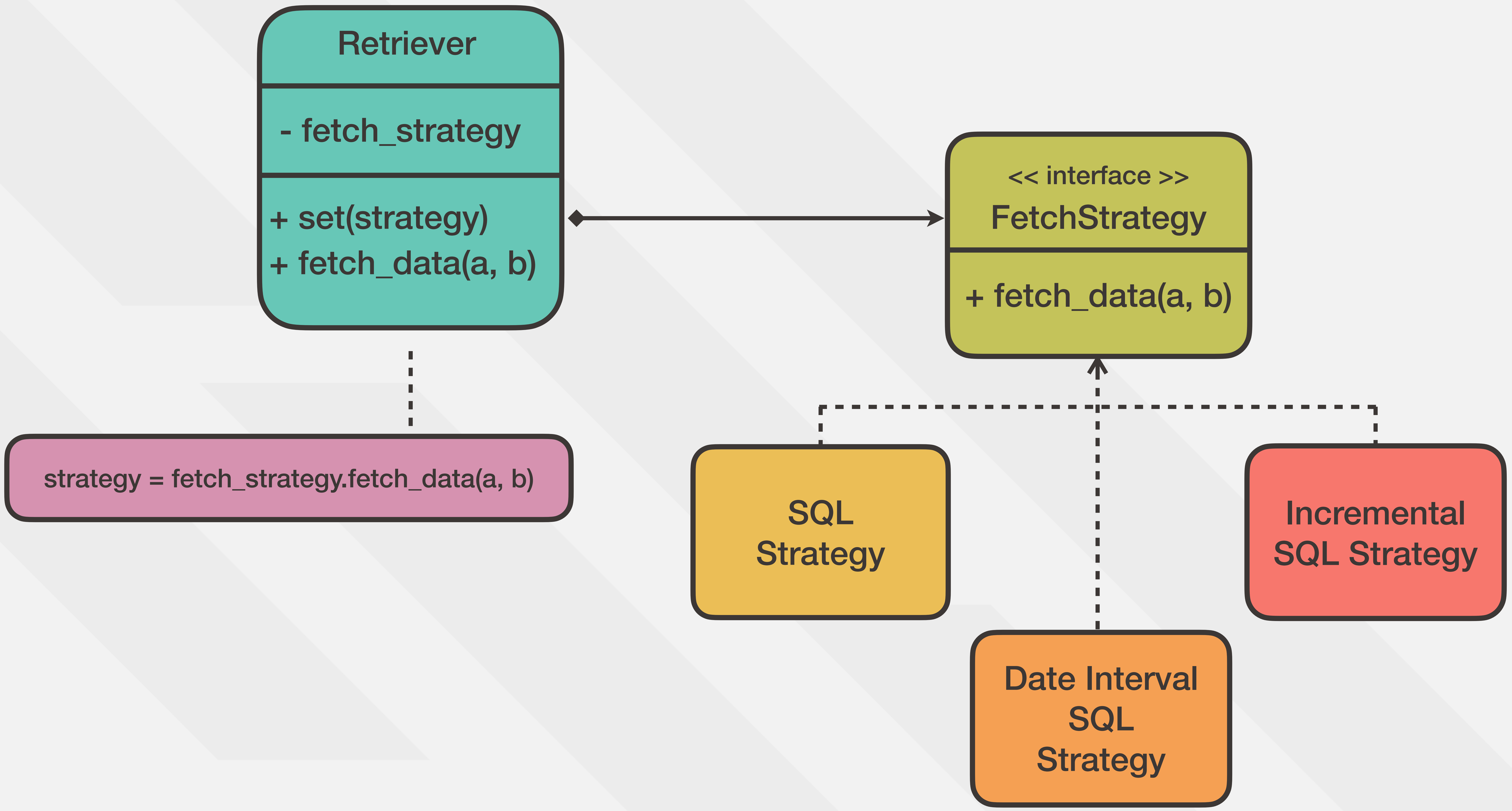
Adapted from
https://refactoring.guru/design-patterns/strategy

