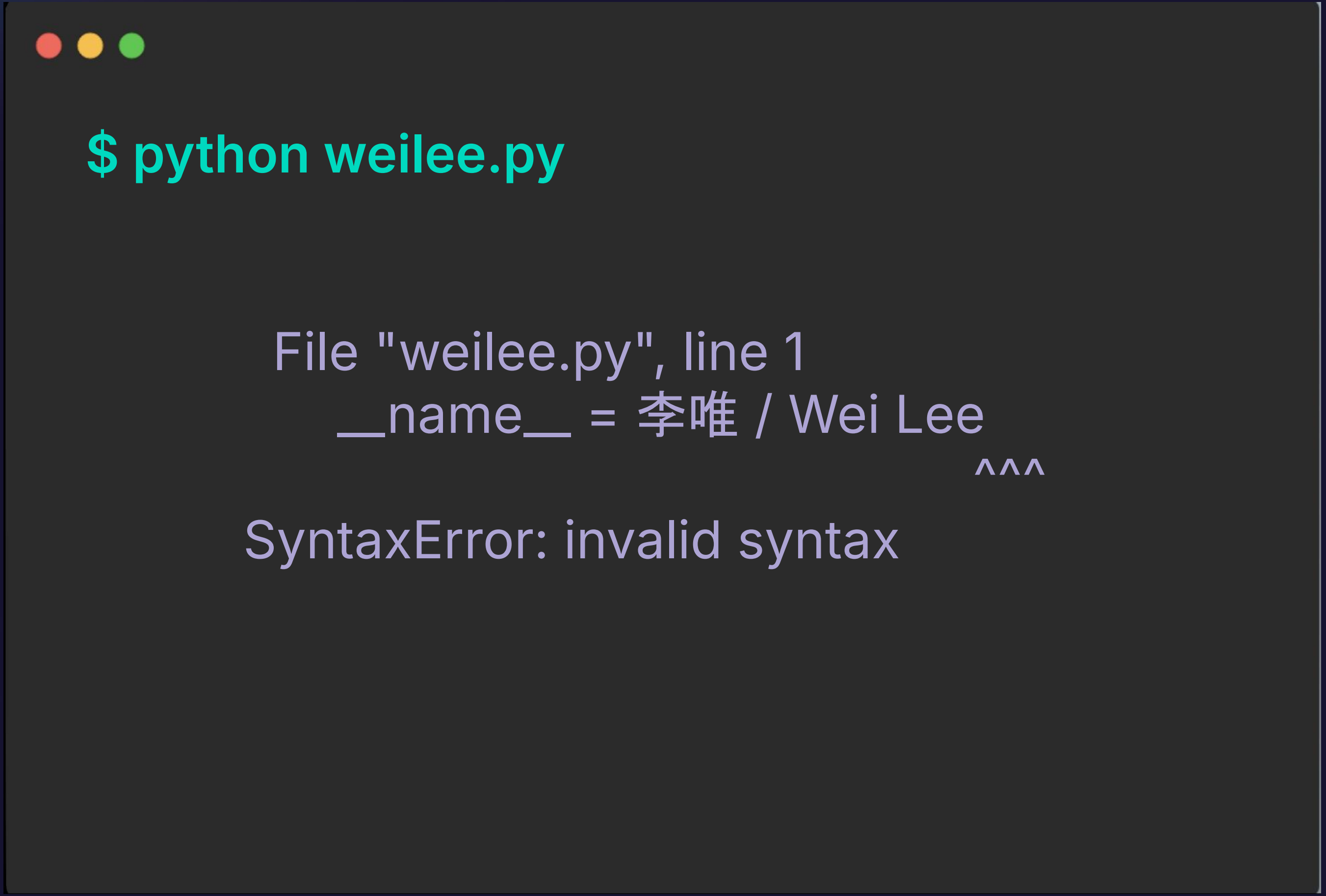
What If…?
Running Airflow Tasks without the workers

ASTRONOMER

```
$ cat weilee.py

__name__ = 李唯 / Wei Lee
__what_i_am_doing__ = [
    Software Engineer @ Astronomer,
    Committer @ Apache Airflow,
    First Time Speaker @ Airflow Summit
]
__github__ = Lee-W
__linkedin__ = clleew
__site__ = https://wei-lee.me
```

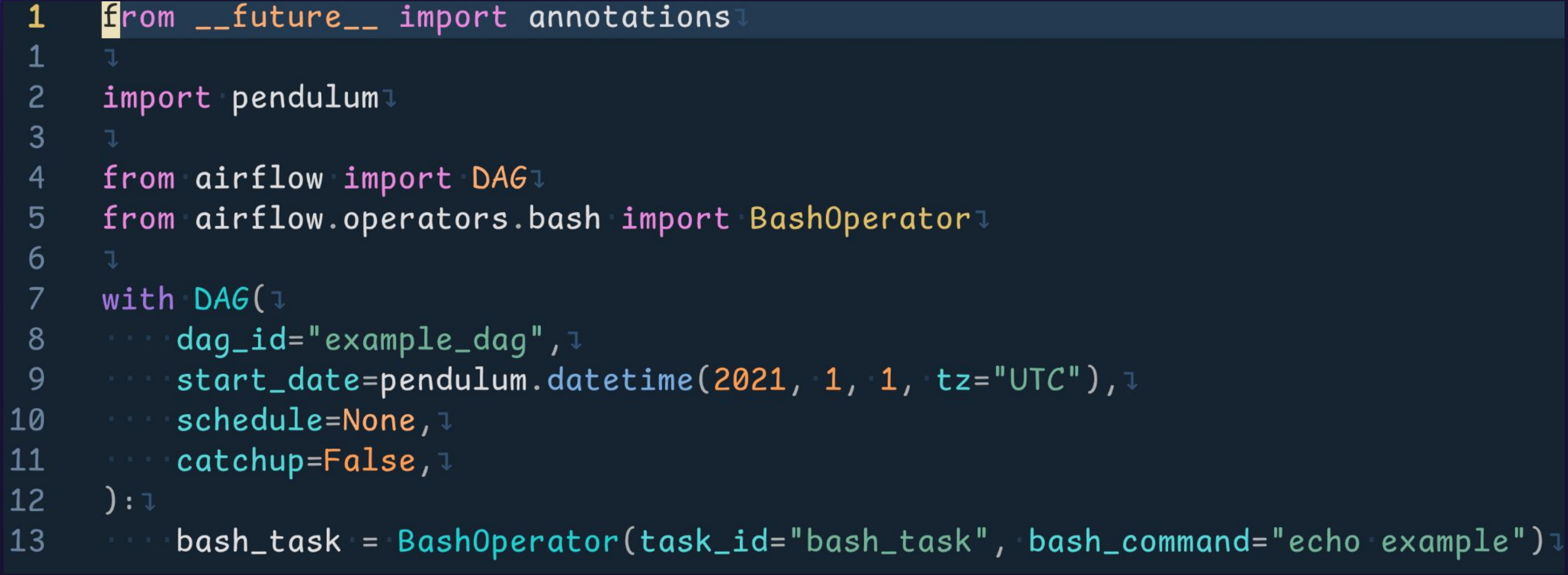


```
$ python weilee.py


    File "weilee.py", line 1
      __name__ = 李唯 / Wei Lee
                             ^^^
   SyntaxError: invalid syntax
```

QR Code links to this slide deck

Let's start with how a typical task works now

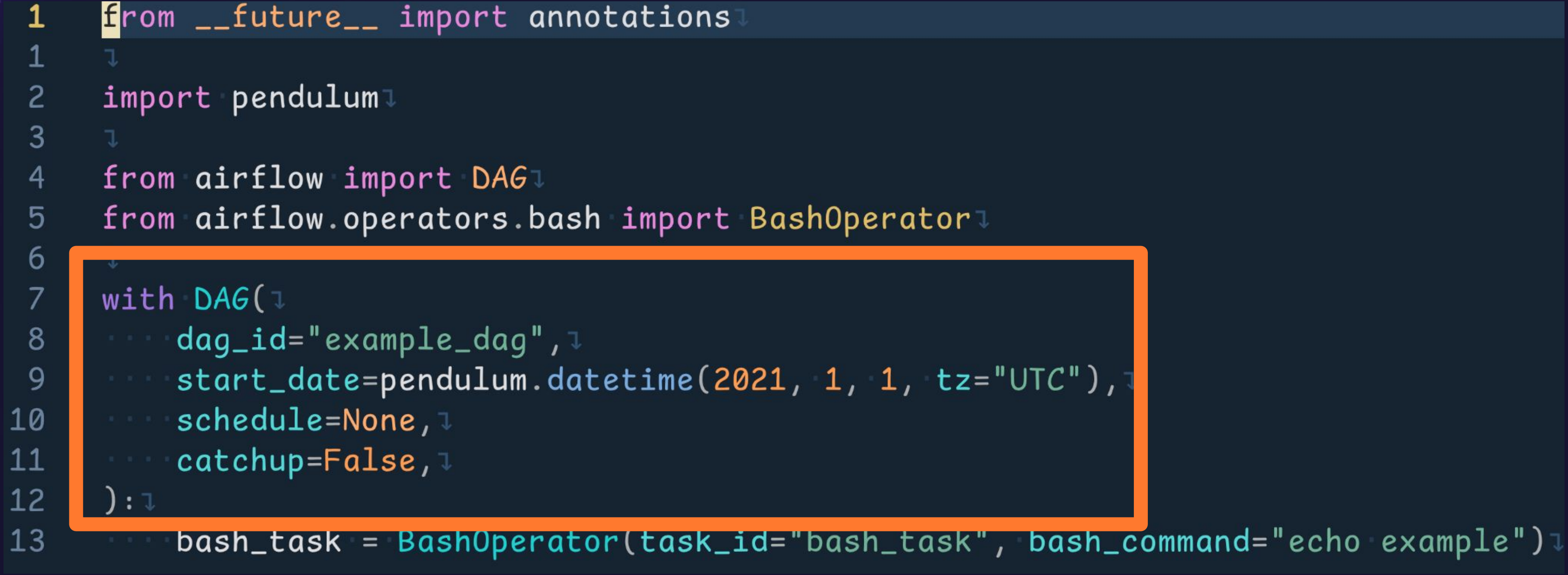
Define a DAG



```python
from __future__ import annotations

import pendulum

from airflow import DAG
from airflow.operators.bash import BashOperator

with DAG(
    dag_id="example_dag",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    schedule=None,
    catchup=False,
):
    bash_task = BashOperator(task_id="bash_task", bash_command="echo example")
```

Let's start with how a typical task works now

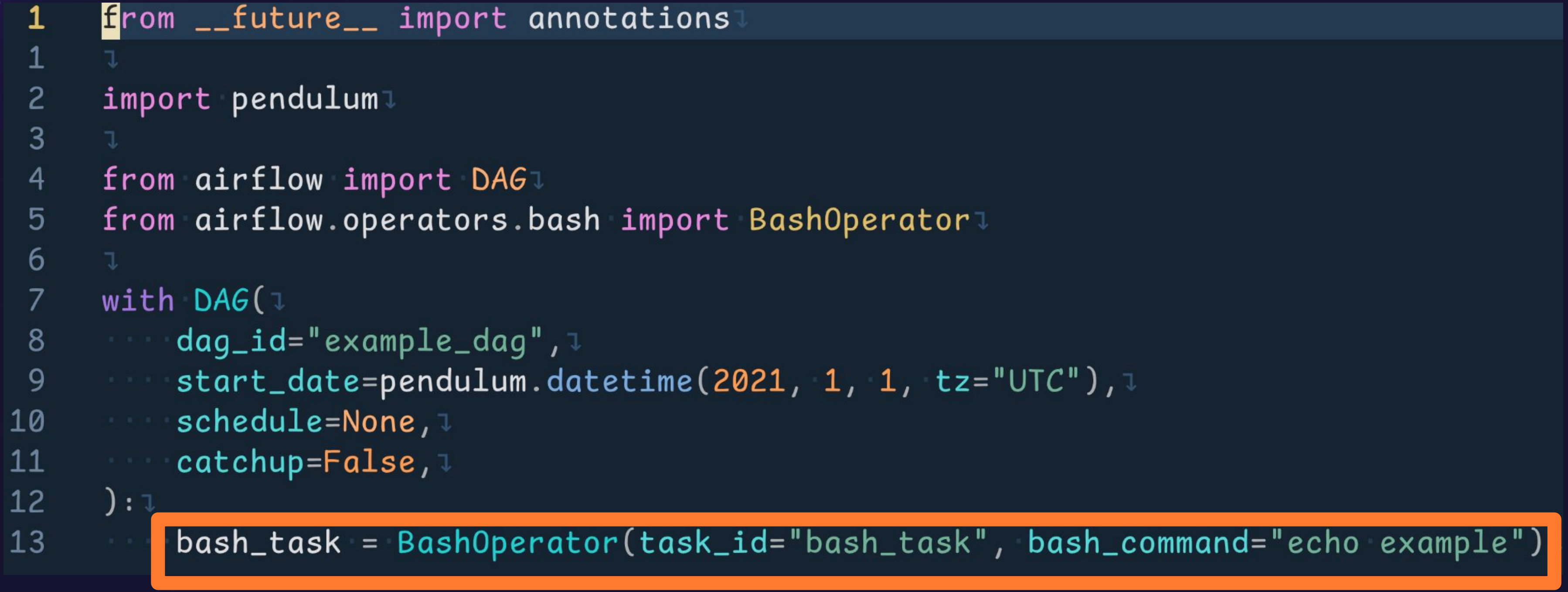
Define a DAG



```python
from __future__ import annotations

import pendulum

from airflow import DAG
from airflow.operators.bash import BashOperator

with DAG(
    dag_id="example_dag",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    schedule=None,
    catchup=False,
):
    bash_task = BashOperator(task_id="bash_task", bash_command="echo example")
```

Let's start with how a typical task works now

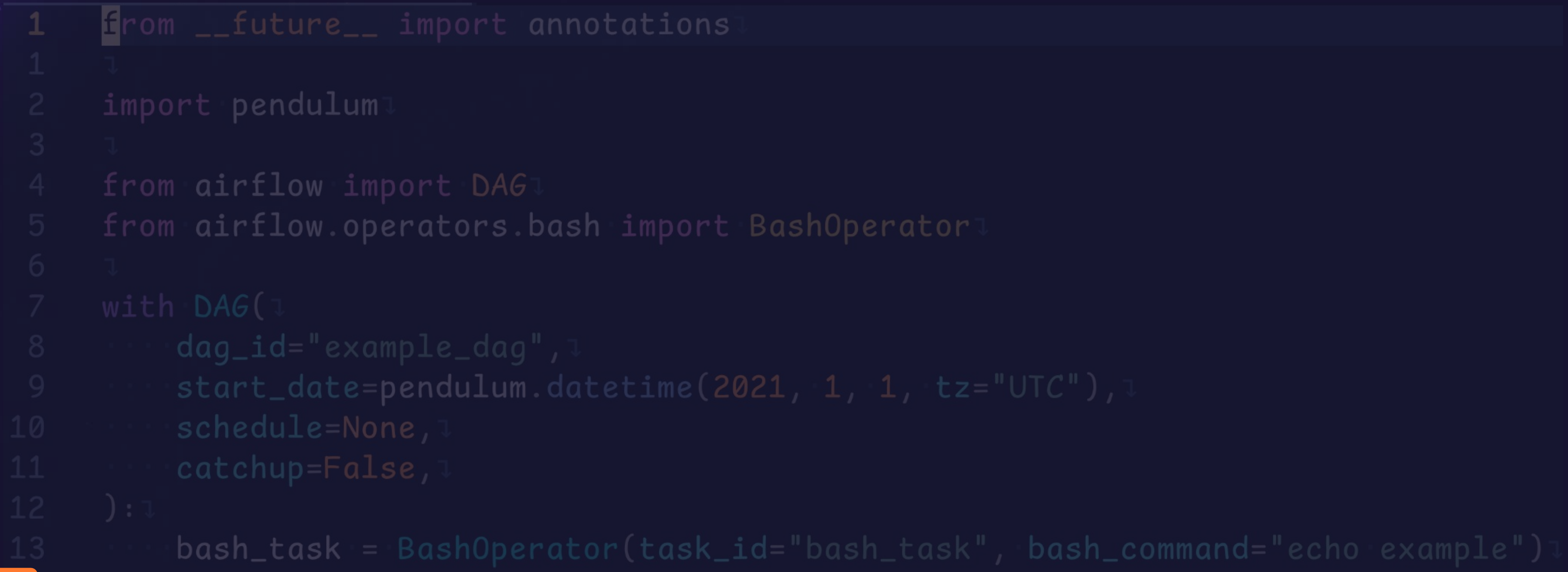
Define a DAG



```
from __future__ import annotations

import pendulum

from airflow import DAG
from airflow.operators.bash import BashOperator

with DAG(
    dag_id="example_dag",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    schedule=None,
    catchup=False,
):
    bash_task = BashOperator(task_id="bash_task", bash_command="echo example")
```