STRATEGY PATTERN



Adapted from
https://refactoring.guru/design-patterns/strategy

STRATEGY PATTERN



STRATEGY PATTERN

Adapted from
https://refactoring.guru/design-patterns/strategy

Interface

Context

Strategy

Config

Utilize

STRATEGY PATTERN

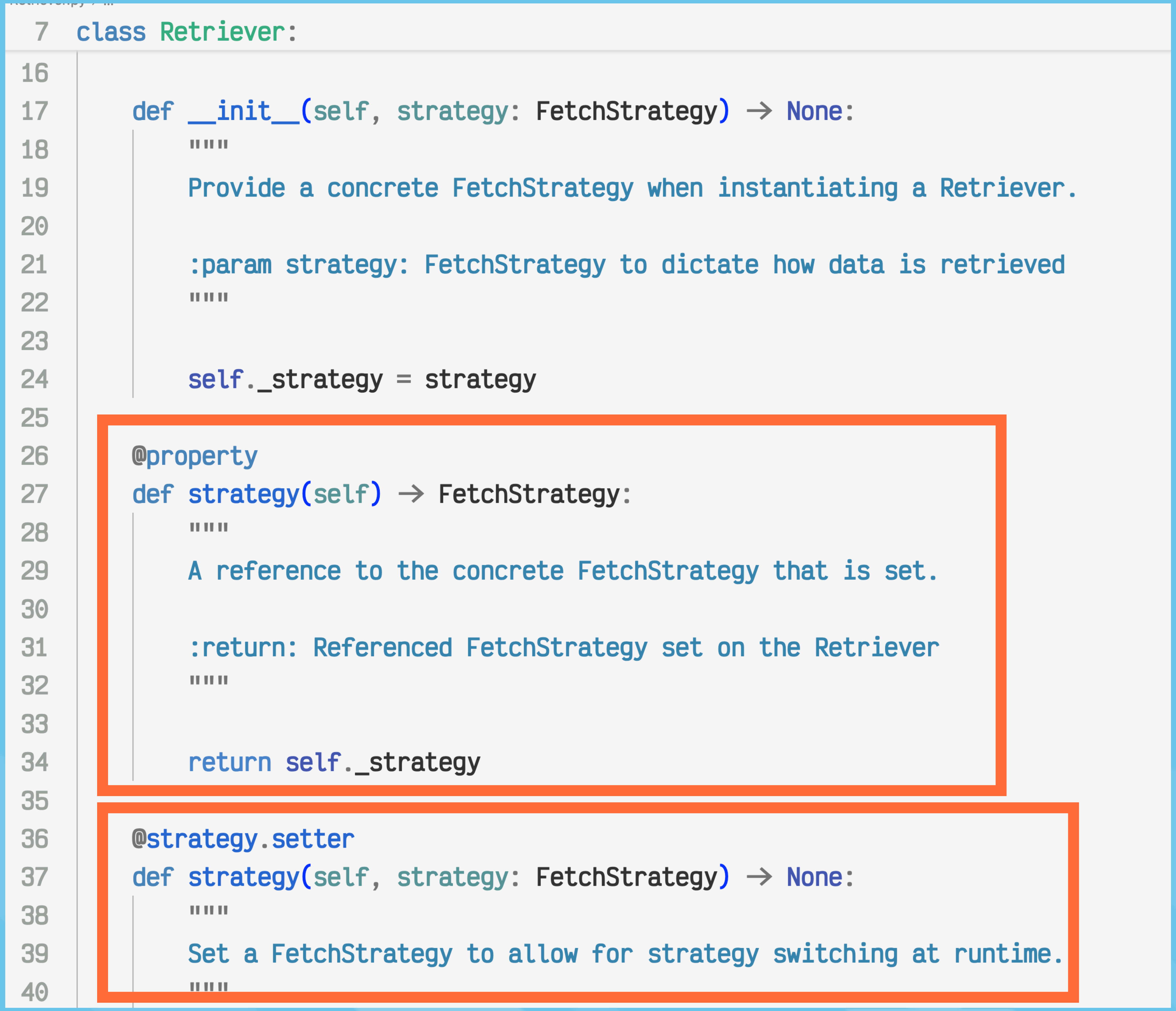
Interface

Context

Strategy

Config

Utilize



```python
class Retriever:

    def __init__(self, strategy: FetchStrategy) → None:
        """
        Provide a concrete FetchStrategy when instantiating a Retriever.

        :param strategy: FetchStrategy to dictate how data is retrieved
        """

        self._strategy = strategy

    @property
    def strategy(self) → FetchStrategy:
        """
        A reference to the concrete FetchStrategy that is set.

        :return: Referenced FetchStrategy set on the Retriever
        """

        return self._strategy

    @strategy.setter
    def strategy(self, strategy: FetchStrategy) → None:
        """
        Set a FetchStrategy to allow for strategy switching at runtime.
        """
```

STRATEGY PATTERN

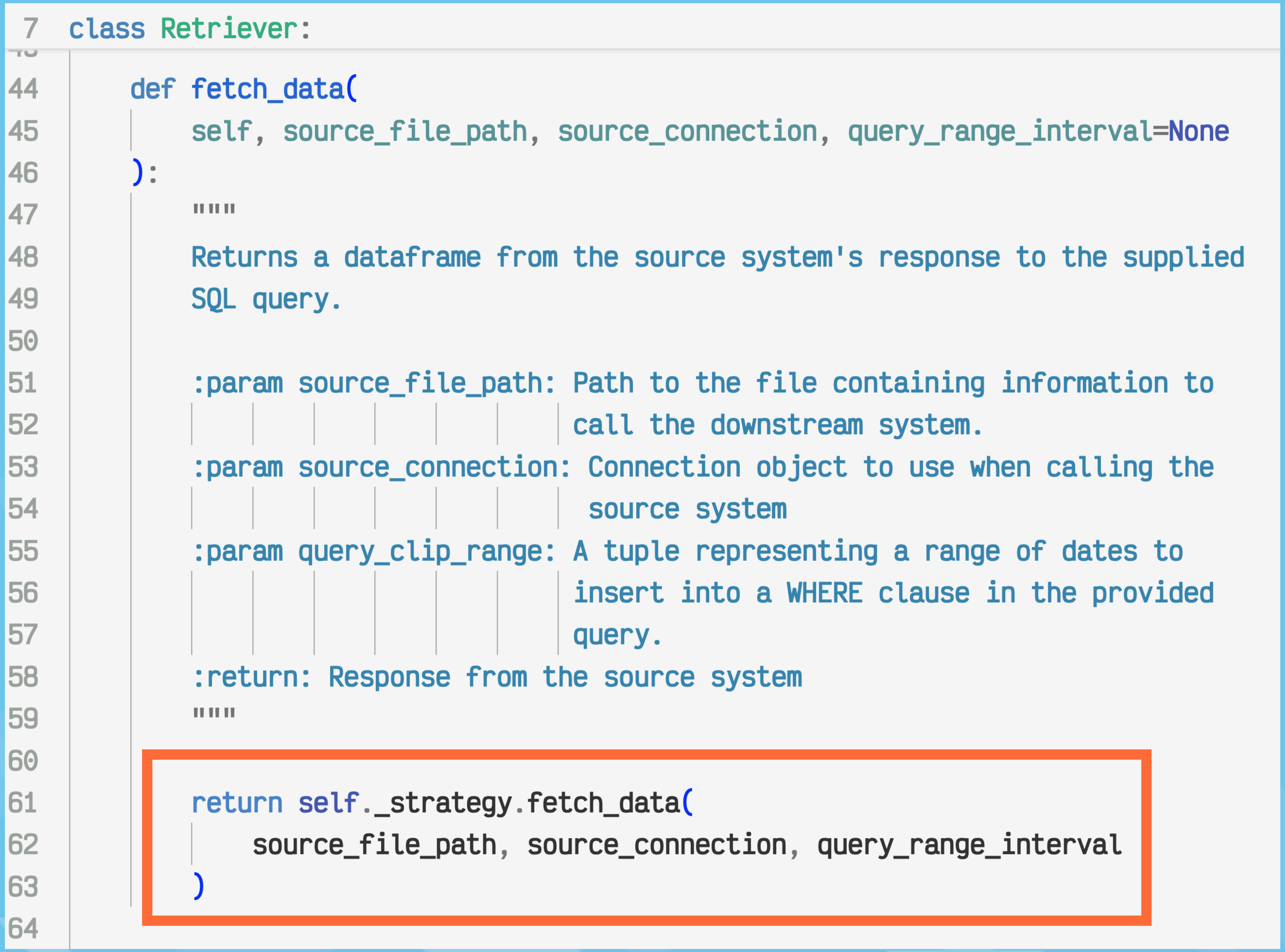
Interface

Context

Strategy

Config

Utilize



```
class Retriever:

    def fetch_data(
        self, source_file_path, source_connection, query_range_interval=None
    ):
        """
        Returns a dataframe from the source system's response to the supplied
        SQL query.

        :param source_file_path: Path to the file containing information to
                                 call the downstream system.
        :param source_connection: Connection object to use when calling the
                                  source system
        :param query_clip_range: A tuple representing a range of dates to
                                 insert into a WHERE clause in the provided
                                 query.
        :return: Response from the source system
        """

        return self._strategy.fetch_data(
            source_file_path, source_connection, query_range_interval
        )
```