Let's start with how a typical task works now

Define a DAG



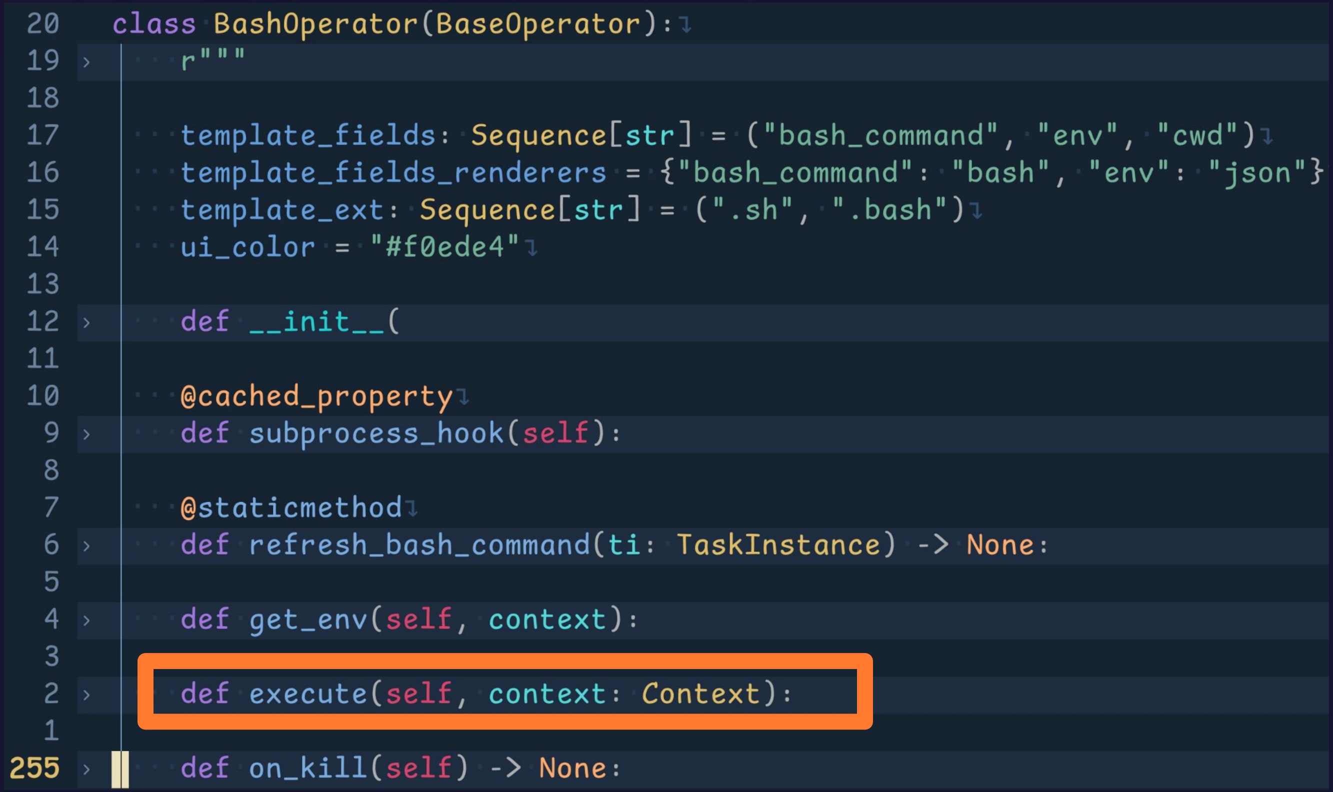
```python
from __future__ import annotations

import pendulum

from airflow import DAG
from airflow.operators.bash import BashOperator

with DAG(
    dag_id="example_dag",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    schedule=None,
    catchup=False,
):
    bash_task = BashOperator(task_id="bash_task", bash_command="echo example")
```

This page is for traditional operators.

ASTRONOMER

Let's start with how a typical task works now

Under the hood, it runs “execute”.

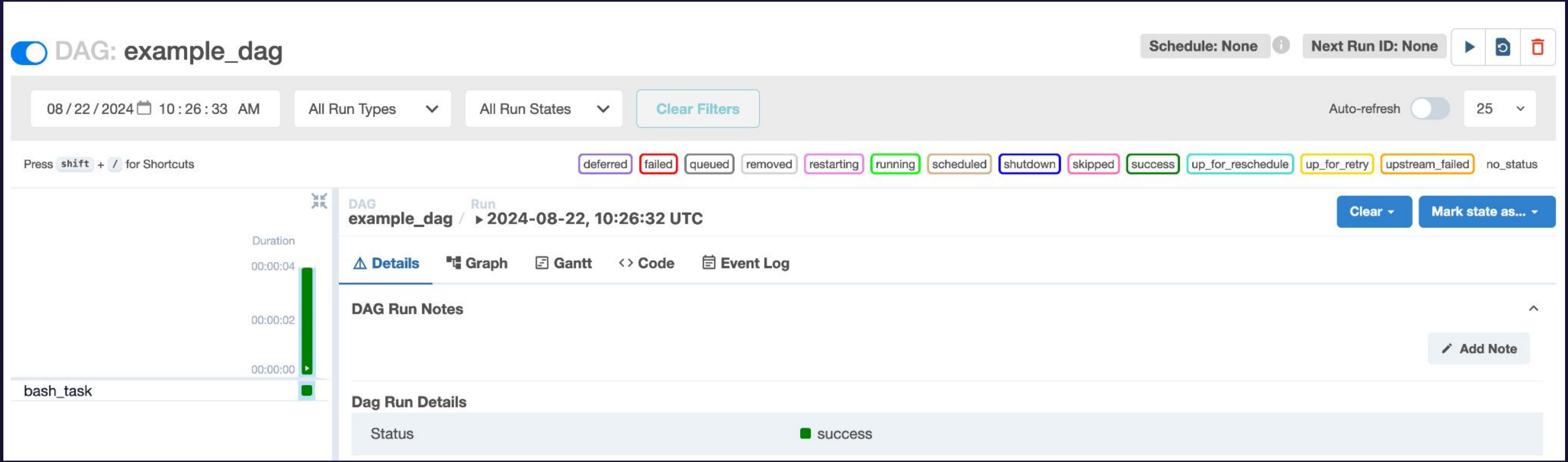


```
class BashOperator(BaseOperator):
    r"""

    template_fields: Sequence[str] = ("bash_command", "env", "cwd")
    template_fields_renderers = {"bash_command": "bash", "env": "json"}
    template_ext: Sequence[str] = (".sh", ".bash")
    ui_color = "#f0ede4"

    def __init__(

    @cached_property
    def subprocess_hook(self):

    @staticmethod
    def refresh_bash_command(ti: TaskInstance) -> None:

    def get_env(self, context):

    def execute(self, context: Context):

    def on_kill(self) -> None:
```

ASTRONOMER

Let's start with how a typical task works now

🎉



DAG: example_dag

Schedule: None | Next Run ID: None

08 / 22 / 2024 10 : 26 : 33 AM | All Run Types | All Run States | Clear Filters | Auto-refresh | 25

Press shift + / for Shortcuts

deferred failed queued removed restarting running scheduled shutdown skipped success up_for_reschedule up_for_retry upstream_failed no_status

Duration 00:00:04 00:00:02 00:00:00

bash_task

DAG example_dag / Run 2024-08-22, 10:26:32 UTC

Clear | Mark state as...

Details | Graph | Gantt | Code | Event Log

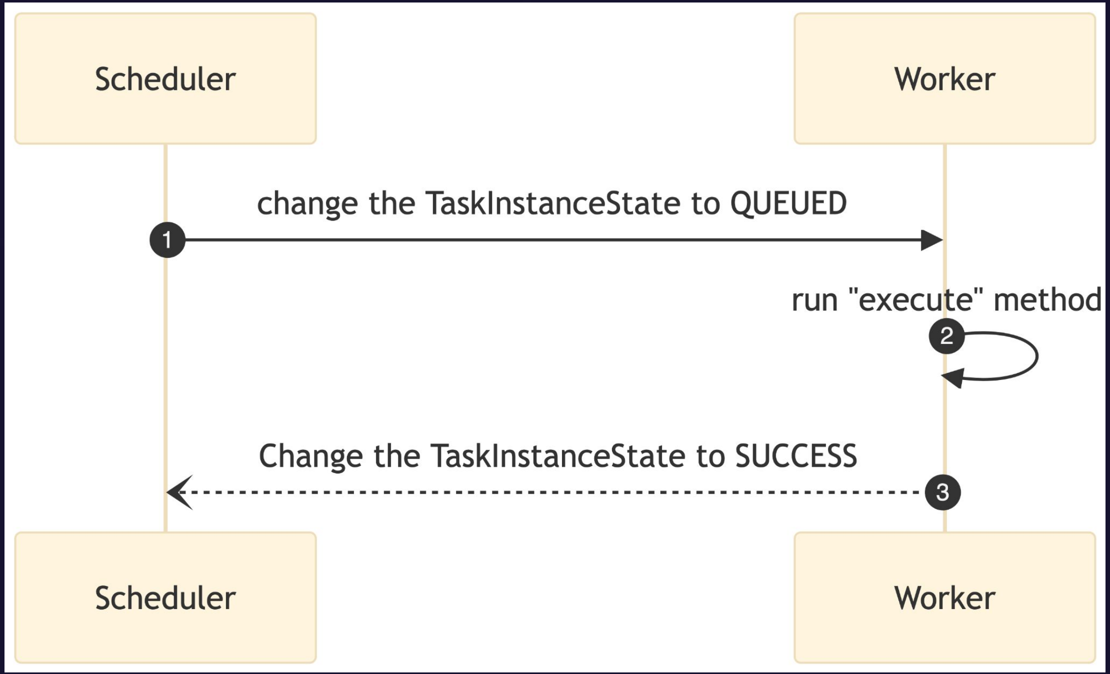
DAG Run Notes

Add Note

Dag Run Details

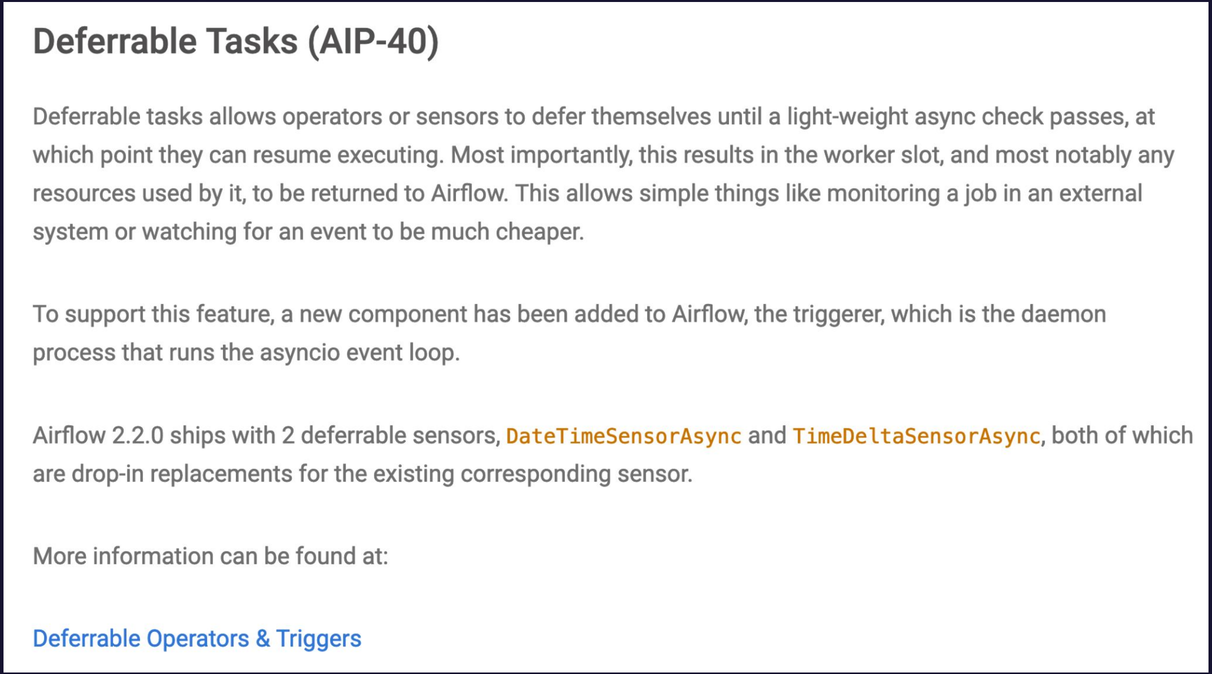
Status	🟩 success

Let's start with how a typical task works now


Scheduler
Worker
change the TaskInstanceState to QUEUED
1
run "execute" method
2
Change the TaskInstanceState to SUCCESS
3
Scheduler
Worker

Since Airflow 2.2

deferrable operator was introduced



Deferrable Tasks (AIP-40)

Deferrable tasks allows operators or sensors to defer themselves until a light-weight async check passes, at which point they can resume executing. Most importantly, this results in the worker slot, and most notably any resources used by it, to be returned to Airflow. This allows simple things like monitoring a job in an external system or watching for an event to be much cheaper.

To support this feature, a new component has been added to Airflow, the triggerer, which is the daemon process that runs the asyncio event loop.

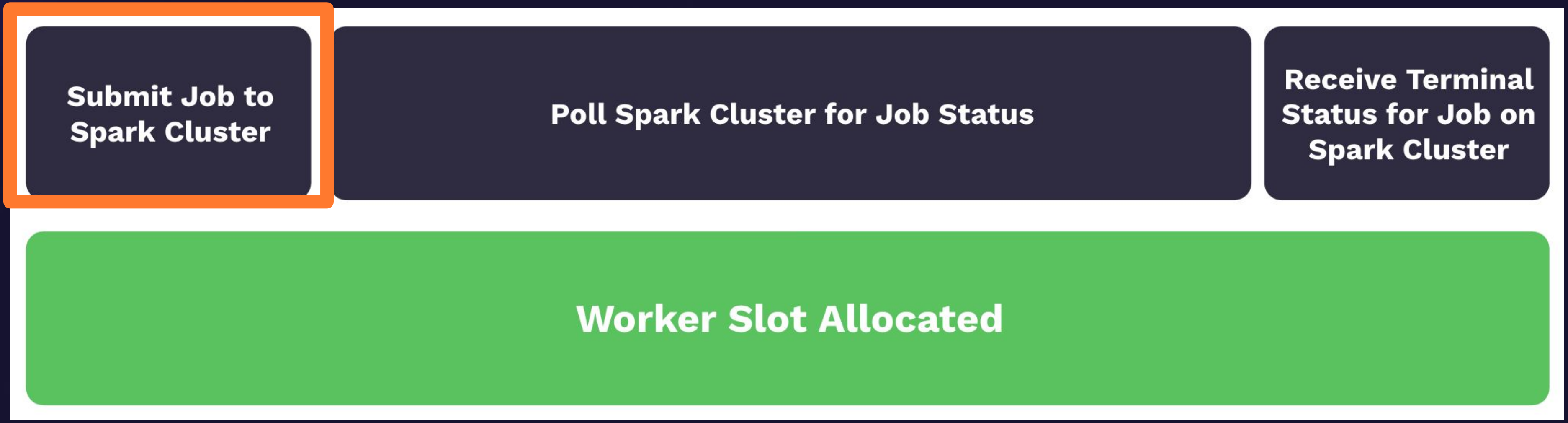
Airflow 2.2.0 ships with 2 deferrable sensors, `DateTimeSensorAsync` and `TimeDeltaSensorAsync`, both of which are drop-in replacements for the existing corresponding sensor.

More information can be found at:

Deferrable Operators & Triggers

But why?