STRATEGY PATTERN

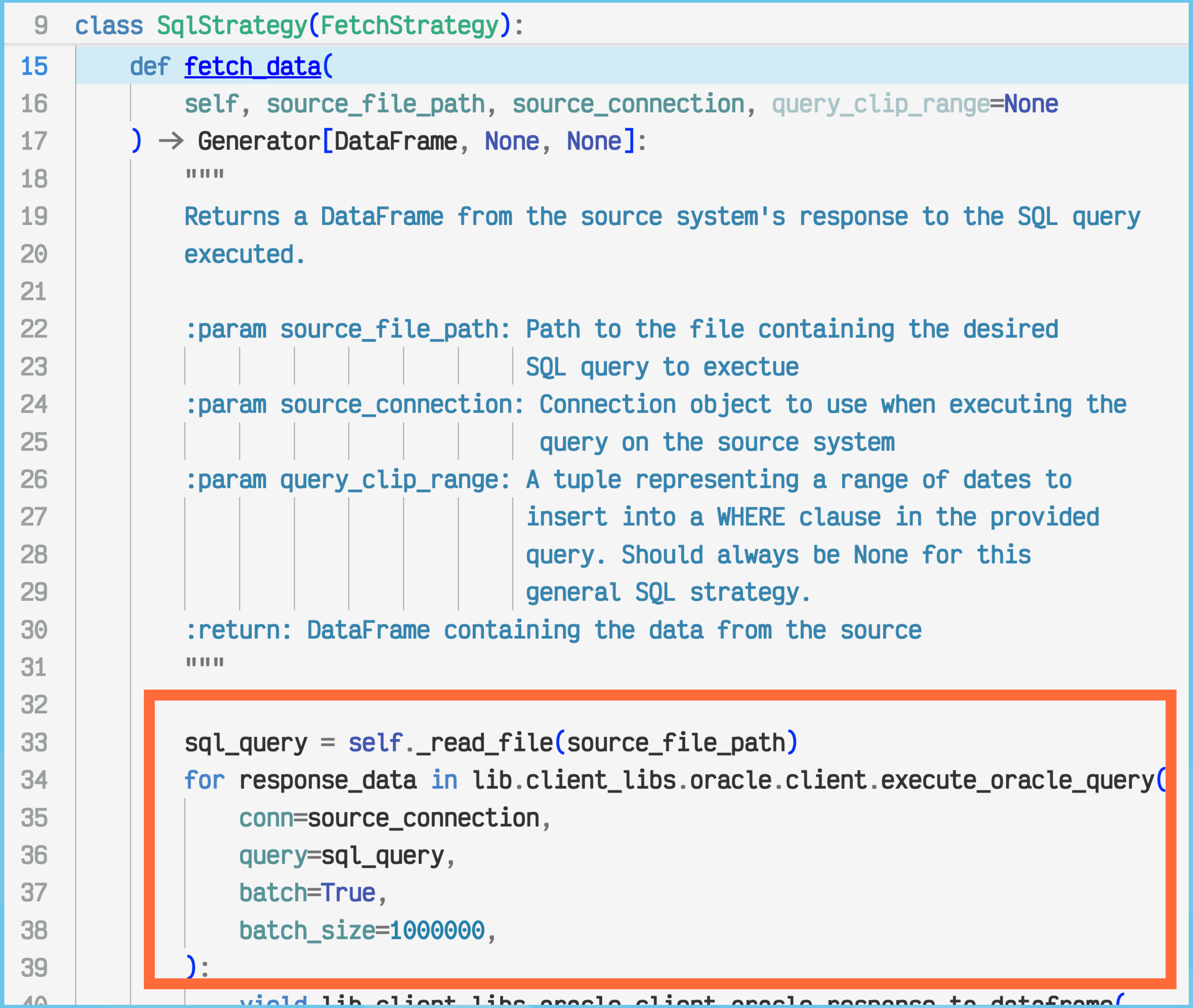
Interface

Context

Strategy

Config

Utilize



```python
class SqlStrategy(FetchStrategy):
    def fetch_data(
        self, source_file_path, source_connection, query_clip_range=None
    ) -> Generator[DataFrame, None, None]:
        """
        Returns a DataFrame from the source system's response to the SQL query
        executed.

        :param source_file_path: Path to the file containing the desired
                                 SQL query to exectue
        :param source_connection: Connection object to use when executing the
                                  query on the source system
        :param query_clip_range: A tuple representing a range of dates to
                                 insert into a WHERE clause in the provided
                                 query. Should always be None for this
                                 general SQL strategy.
        :return: DataFrame containing the data from the source
        """

        sql_query = self._read_file(source_file_path)
        for response_data in lib.client_libs.oracle.client.execute_oracle_query(
            conn=source_connection,
            query=sql_query,
            batch=True,
            batch_size=1000000,
        ):
            yield lib.client_libs.oracle.client.oracle_response_to_dataframe(
```

STRATEGY PATTERN

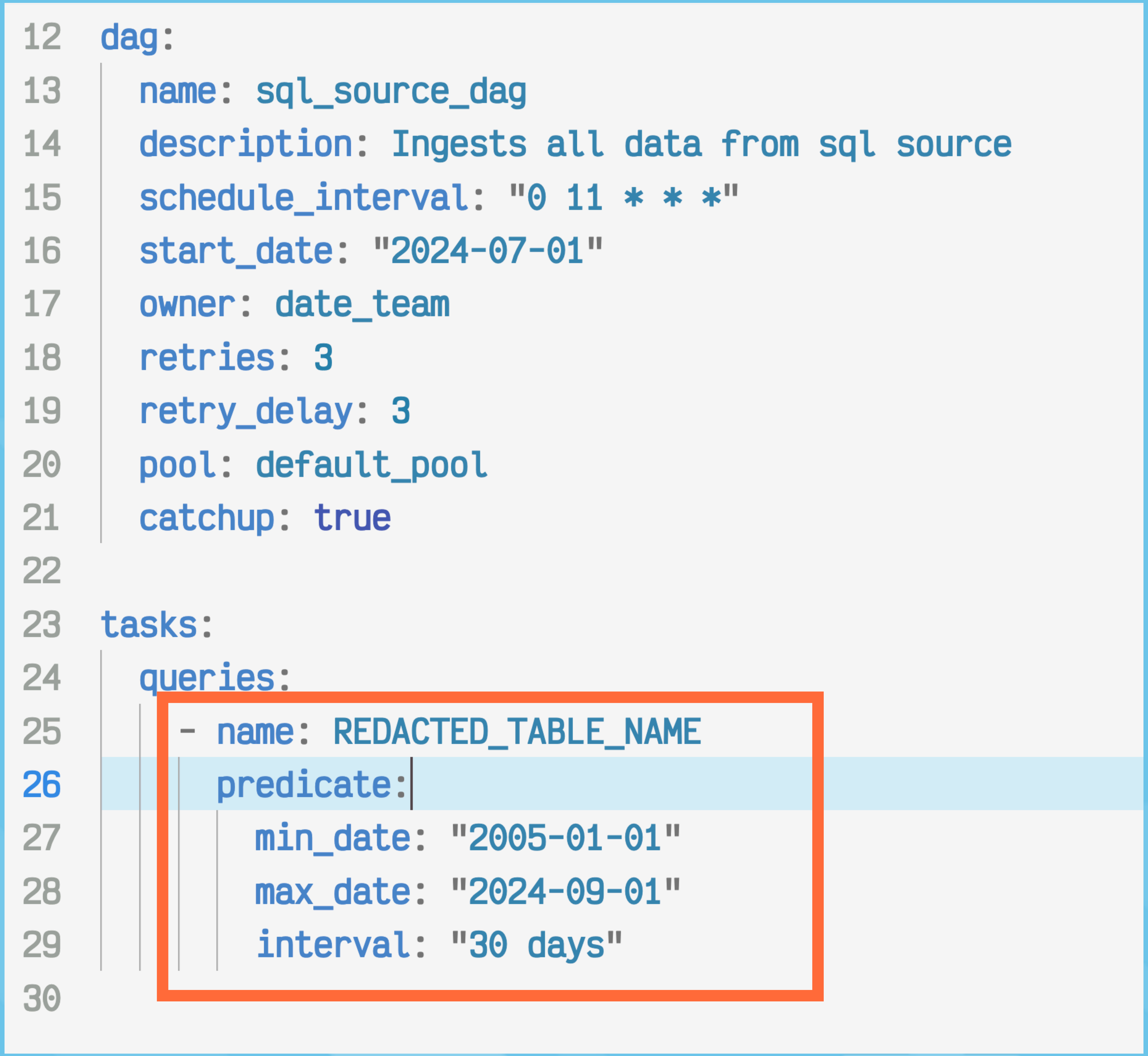
Interface

Context

Strategy

Config

Utilize



```yaml
dag:
  name: sql_source_dag
  description: Ingests all data from sql source
  schedule_interval: "0 11 * * *"
  start_date: "2024-07-01"
  owner: date_team
  retries: 3
  retry_delay: 3
  pool: default_pool
  catchup: true

tasks:
  queries:
    - name: REDACTED_TABLE_NAME
      predicate:
        min_date: "2005-01-01"
        max_date: "2024-09-01"
        interval: "30 days"