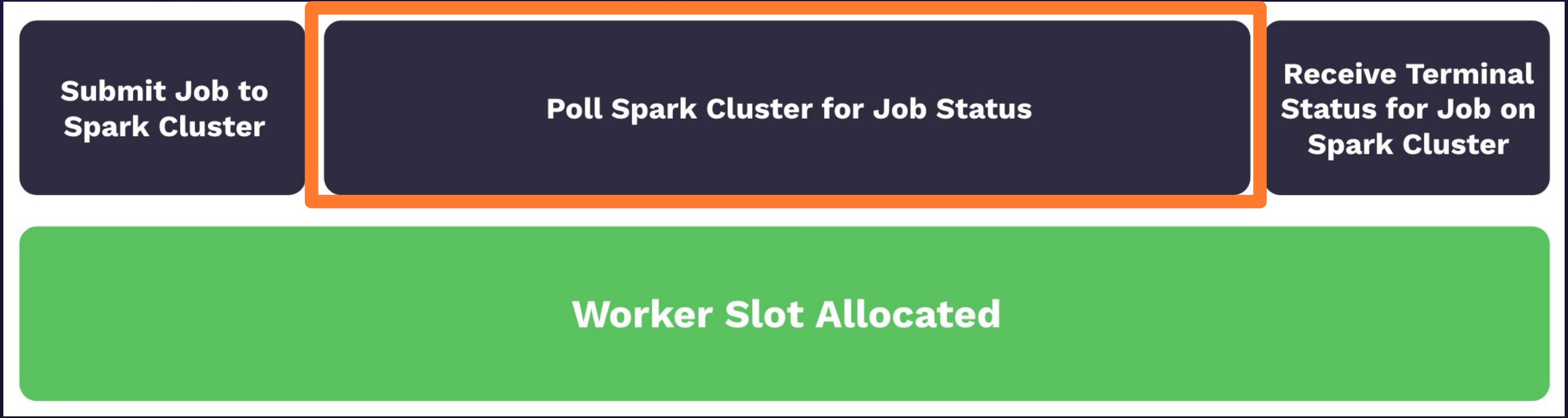
Non-Deferrable Operator


Submit Job to Spark Cluster
Poll Spark Cluster for Job Status
Receive Terminal Status for Job on Spark Cluster
Worker Slot Allocated

from: https://www.astronomer.io/docs/learn/deferrable-operators

But why?

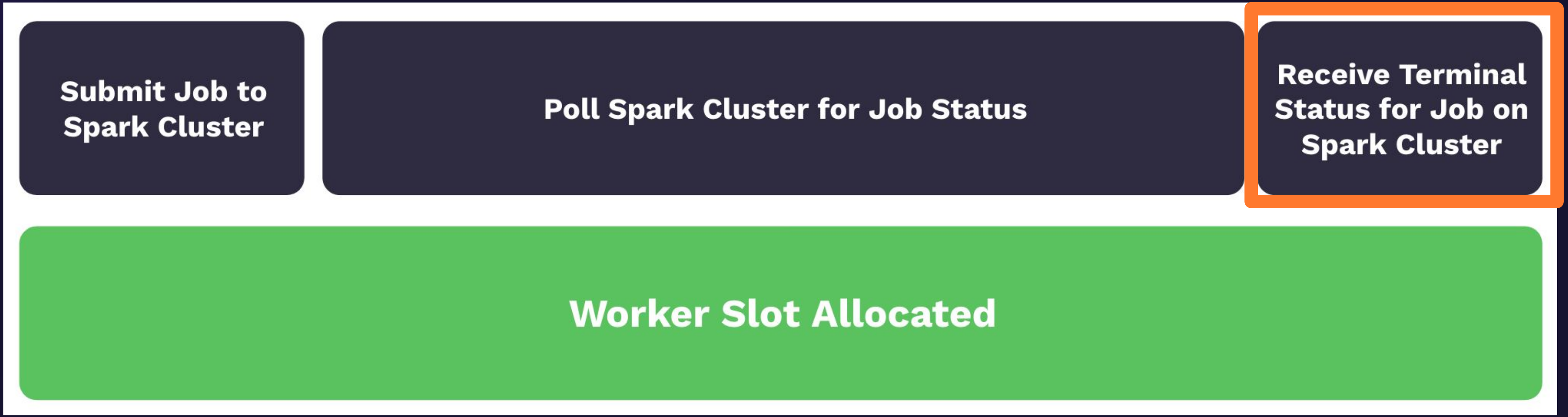
Non-Deferrable Operator


Submit Job to Spark Cluster
Poll Spark Cluster for Job Status
Receive Terminal Status for Job on Spark Cluster
Worker Slot Allocated

from: https://www.astronomer.io/docs/learn/deferrable-operators

But why?

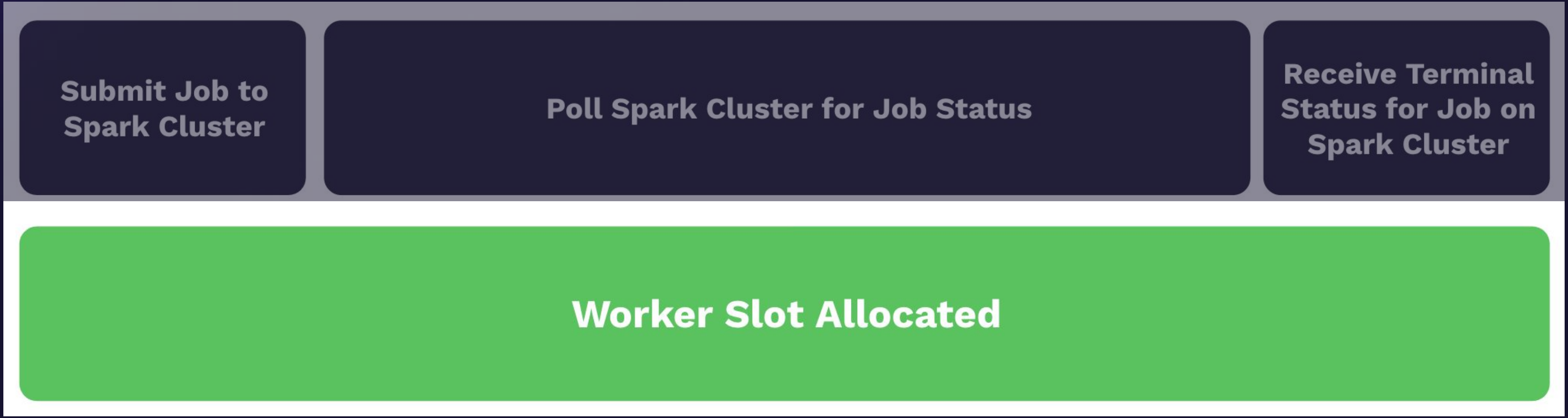
Non-Deferrable Operator


Submit Job to Spark Cluster
Poll Spark Cluster for Job Status
Receive Terminal Status for Job on Spark Cluster
Worker Slot Allocated

from: https://www.astronomer.io/docs/learn/deferrable-operators

But why?

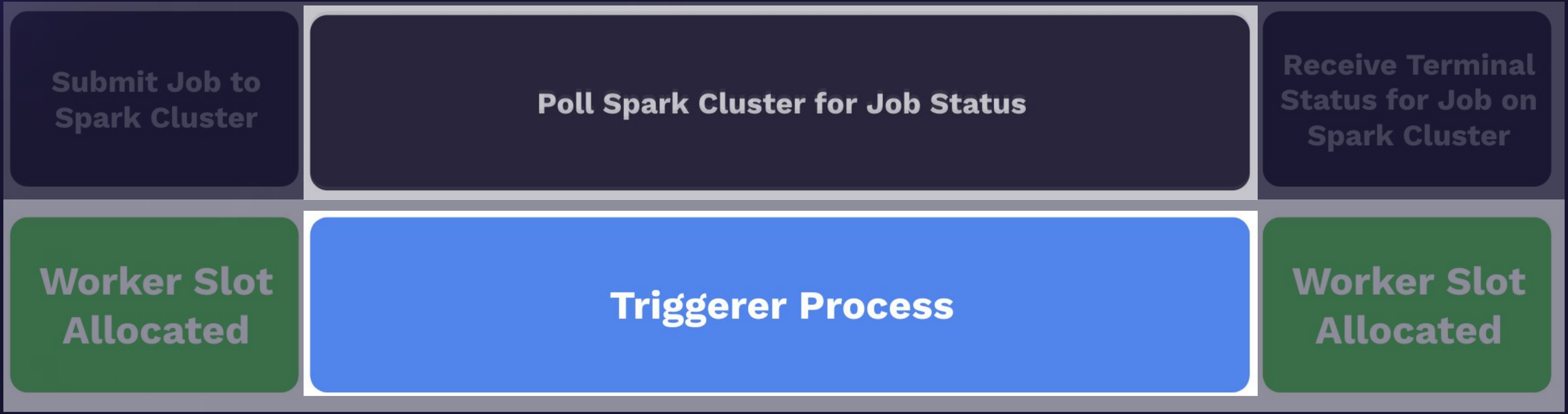
Non-Deferrable Operator


Submit Job to Spark Cluster
Poll Spark Cluster for Job Status
Receive Terminal Status for Job on Spark Cluster
Worker Slot Allocated

from: https://www.astronomer.io/docs/learn/deferrable-operators

But why? → Release worker slots

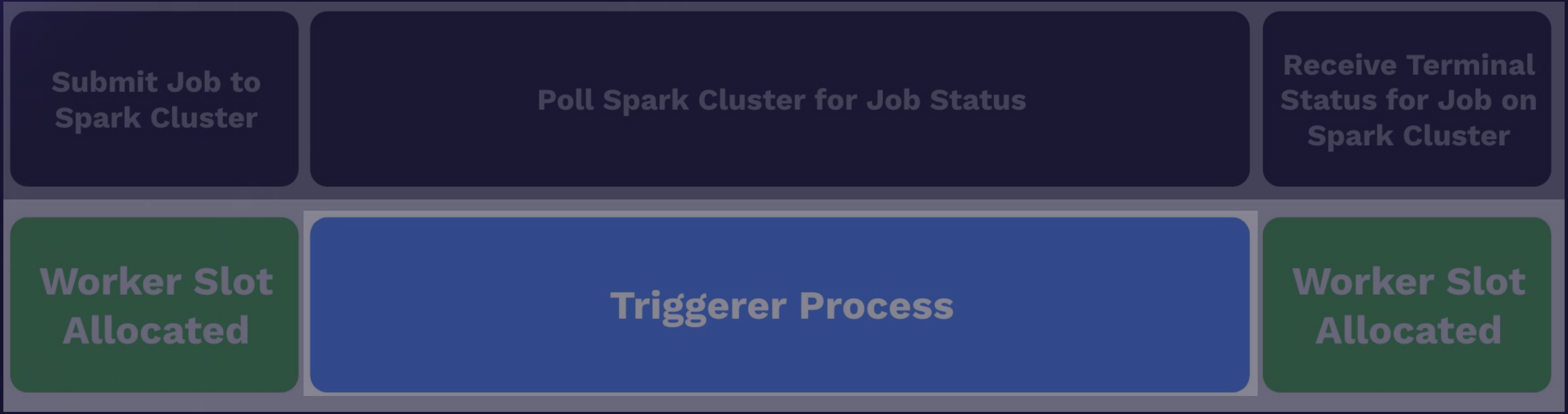
Deferrable Operator


Submit Job to Spark Cluster
Poll Spark Cluster for Job Status
Receive Terminal Status for Job on Spark Cluster
Worker Slot Allocated
Triggerer Process
Worker Slot Allocated

⌛ from: https://www.astronomer.io/docs/learn/deferrable-operators

But why? → Release worker slots

Deferrable Operator


Submit Job to Spark Cluster
Poll Spark Cluster for Job Status
Receive Terminal Status for Job on Spark Cluster
Worker Slot Allocated
Triggerer Process
Worker Slot Allocated

This page is for deferrable operators.

⏳ from: https://www.astronomer.io/docs/learn/deferrable-operators

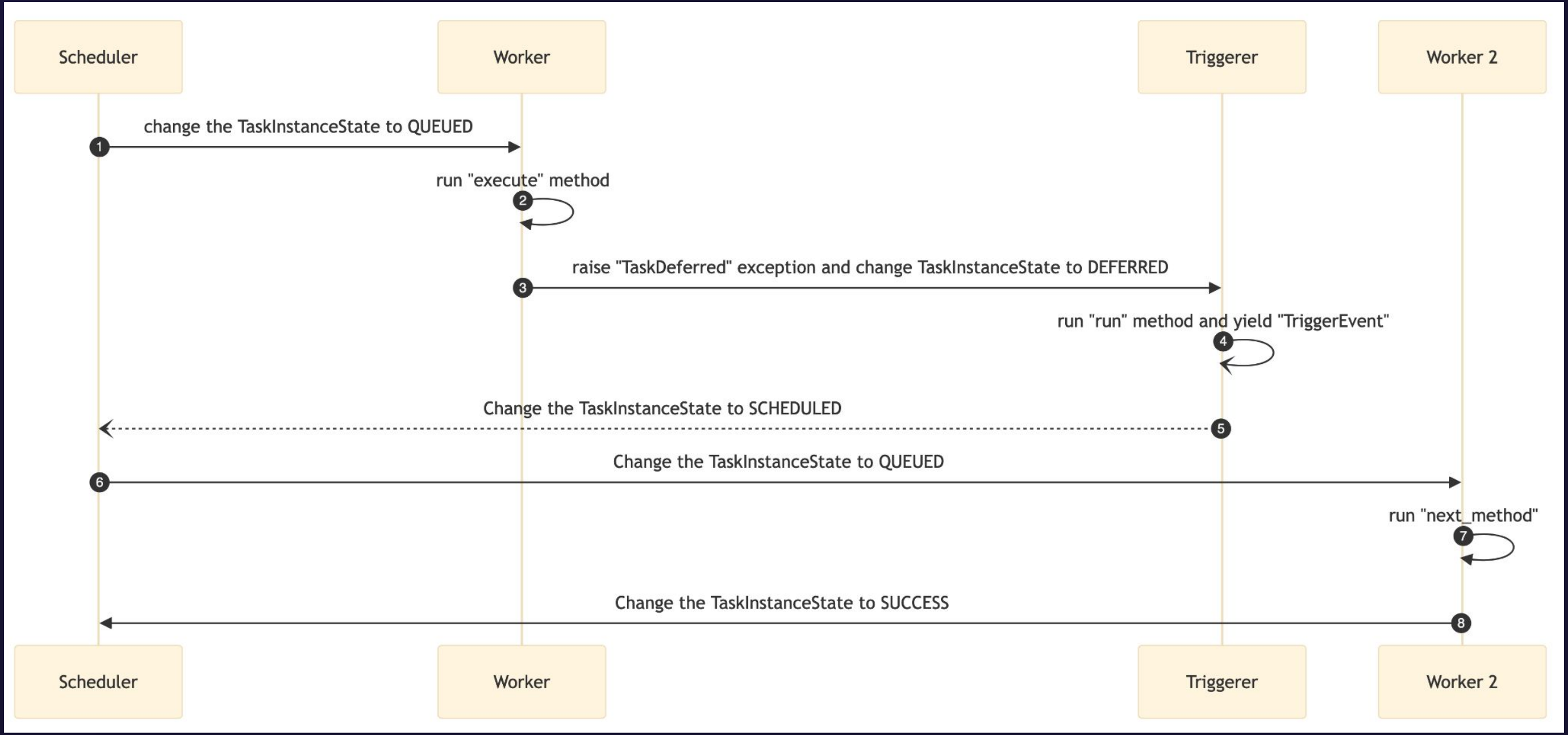
Release worker slots. And...?