```

STRATEGY PATTERN

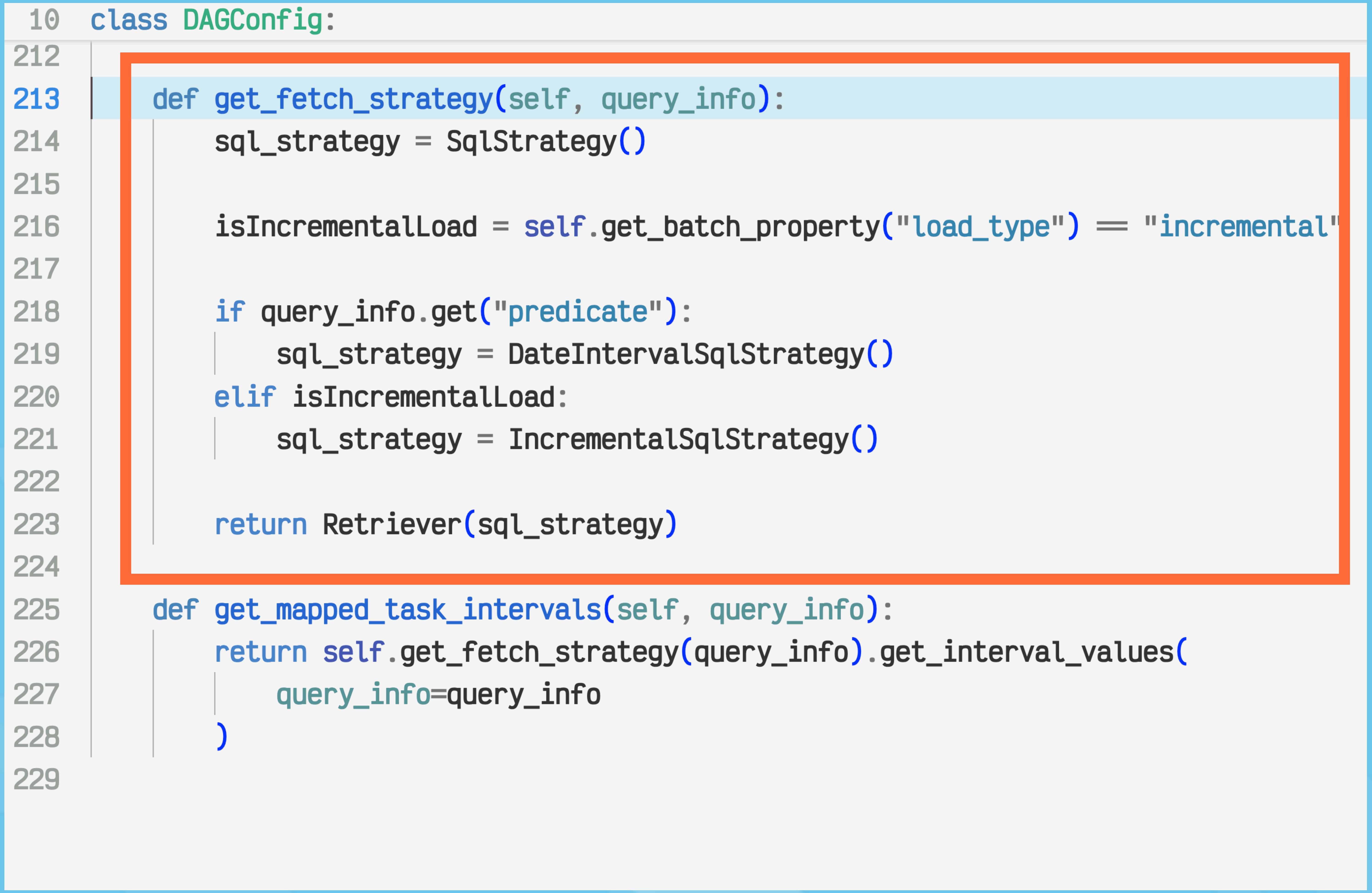
Interface

Context

Strategy

Config

Utilize



```
class DAGConfig:

    def get_fetch_strategy(self, query_info):
        sql_strategy = SqlStrategy()

        isIncrementalLoad = self.get_batch_property("load_type") == "incremental"

        if query_info.get("predicate"):
            sql_strategy = DateIntervalSqlStrategy()
        elif isIncrementalLoad:
            sql_strategy = IncrementalSqlStrategy()

        return Retriever(sql_strategy)

    def get_mapped_task_intervals(self, query_info):
        return self.get_fetch_strategy(query_info).get_interval_values(
            query_info=query_info
        )
```

STRATEGY PATTERN

80 MPH

ORCHESTRATE INGESTION

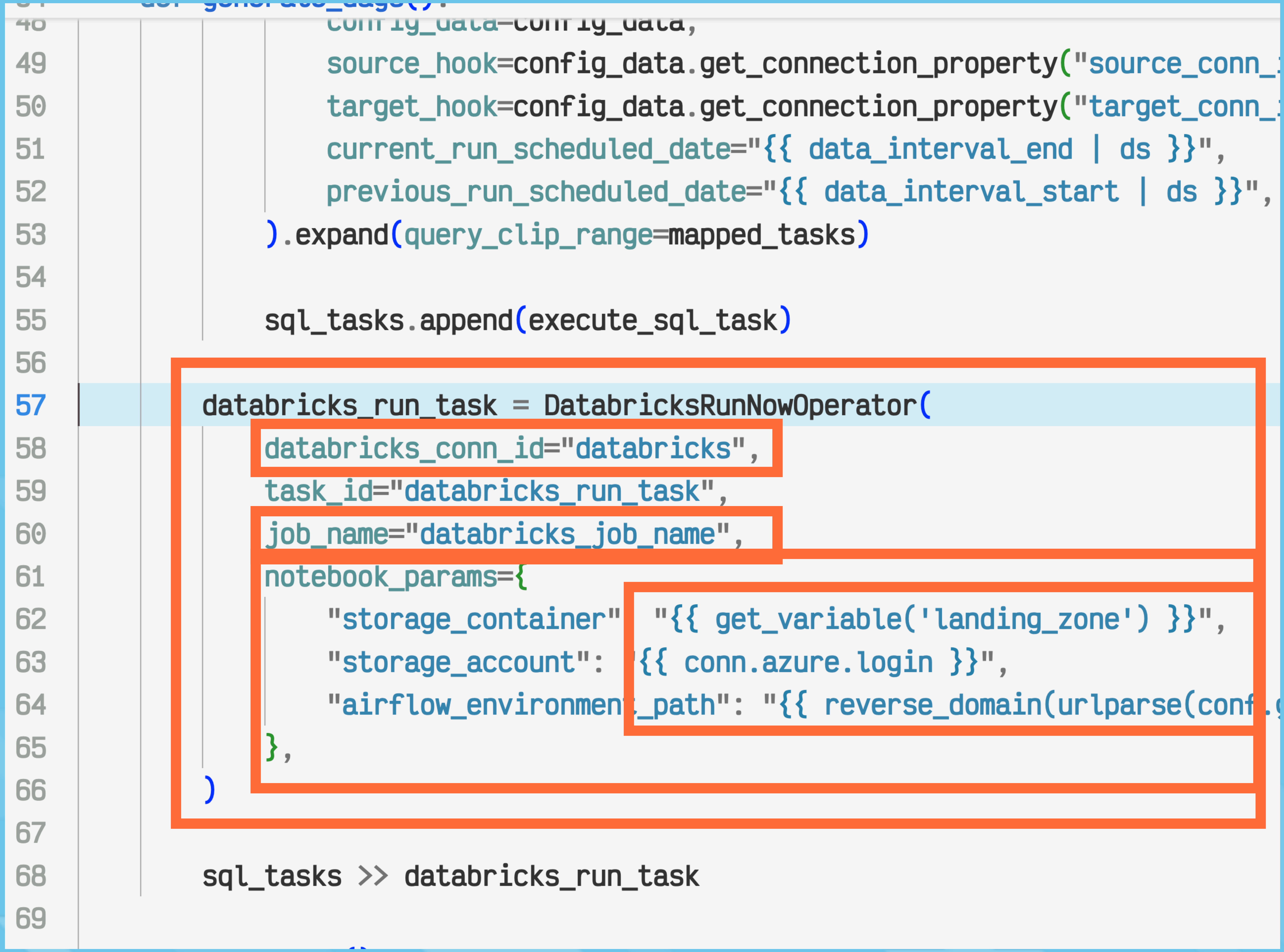
DAG

Task

Operator

Hook

Other Code



```
    def generate_dags():
                config_data=config_data,
                source_hook=config_data.get_connection_property("source_conn_
                target_hook=config_data.get_connection_property("target_conn_
                current_run_scheduled_date="{{ data_interval_end | ds }}",
                previous_run_scheduled_date="{{ data_interval_start | ds }}",
            ).expand(query_clip_range=mapped_tasks)

            sql_tasks.append(execute_sql_task)

        databricks_run_task = DatabricksRunNowOperator(
            databricks_conn_id="databricks",
            task_id="databricks_run_task",
            job_name="databricks_job_name",
            notebook_params={
                "storage_container": "{{ get_variable('landing_zone') }}",
                "storage_account": "{{ conn.azure.login }}",
                "airflow_environment_path": "{{ reverse_domain(urlparse(conf.g
            },
        )

        sql_tasks >> databricks_run_task
```