Reduce resource usage

⏳

ASTRONOMER

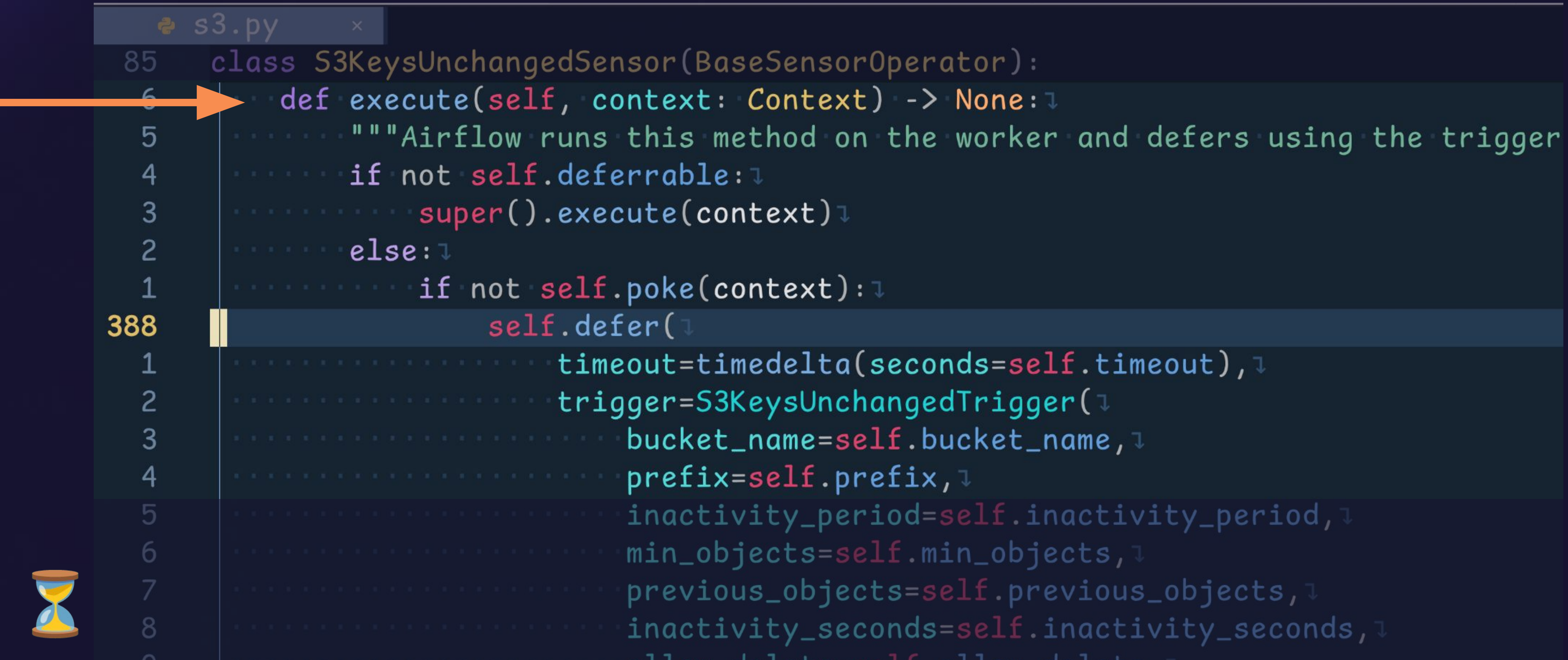
How does deferrable operators work?



Scheduler
Worker
Triggerer
Worker 2
1 change the TaskInstanceState to QUEUED
2 run "execute" method
3 raise "TaskDeferred" exception and change TaskInstanceState to DEFERRED
4 run "run" method and yield "TriggerEvent"
5 Change the TaskInstanceState to SCHEDULED
6 Change the TaskInstanceState to QUEUED
7 run "next_method"
8 Change the TaskInstanceState to SUCCESS
Scheduler
Worker
Triggerer
Worker 2

How does deferrable operator work?

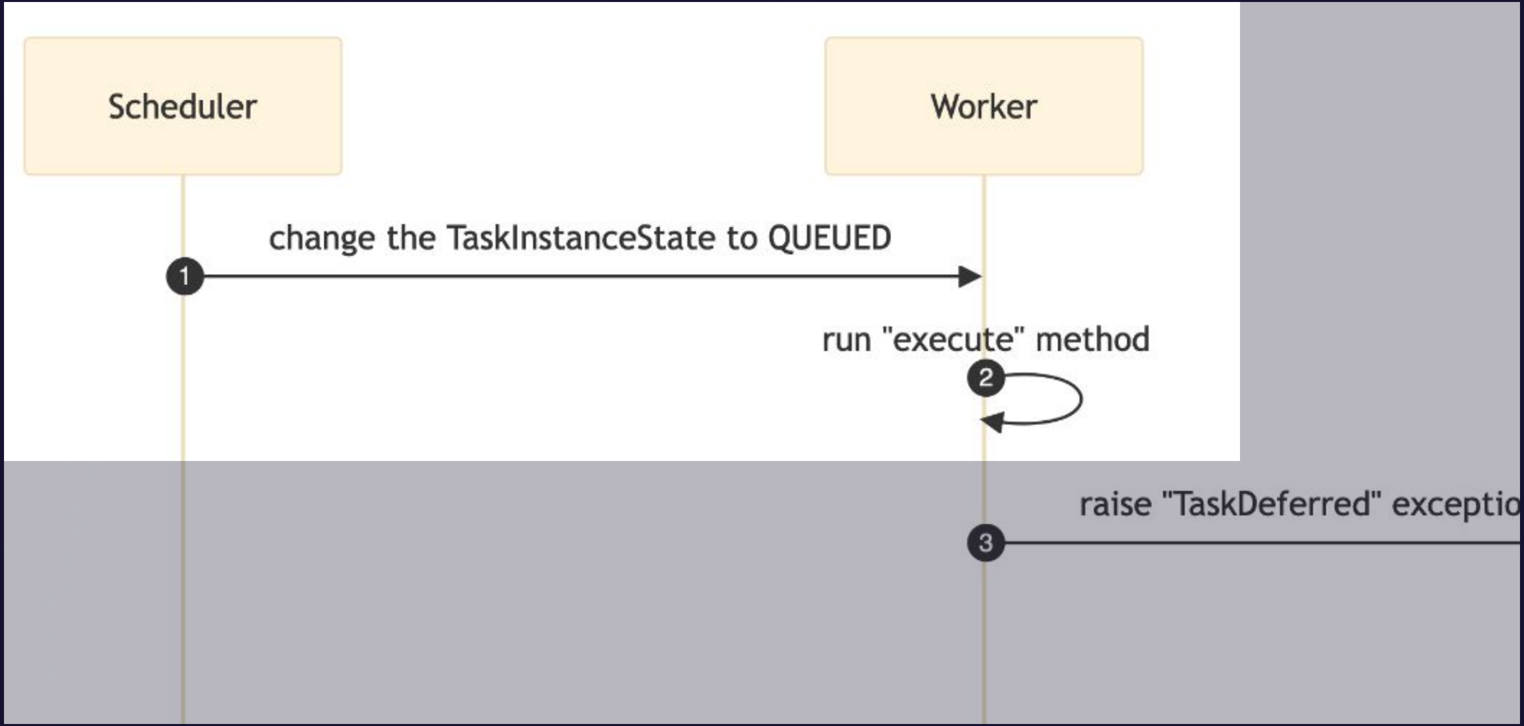
Well… we still run “execute” first.



s3.py

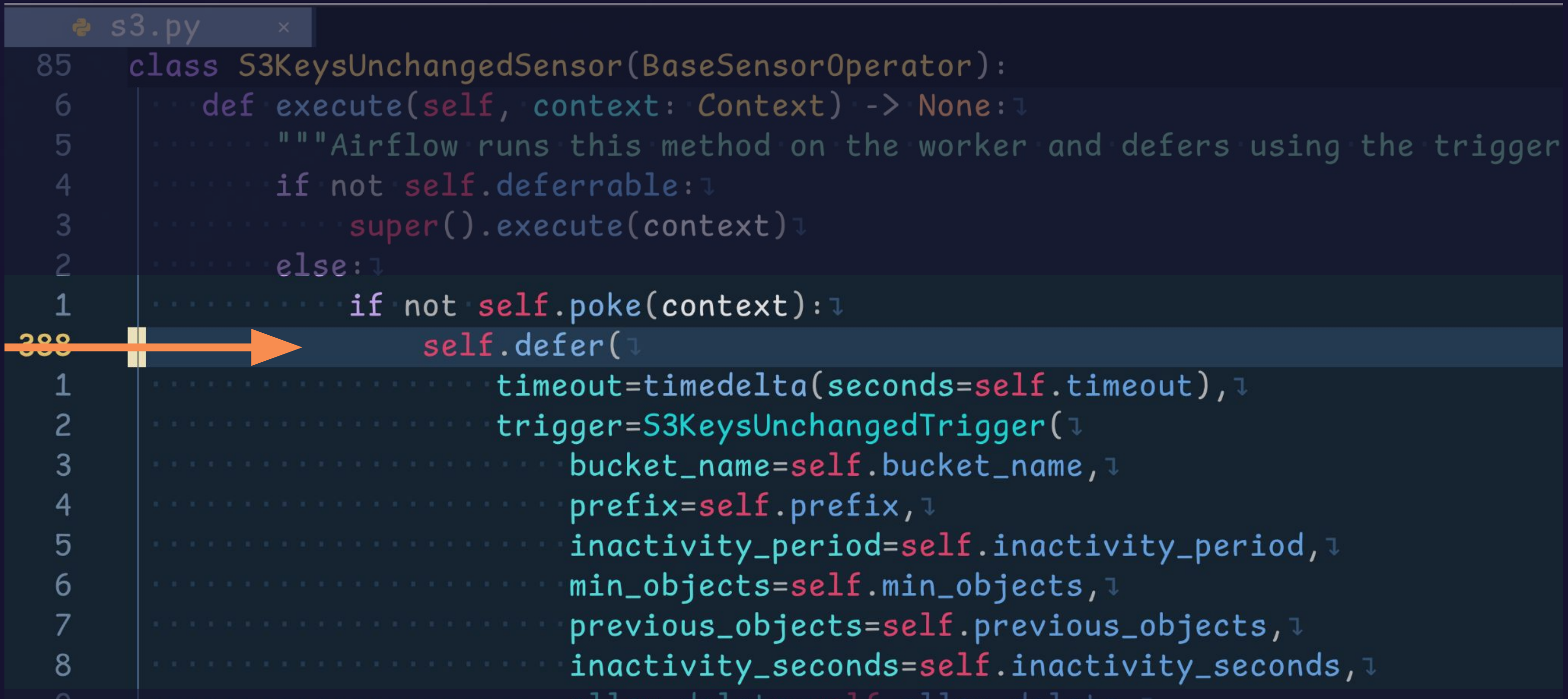
```python
class S3KeysUnchangedSensor(BaseSensorOperator):
    def execute(self, context: Context) -> None:
        """Airflow runs this method on the worker and defers using the trigger
        if not self.deferrable:
            super().execute(context)
        else:
            if not self.poke(context):
                self.defer(
                    timeout=timedelta(seconds=self.timeout),
                    trigger=S3KeysUnchangedTrigger(
                        bucket_name=self.bucket_name,
                        prefix=self.prefix,
                        inactivity_period=self.inactivity_period,
                        min_objects=self.min_objects,
                        previous_objects=self.previous_objects,
                        inactivity_seconds=self.inactivity_seconds,
```

How does deferrable operators work?


Scheduler
Worker
change the TaskInstanceState to QUEUED
1
run "execute" method
2
raise "TaskDeferred" exceptio
3

How does deferrable operator work?

The main difference is executing "self.defer" and raise a TaskDeferred exception through it

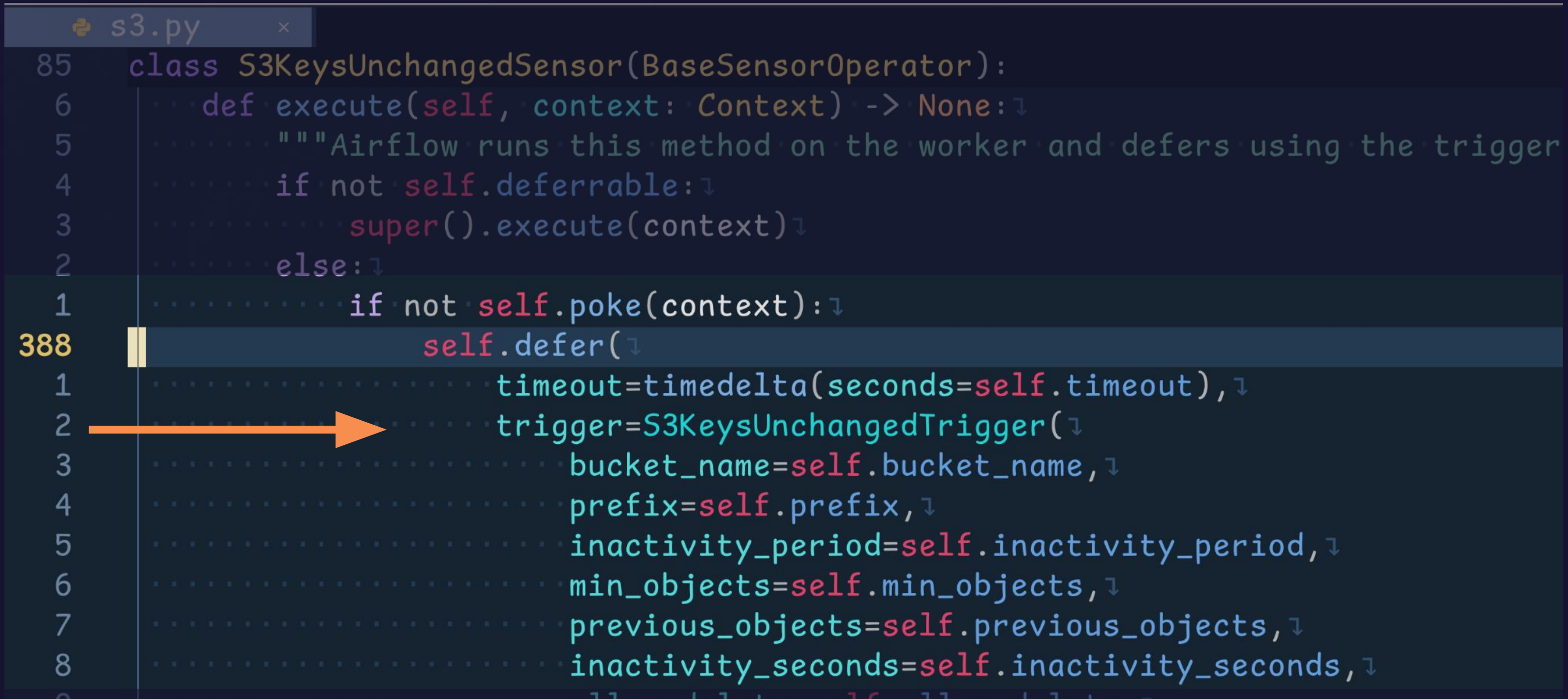


s3.py

```python
class S3KeysUnchangedSensor(BaseSensorOperator):
    def execute(self, context: Context) -> None:
        """Airflow runs this method on the worker and defers using the trigger
        if not self.deferrable:
            super().execute(context)
        else:
            if not self.poke(context):
                self.defer(
                    timeout=timedelta(seconds=self.timeout),
                    trigger=S3KeysUnchangedTrigger(
                        bucket_name=self.bucket_name,
                        prefix=self.prefix,
                        inactivity_period=self.inactivity_period,
                        min_objects=self.min_objects,
                        previous_objects=self.previous_objects,
                        inactivity_seconds=self.inactivity_seconds,
```

How does deferrable operator work?

Then, it's the triggerer's turn to run.