ORCHESTRATE INGESTION

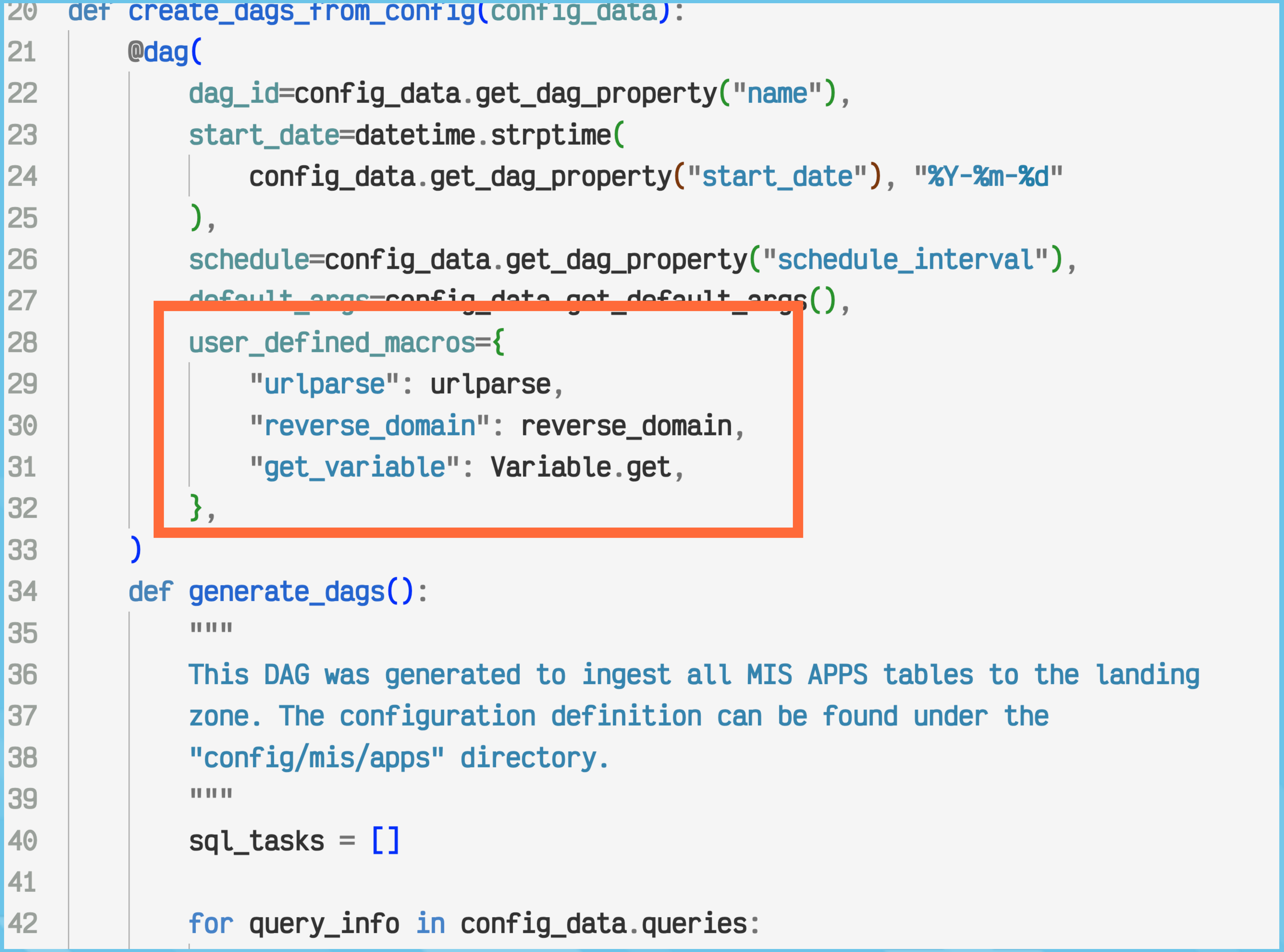
DAG

Task

Operator

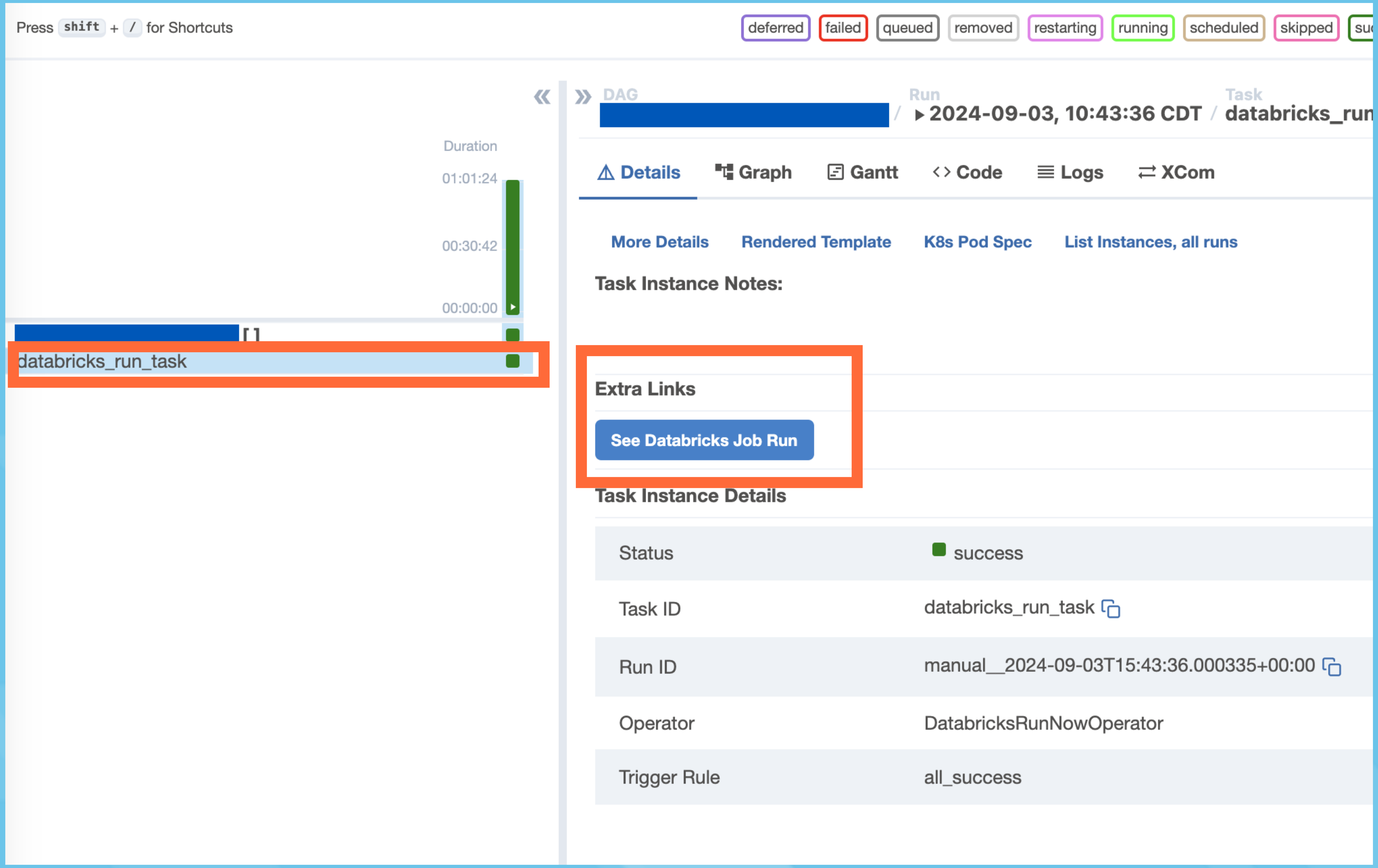
Hook

Other Code



```python
def create_dags_from_config(config_data):
    @dag(
        dag_id=config_data.get_dag_property("name"),
        start_date=datetime.strptime(
            config_data.get_dag_property("start_date"), "%Y-%m-%d"
        ),
        schedule=config_data.get_dag_property("schedule_interval"),
        default_args=config_data.get_default_args(),
        user_defined_macros={
            "urlparse": urlparse,
            "reverse_domain": reverse_domain,
            "get_variable": Variable.get,
        },
    )
    def generate_dags():
        """
        This DAG was generated to ingest all MIS APPS tables to the landing
        zone. The configuration definition can be found under the
        "config/mis/apps" directory.
        """
        sql_tasks = []

        for query_info in config_data.queries:
```

ORCHESTRATE INGESTION



Press shift + / for Shortcuts

deferred failed queued removed restarting running scheduled skipped su

DAG [redacted] / Run ▸ 2024-09-03, 10:43:36 CDT / Task databricks_run

Duration
01:01:24
00:30:42
00:00:00

[redacted] [ ]
databricks_run_task

Details | Graph | Gantt | Code | Logs | XCom

More Details | Rendered Template | K8s Pod Spec | List Instances, all runs

Task Instance Notes:

Extra Links

See Databricks Job Run

Task Instance Details

Status	🟩 success
Task ID	databricks_run_task
Run ID	manual__2024-09-03T15:43:36.000335+00:00
Operator	DatabricksRunNowOperator
Trigger Rule	all_success

ORCHESTRATE INGESTION

100 MPH

Ludicrous Speed GO!
MGM

ORCHESTRATE EVERYTHING

- Config-driven API DAGs
- ExternalTaskSensors for silver jobs
- Improved DLT job integration

FINISH LINE

DATA AT BURNS & MCDONNELL

SCALEABLE

RELIABLE

EVOLVABLE

THANK YOU!

Slide Deck and Resources