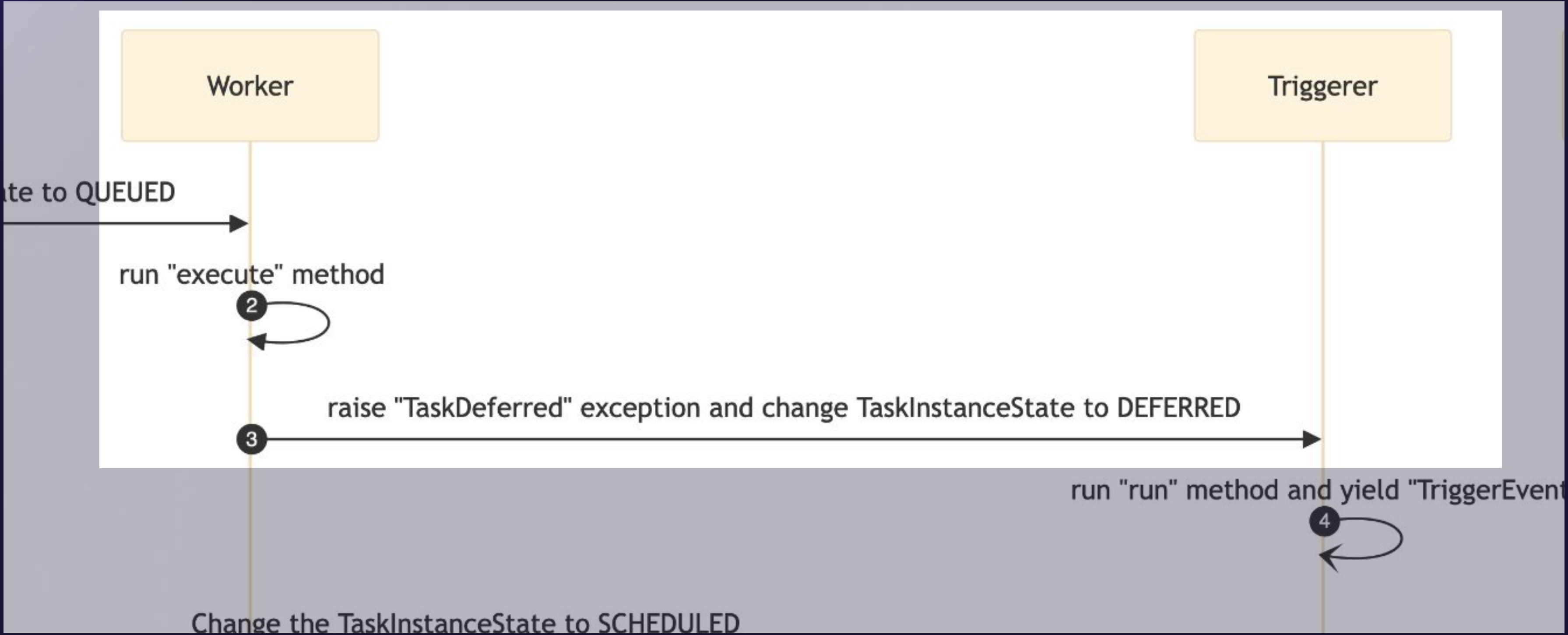


s3.py

```python
class S3KeysUnchangedSensor(BaseSensorOperator):
    def execute(self, context: Context) -> None:
        """Airflow runs this method on the worker and defers using the trigger
        if not self.deferrable:
            super().execute(context)
        else:
            if not self.poke(context):
                self.defer(
                    timeout=timedelta(seconds=self.timeout),
                    trigger=S3KeysUnchangedTrigger(
                        bucket_name=self.bucket_name,
                        prefix=self.prefix,
                        inactivity_period=self.inactivity_period,
                        min_objects=self.min_objects,
                        previous_objects=self.previous_objects,
                        inactivity_seconds=self.inactivity_seconds,
```

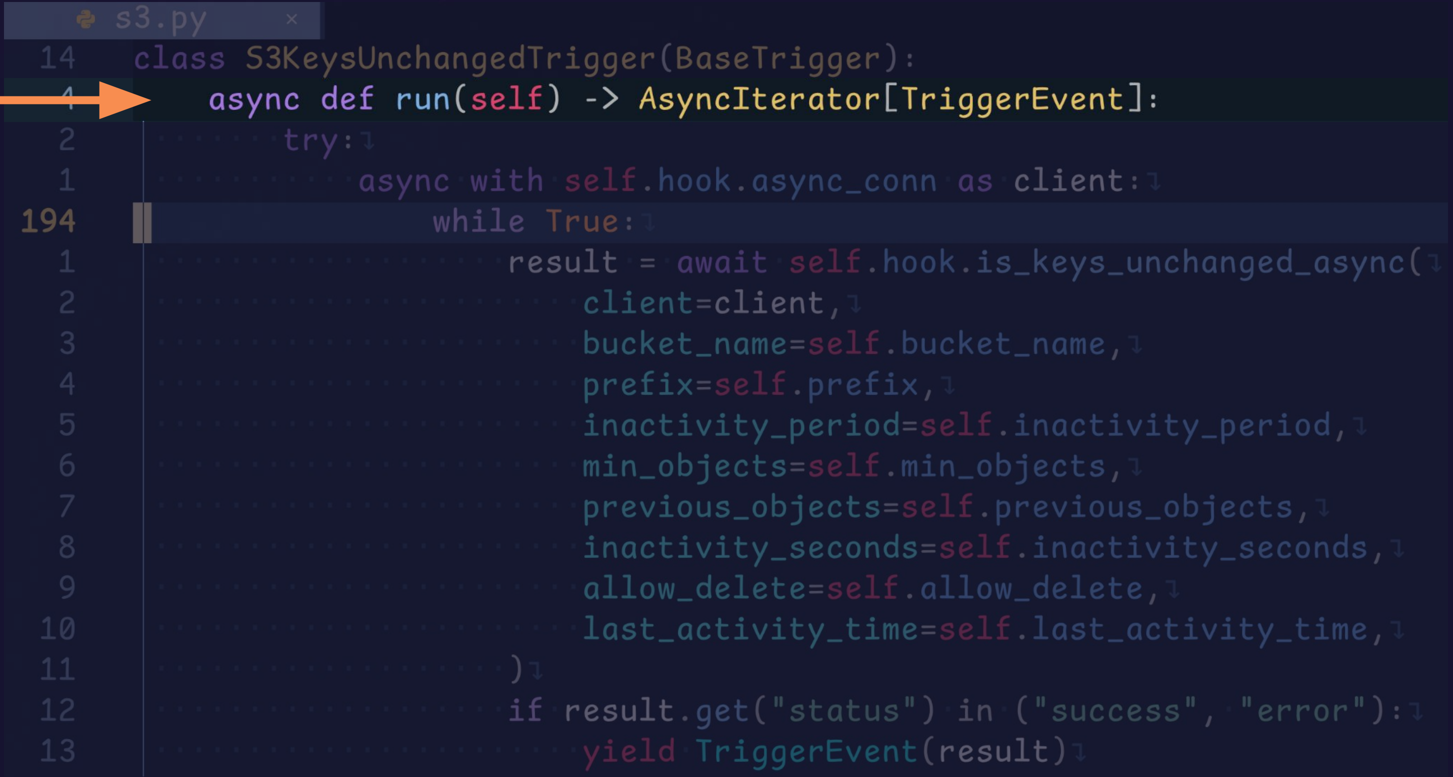
ASTRONOMER

How does deferrable operators work?


Worker
Triggerer
te to QUEUED
run "execute" method
2
raise "TaskDeferred" exception and change TaskInstanceState to DEFERRED
3
run "run" method and yield "TriggerEvent
4
Change the TaskInstanceState to SCHEDULED

How does deferrable operator work?

Execute the async "run" method in the triggerer

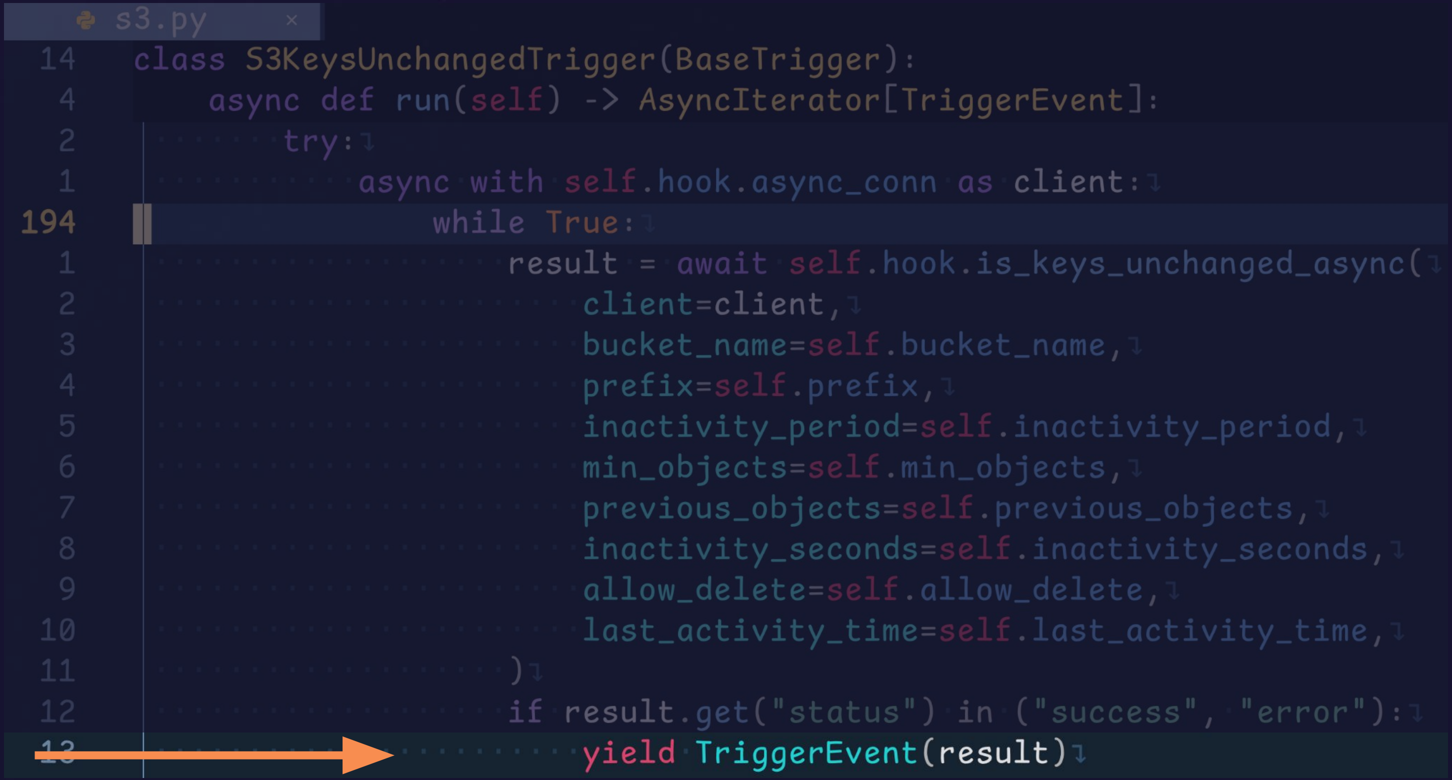


s3.py

```python
class S3KeysUnchangedTrigger(BaseTrigger):
    async def run(self) -> AsyncIterator[TriggerEvent]:
        try:
            async with self.hook.async_conn as client:
                while True:
                    result = await self.hook.is_keys_unchanged_async(
                        client=client,
                        bucket_name=self.bucket_name,
                        prefix=self.prefix,
                        inactivity_period=self.inactivity_period,
                        min_objects=self.min_objects,
                        previous_objects=self.previous_objects,
                        inactivity_seconds=self.inactivity_seconds,
                        allow_delete=self.allow_delete,
                        last_activity_time=self.last_activity_time,
                    )
                    if result.get("status") in ("success", "error"):
                        yield TriggerEvent(result)
```

How does deferrable operator work?

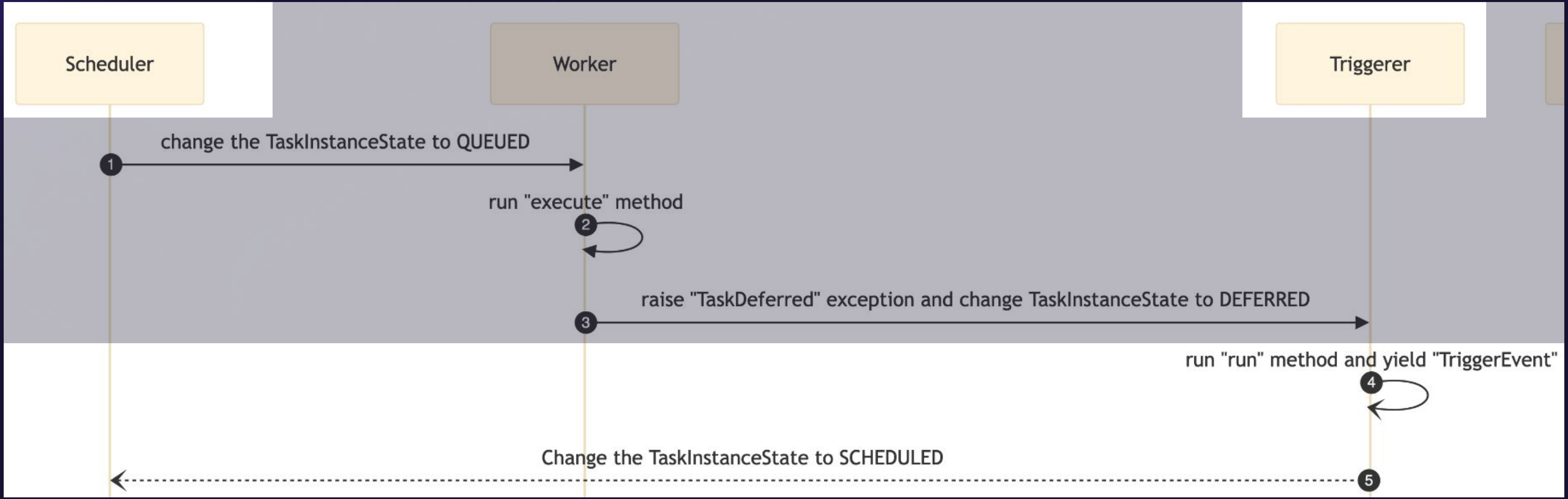
yield a TriggerEvent when it finished



s3.py

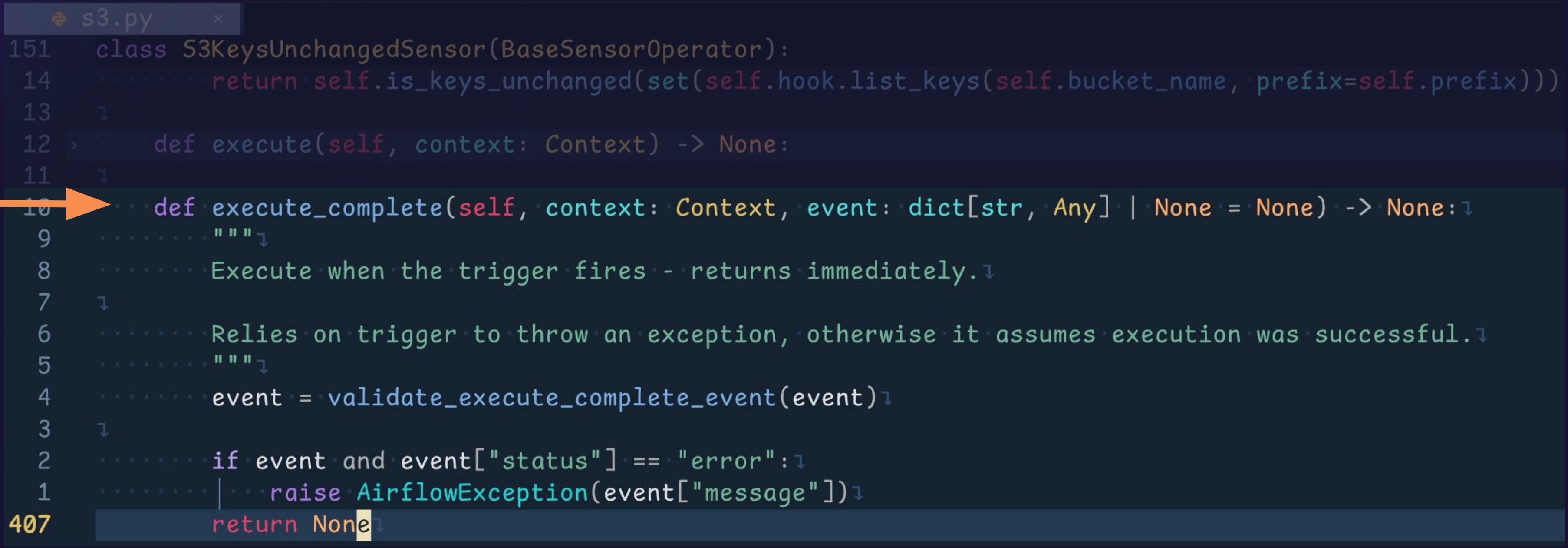
```
class S3KeysUnchangedTrigger(BaseTrigger):
    async def run(self) -> AsyncIterator[TriggerEvent]:
        try:
            async with self.hook.async_conn as client:
                while True:
                    result = await self.hook.is_keys_unchanged_async(
                        client=client,
                        bucket_name=self.bucket_name,
                        prefix=self.prefix,
                        inactivity_period=self.inactivity_period,
                        min_objects=self.min_objects,
                        previous_objects=self.previous_objects,
                        inactivity_seconds=self.inactivity_seconds,
                        allow_delete=self.allow_delete,
                        last_activity_time=self.last_activity_time,
                    )
                    if result.get("status") in ("success", "error"):
                        yield TriggerEvent(result)
```

How does deferrable operators work?


Scheduler
Worker
Triggerer
1
change the TaskInstanceState to QUEUED
2
run "execute" method
3
raise "TaskDeferred" exception and change TaskInstanceState to DEFERRED
4
run "run" method and yield "TriggerEvent"
5
Change the TaskInstanceState to SCHEDULED

How does deferrable operator work?

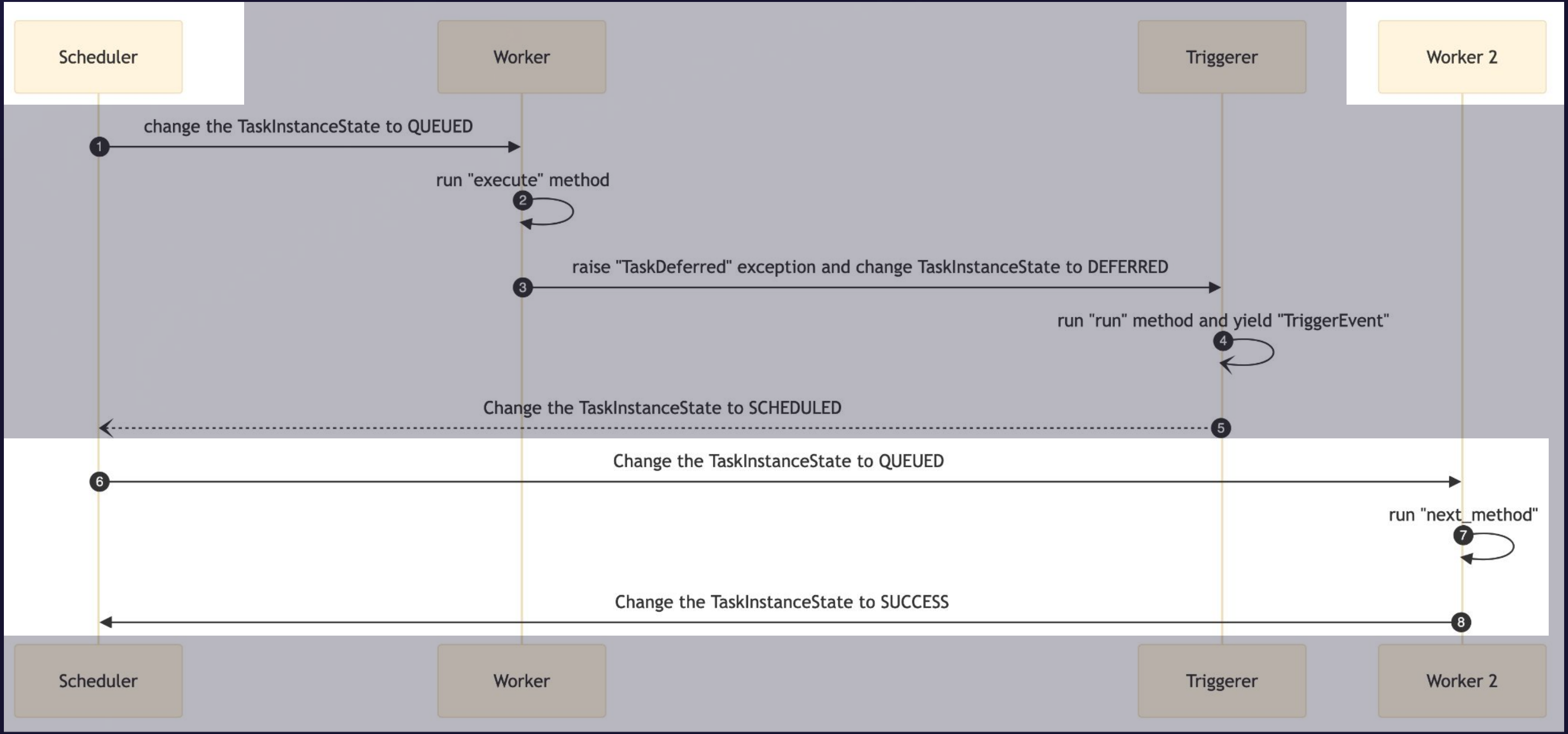
Execute the "execute_complete" method in another worker



```
s3.py

class S3KeysUnchangedSensor(BaseSensorOperator):
        return self.is_keys_unchanged(set(self.hook.list_keys(self.bucket_name, prefix=self.prefix)))

    def execute(self, context: Context) -> None:

    def execute_complete(self, context: Context, event: dict[str, Any] | None = None) -> None:
        """
        Execute when the trigger fires - returns immediately.

        Relies on trigger to throw an exception, otherwise it assumes execution was successful.
        """
        event = validate_execute_complete_event(event)

        if event and event["status"] == "error":
            raise AirflowException(event["message"])
        return None
```

ASTRONOMER

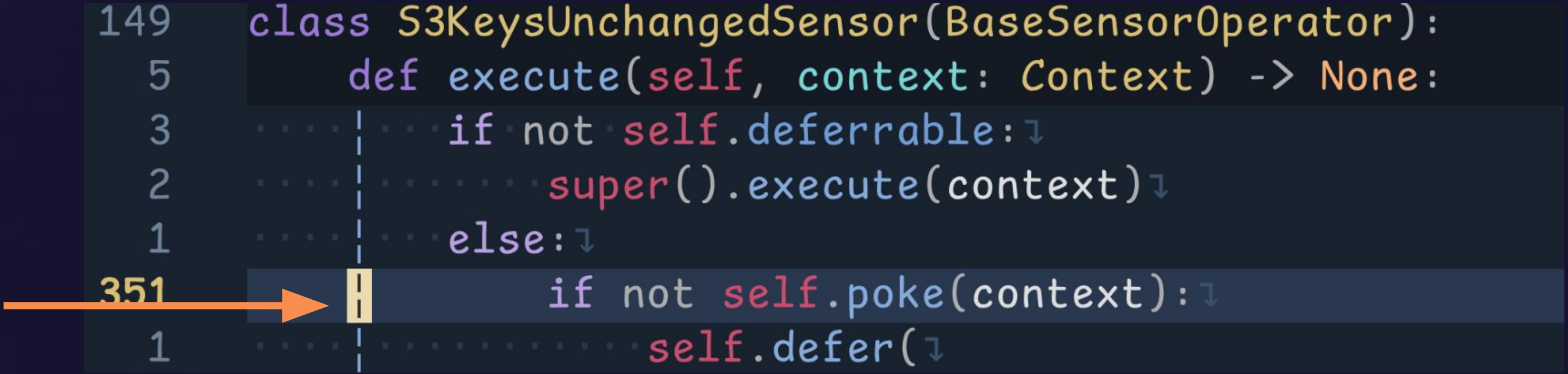
How does deferrable operators work?


Scheduler
Worker
Triggerer
Worker 2
1 change the TaskInstanceState to QUEUED
2 run "execute" method
3 raise "TaskDeferred" exception and change TaskInstanceState to DEFERRED
4 run "run" method and yield "TriggerEvent"
5 Change the TaskInstanceState to SCHEDULED
6 Change the TaskInstanceState to QUEUED
7 run "next_method"
8 Change the TaskInstanceState to SUCCESS
Scheduler
Worker
Triggerer
Worker 2

ASTRONOMER

Do we really need to run it in the worker first?

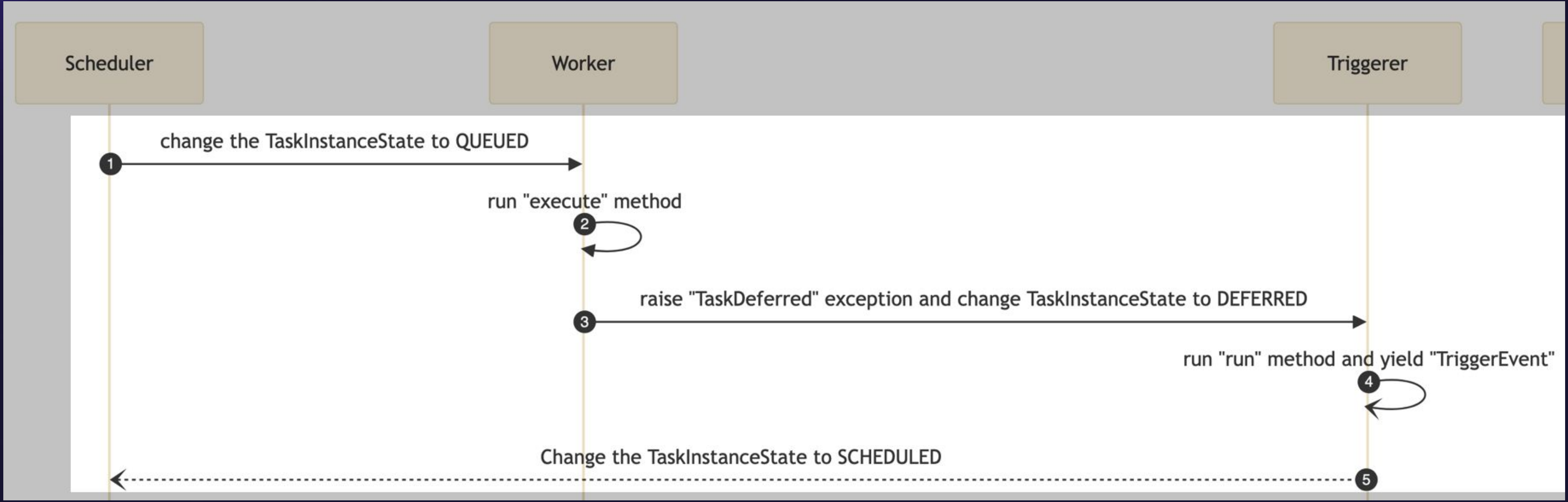
the only logic before deferring



```python
class S3KeysUnchangedSensor(BaseSensorOperator):
    def execute(self, context: Context) -> None:
        if not self.deferrable:
            super().execute(context)
        else:
            if not self.poke(context):
                self.defer(
```

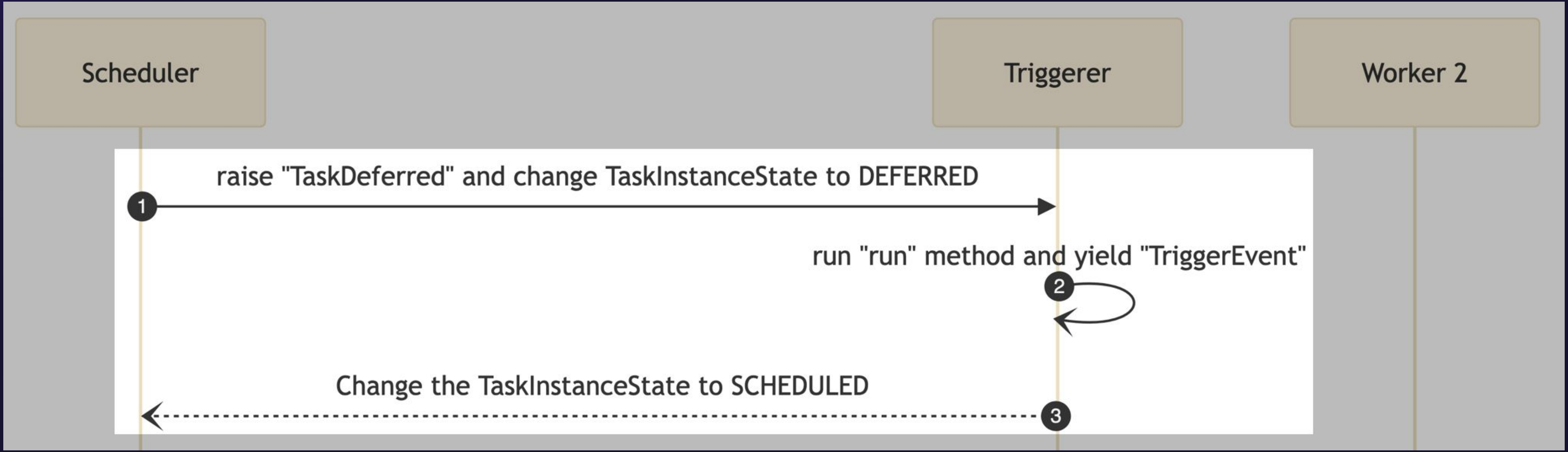
I DON'T THINK SO, SIR.

What can we do?


Scheduler
Worker
Triggerer
1 change the TaskInstanceState to QUEUED
2 run "execute" method
3 raise "TaskDeferred" exception and change TaskInstanceState to DEFERRED
4 run "run" method and yield "TriggerEvent"
5 Change the TaskInstanceState to SCHEDULED

What can we do?

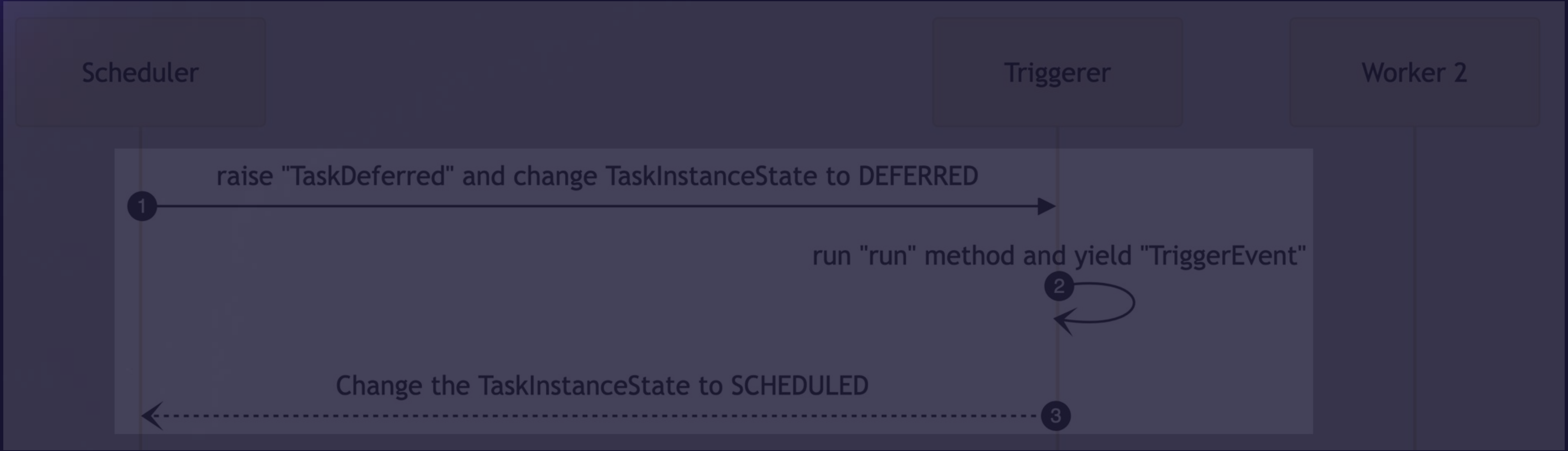
Start task execution in the triggerer


Scheduler
Triggerer
Worker 2
1
raise "TaskDeferred" and change TaskInstanceState to DEFERRED
2
run "run" method and yield "TriggerEvent"
3
Change the TaskInstanceState to SCHEDULED

ASTRONOMER

What can we do?

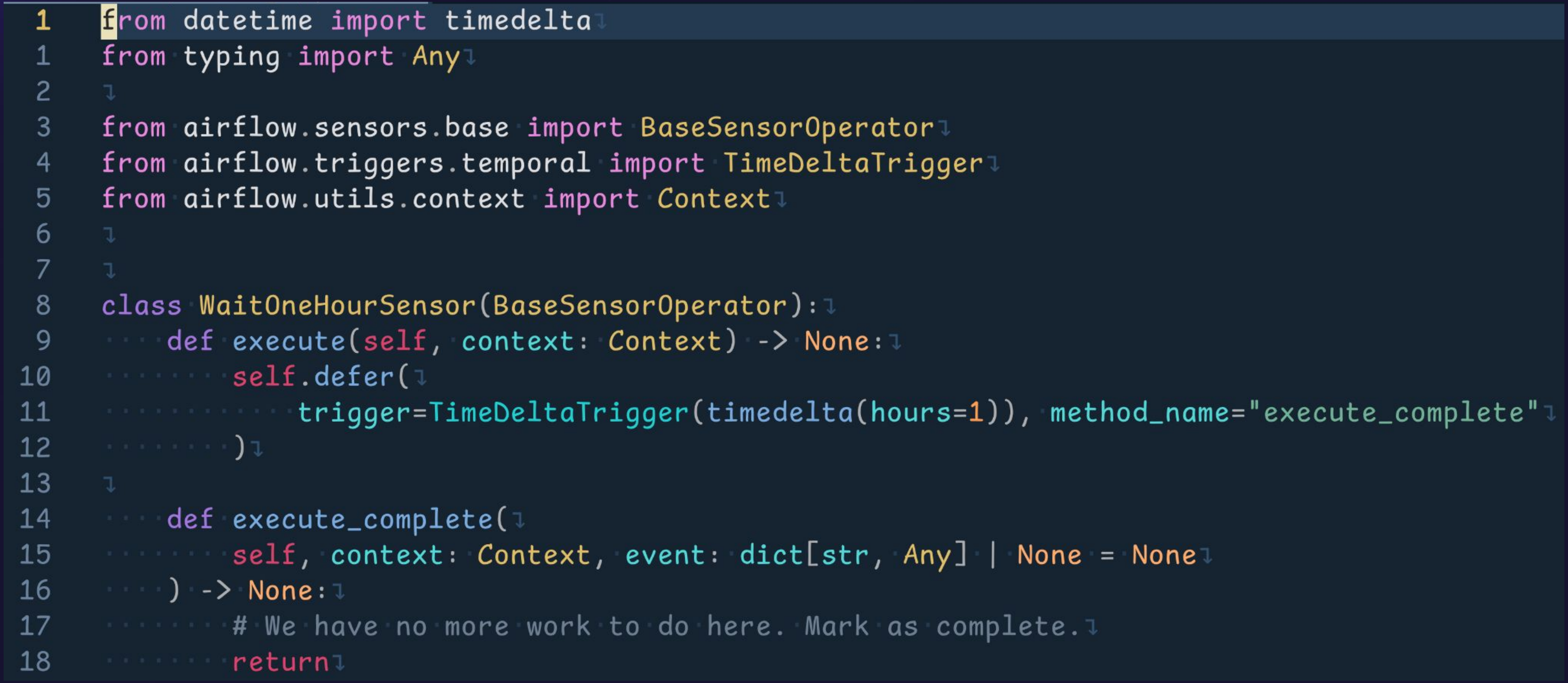
Start task execution in the triggerer


Scheduler
Triggerer
Worker 2
raise "TaskDeferred" and change TaskInstanceState to DEFERRED
1
run "run" method and yield "TriggerEvent"
2
Change the TaskInstanceState to SCHEDULED
3

This page is for new features.

ASTRONOMER

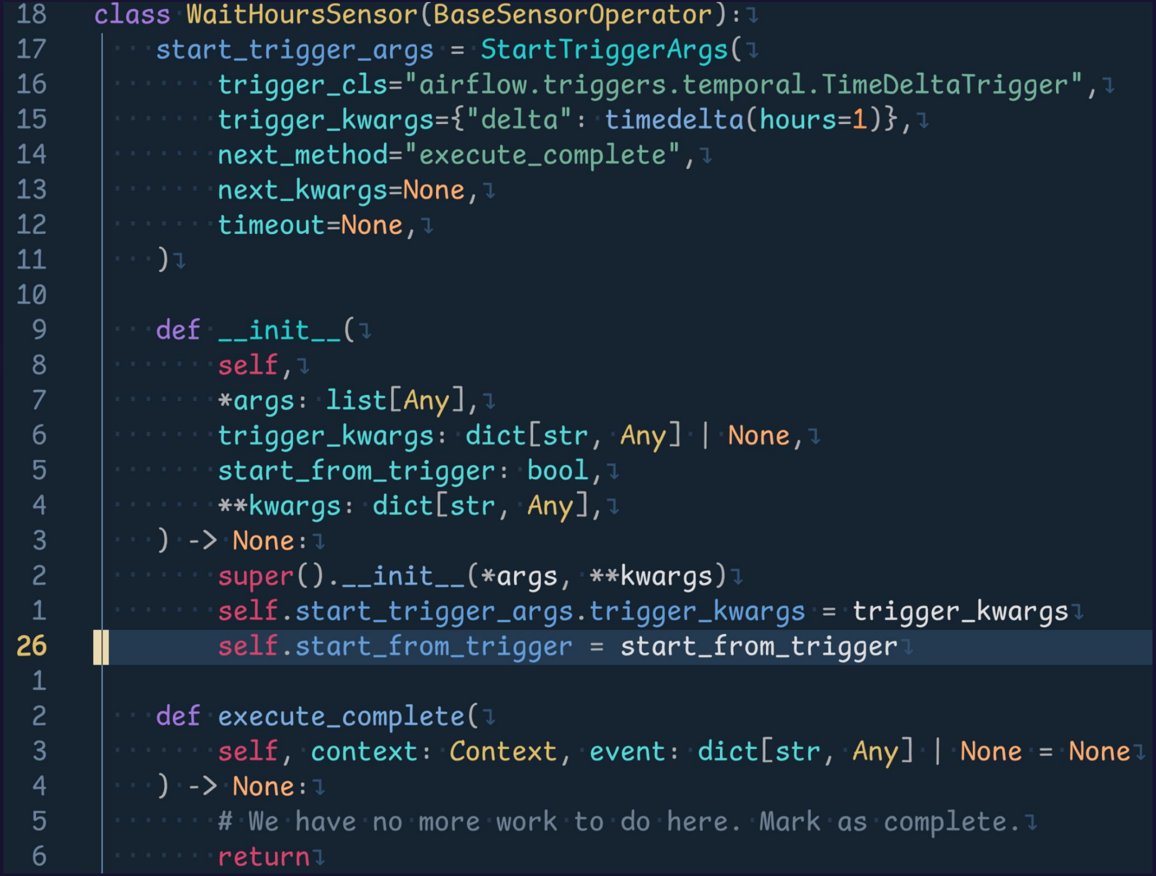
Start task execution in the worker



```python
from datetime import timedelta
from typing import Any

from airflow.sensors.base import BaseSensorOperator
from airflow.triggers.temporal import TimeDeltaTrigger
from airflow.utils.context import Context


class WaitOneHourSensor(BaseSensorOperator):
    def execute(self, context: Context) -> None:
        self.defer(
            trigger=TimeDeltaTrigger(timedelta(hours=1)), method_name="execute_complete"
        )

    def execute_complete(
        self, context: Context, event: dict[str, Any] | None = None
    ) -> None:
        # We have no more work to do here. Mark as complete.
        return
```

ASTRONOMER

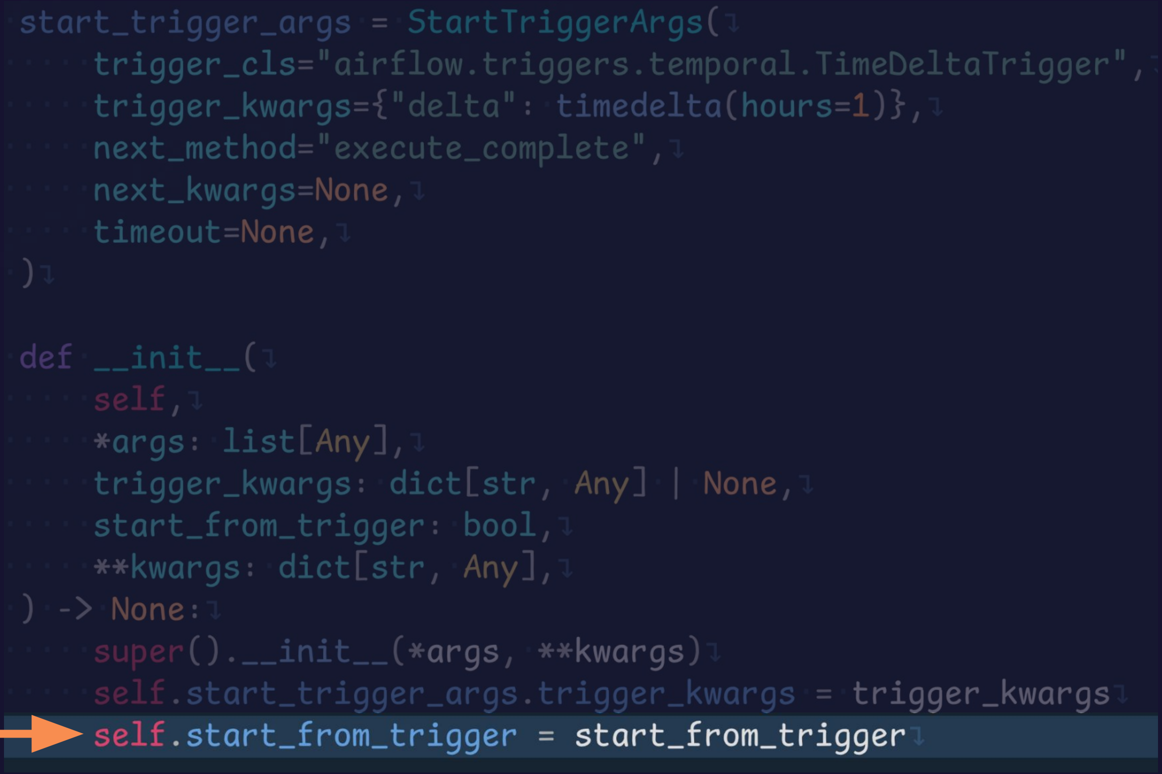
Start task execution in the triggerer



```
class WaitHoursSensor(BaseSensorOperator):
    start_trigger_args = StartTriggerArgs(
        trigger_cls="airflow.triggers.temporal.TimeDeltaTrigger",
        trigger_kwargs={"delta": timedelta(hours=1)},
        next_method="execute_complete",
        next_kwargs=None,
        timeout=None,
    )

    def __init__(
        self,
        *args: list[Any],
        trigger_kwargs: dict[str, Any] | None,
        start_from_trigger: bool,
        **kwargs: dict[str, Any],
    ) -> None:
        super().__init__(*args, **kwargs)
        self.start_trigger_args.trigger_kwargs = trigger_kwargs
        self.start_from_trigger = start_from_trigger

    def execute_complete(
        self, context: Context, event: dict[str, Any] | None = None
    ) -> None:
        # We have no more work to do here. Mark as complete.
        return
```

Start task execution in the triggerer

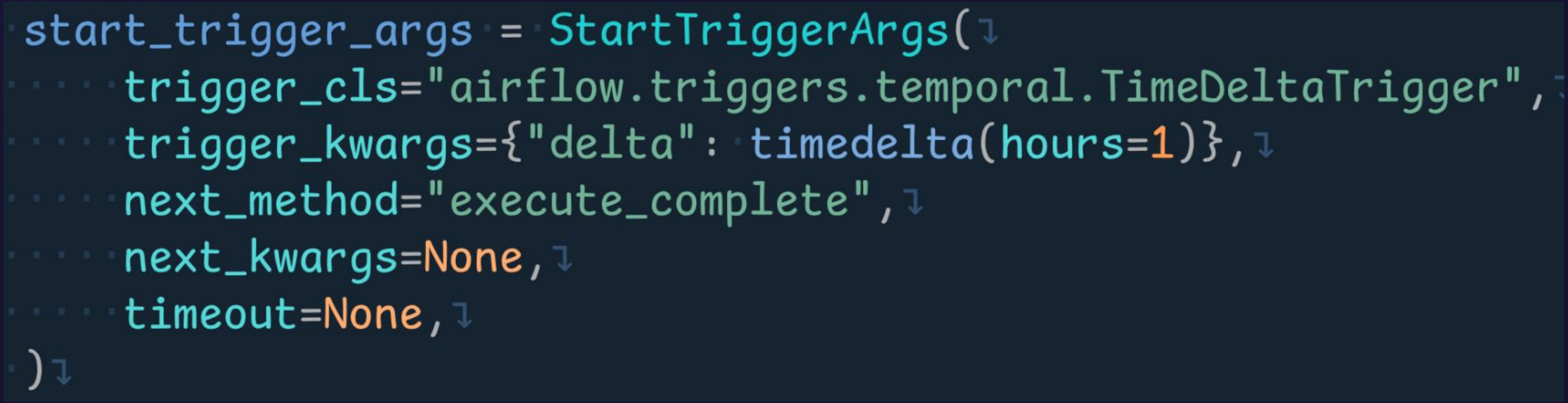
StartTriggerArgs and start_from_trigger



```python
start_trigger_args = StartTriggerArgs(
    trigger_cls="airflow.triggers.temporal.TimeDeltaTrigger",
    trigger_kwargs={"delta": timedelta(hours=1)},
    next_method="execute_complete",
    next_kwargs=None,
    timeout=None,
)

def __init__(
    self,
    *args: list[Any],
    trigger_kwargs: dict[str, Any] | None,
    start_from_trigger: bool,
    **kwargs: dict[str, Any],
) -> None:
    super().__init__(*args, **kwargs)
    self.start_trigger_args.trigger_kwargs = trigger_kwargs
    self.start_from_trigger = start_from_trigger
```

Start task execution in the triggerer

StartTriggerArgs and start_from_trigger

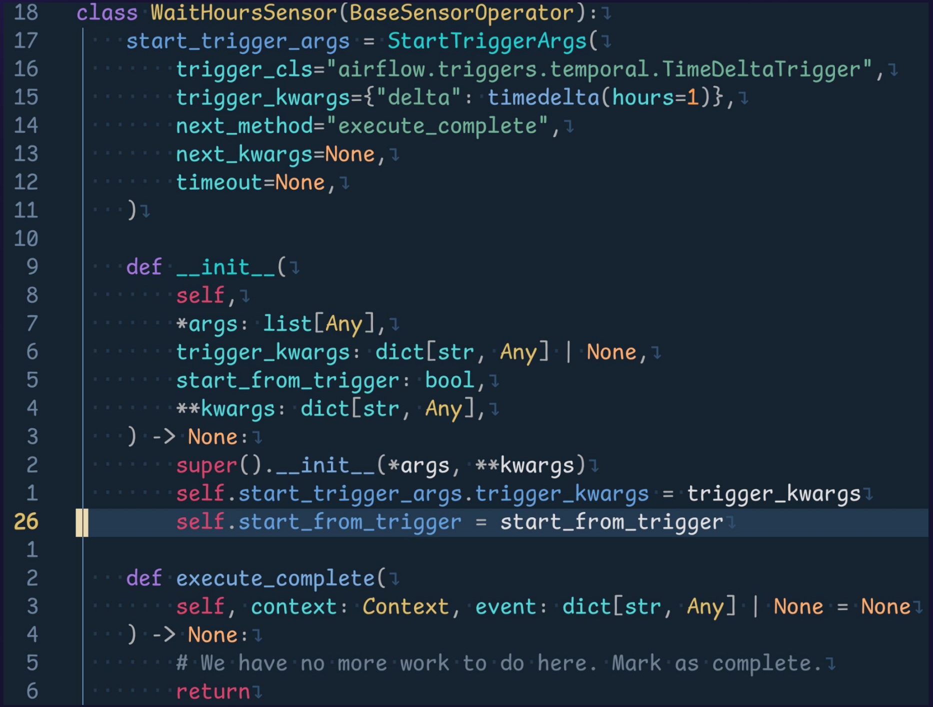


```python
start_trigger_args = StartTriggerArgs(
    trigger_cls="airflow.triggers.temporal.TimeDeltaTrigger",
    trigger_kwargs={"delta": timedelta(hours=1)},
    next_method="execute_complete",
    next_kwargs=None,
    timeout=None,
)
```

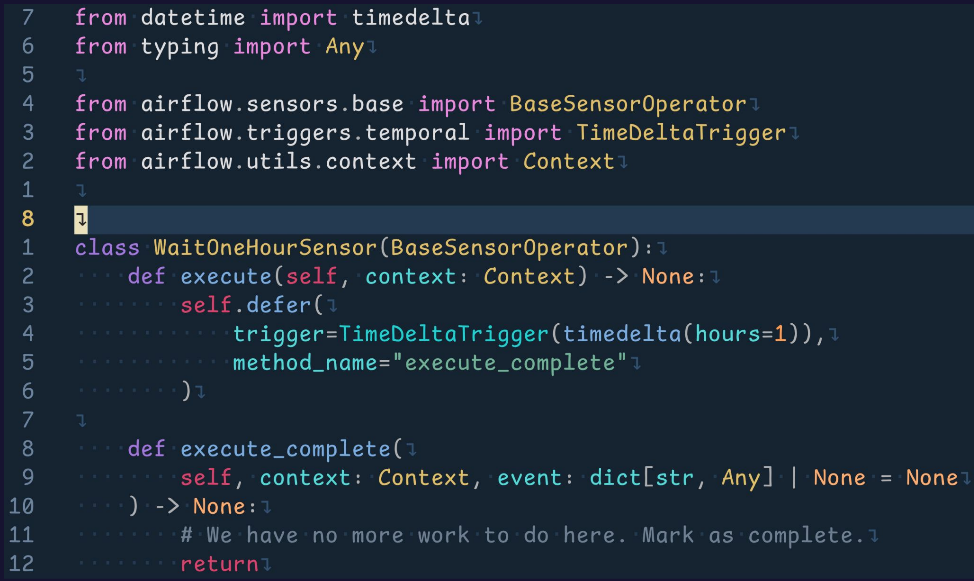
Start task execution in the triggerer

StartTriggerArgs vs self.defer

⏳



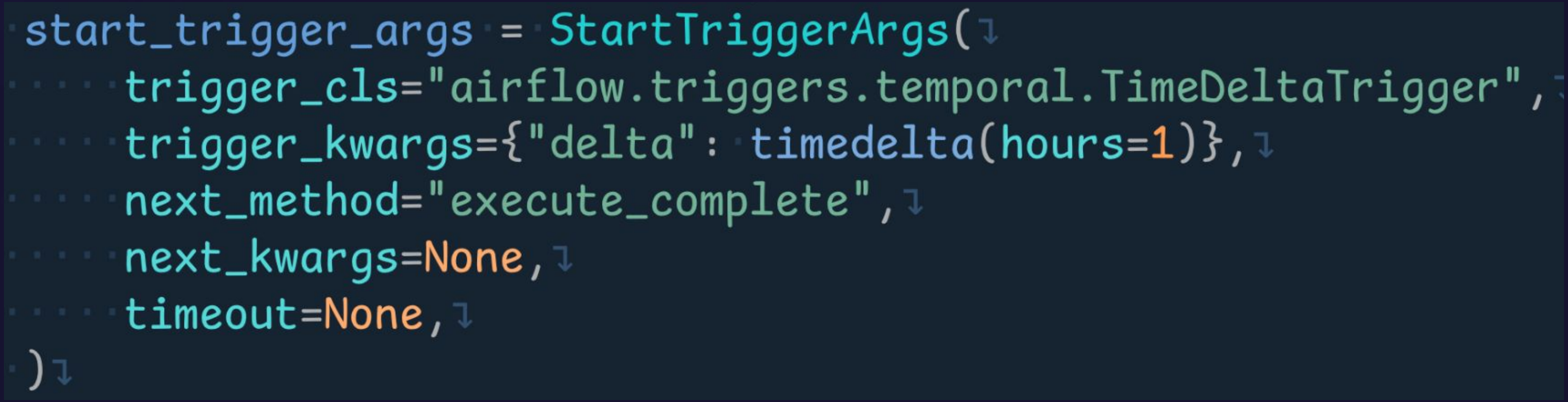
```python
class WaitHoursSensor(BaseSensorOperator):
    start_trigger_args = StartTriggerArgs(
        trigger_cls="airflow.triggers.temporal.TimeDeltaTrigger",
        trigger_kwargs={"delta": timedelta(hours=1)},
        next_method="execute_complete",
        next_kwargs=None,
        timeout=None,
    )

    def __init__(
        self,
        *args: list[Any],
        trigger_kwargs: dict[str, Any] | None,
        start_from_trigger: bool,
        **kwargs: dict[str, Any],
    ) -> None:
        super().__init__(*args, **kwargs)
        self.start_trigger_args.trigger_kwargs = trigger_kwargs
        self.start_from_trigger = start_from_trigger

    def execute_complete(
        self, context: Context, event: dict[str, Any] | None = None
    ) -> None:
        # We have no more work to do here. Mark as complete.
        return
```



```python
from datetime import timedelta
from typing import Any

from airflow.sensors.base import BaseSensorOperator
from airflow.triggers.temporal import TimeDeltaTrigger
from airflow.utils.context import Context


class WaitOneHourSensor(BaseSensorOperator):
    def execute(self, context: Context) -> None:
        self.defer(
            trigger=TimeDeltaTrigger(timedelta(hours=1)),
            method_name="execute_complete"
        )

    def execute_complete(
        self, context: Context, event: dict[str, Any] | None = None
    ) -> None:
        # We have no more work to do here. Mark as complete.
        return
```

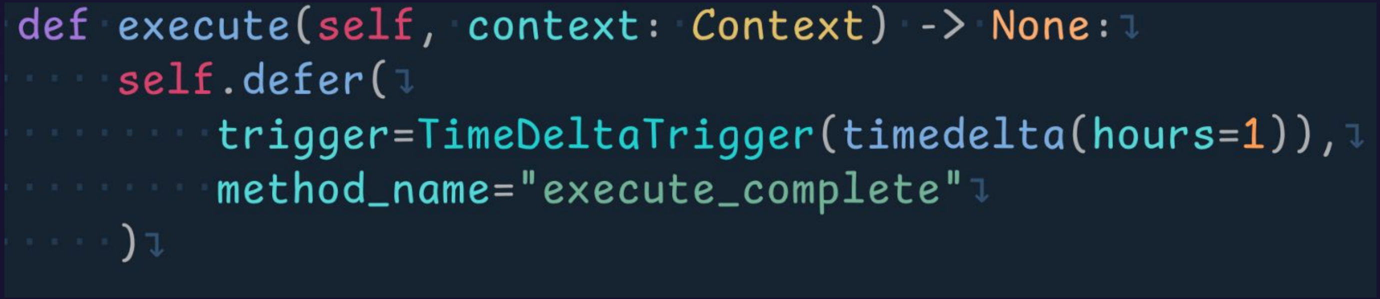
Start task execution in the triggerer

StartTriggerArgs vs self.defer



```python
start_trigger_args = StartTriggerArgs(
    trigger_cls="airflow.triggers.temporal.TimeDeltaTrigger",
    trigger_kwargs={"delta": timedelta(hours=1)},
    next_method="execute_complete",
    next_kwargs=None,
    timeout=None,
)
```

⏳



```python
def execute(self, context: Context) -> None:
    self.defer(
        trigger=TimeDeltaTrigger(timedelta(hours=1)),
        method_name="execute_complete"
    )
```

Start task execution in the triggerer

trigger_cls

```python
start_trigger_args = StartTriggerArgs(
    trigger_cls="airflow.triggers.temporal.TimeDeltaTrigger",
    trigger_kwargs={"delta": timedelta(hours=1)},
    next_method="execute_complete",
    next_kwargs=None,
    timeout=None,
)
```

⏳

```python
def execute(self, context: Context) -> None:
    self.defer(
        trigger=TimeDeltaTrigger(timedelta(hours=1)),
        method_name="execute_complete"
    )
```

ASTRONOMER

Start task execution in the triggerer

trigger_kwargs

```
start_trigger_args = StartTriggerArgs(
    trigger_cls="airflow.triggers.temporal.TimeDeltaTrigger",
    trigger_kwargs={"delta": timedelta(hours=1)},
    next_method="execute_complete",
    next_kwargs=None,
    timeout=None,
)
```

⏳

```
def execute(self, context: Context) -> None:
    self.defer(
        trigger=TimeDeltaTrigger(timedelta(hours=1)),
        method_name="execute_complete"
    )
```

Start task execution in the triggerer

next_method

```python
start_trigger_args = StartTriggerArgs(
    trigger_cls="airflow.triggers.temporal.TimeDeltaTrigger",
    trigger_kwargs={"delta": timedelta(hours=1)},
    next_method="execute_complete",
    next_kwargs=None,
    timeout=None,
)
```

⏳

```python
def execute(self, context: Context) -> None:
    self.defer(
        trigger=TimeDeltaTrigger(timedelta(hours=1)),
        method_name="execute_complete"
    )
```

Start task execution in the triggerer

args with default values

```
start_trigger_args = StartTriggerArgs(
    trigger_cls="airflow.triggers.temporal.TimeDeltaTrigger",
    trigger_kwargs={"delta": timedelta(hours=1)},
    next_method="execute_complete",
    next_kwargs=None,
    timeout=None,
)
```

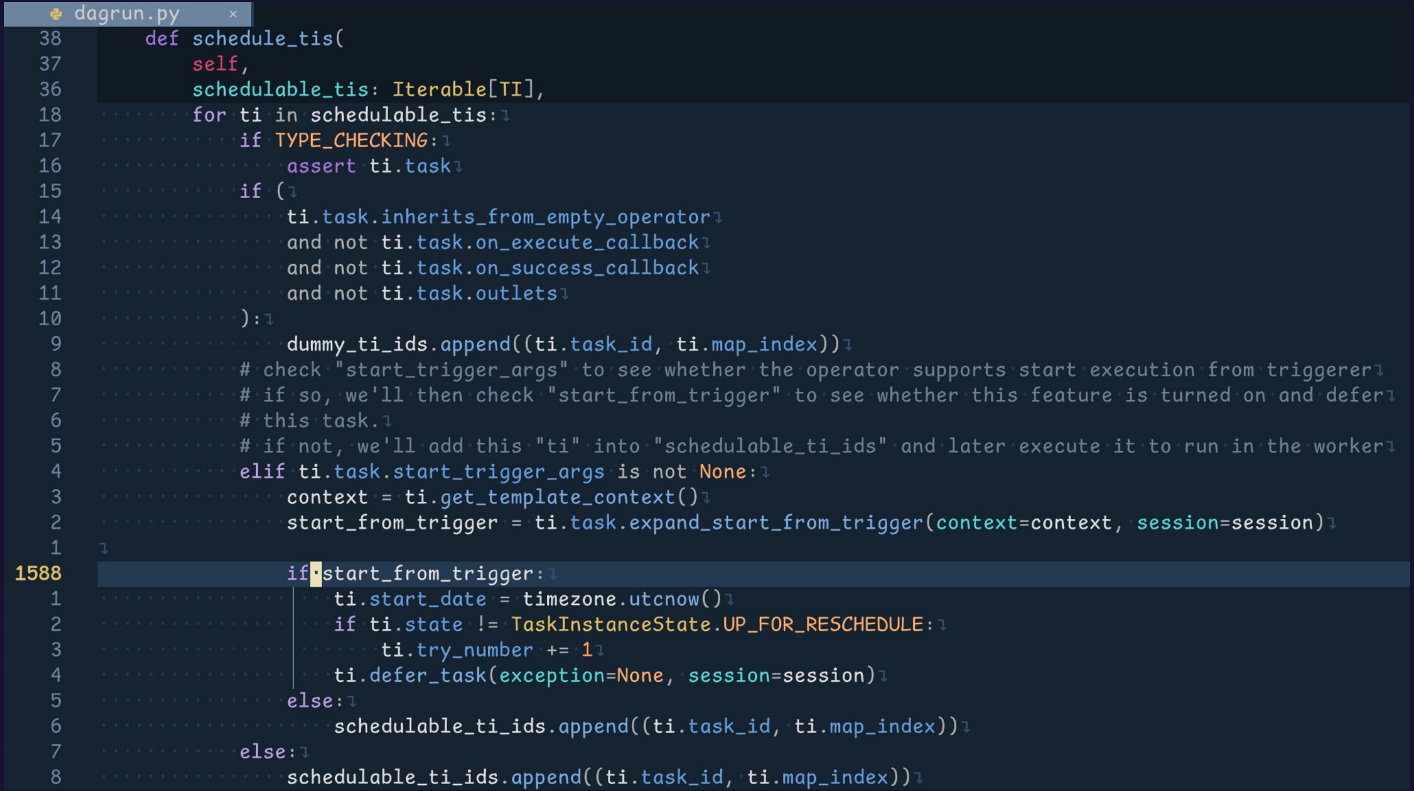
⏳

```
def execute(self, context: Context) -> None:
    self.defer(
        trigger=TimeDeltaTrigger(timedelta(hours=1)),
        method_name="execute_complete"
    )
```

ASTRONOMER

Start task execution in the triggerer

Under the hood

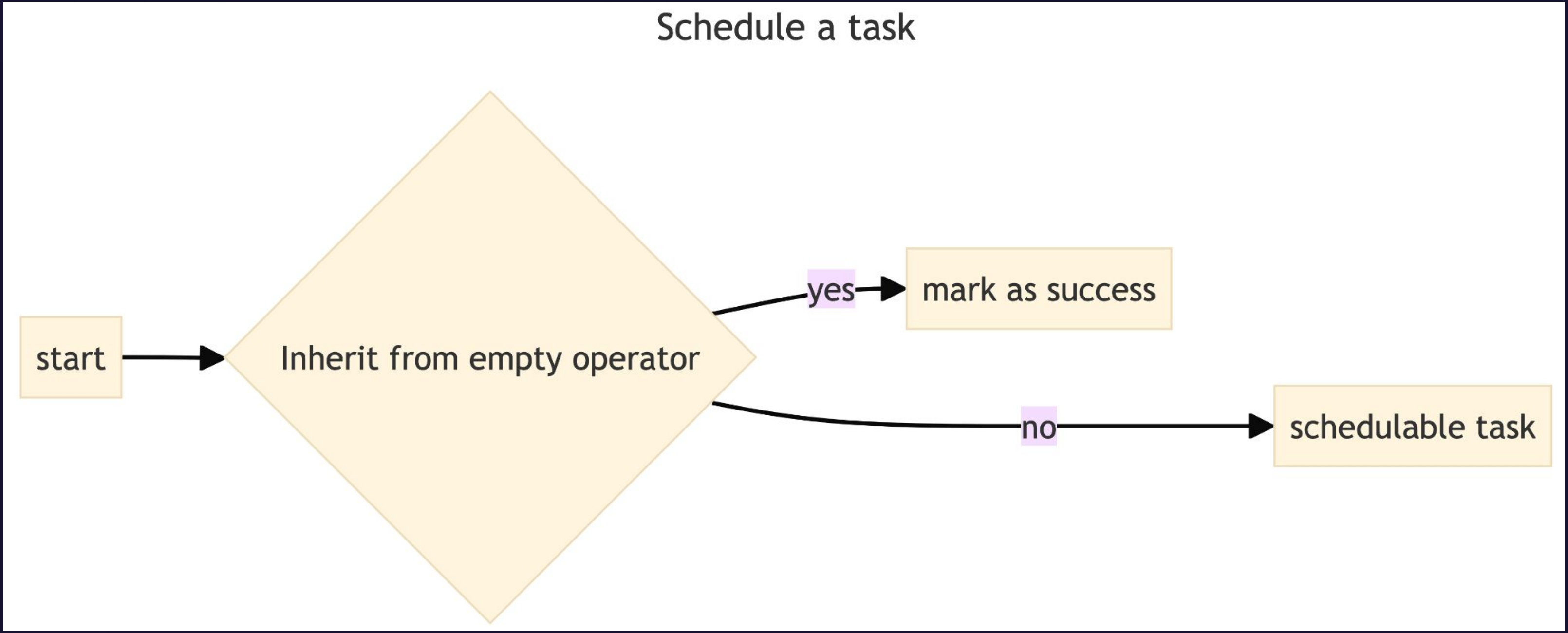


dagrun.py

```python
    def schedule_tis(
        self,
        schedulable_tis: Iterable[TI],
        for ti in schedulable_tis:
            if TYPE_CHECKING:
                assert ti.task
            if (
                ti.task.inherits_from_empty_operator
                and not ti.task.on_execute_callback
                and not ti.task.on_success_callback
                and not ti.task.outlets
            ):
                dummy_ti_ids.append((ti.task_id, ti.map_index))
            # check "start_trigger_args" to see whether the operator supports start execution from triggerer
            # if so, we'll then check "start_from_trigger" to see whether this feature is turned on and defer
            # this task.
            # if not, we'll add this "ti" into "schedulable_ti_ids" and later execute it to run in the worker
            elif ti.task.start_trigger_args is not None:
                context = ti.get_template_context()
                start_from_trigger = ti.task.expand_start_from_trigger(context=context, session=session)

                if start_from_trigger:
                    ti.start_date = timezone.utcnow()
                    if ti.state != TaskInstanceState.UP_FOR_RESCHEDULE:
                        ti.try_number += 1
                    ti.defer_task(exception=None, session=session)
                else:
                    schedulable_ti_ids.append((ti.task_id, ti.map_index))
            else:
                schedulable_ti_ids.append((ti.task_id, ti.map_index))
```

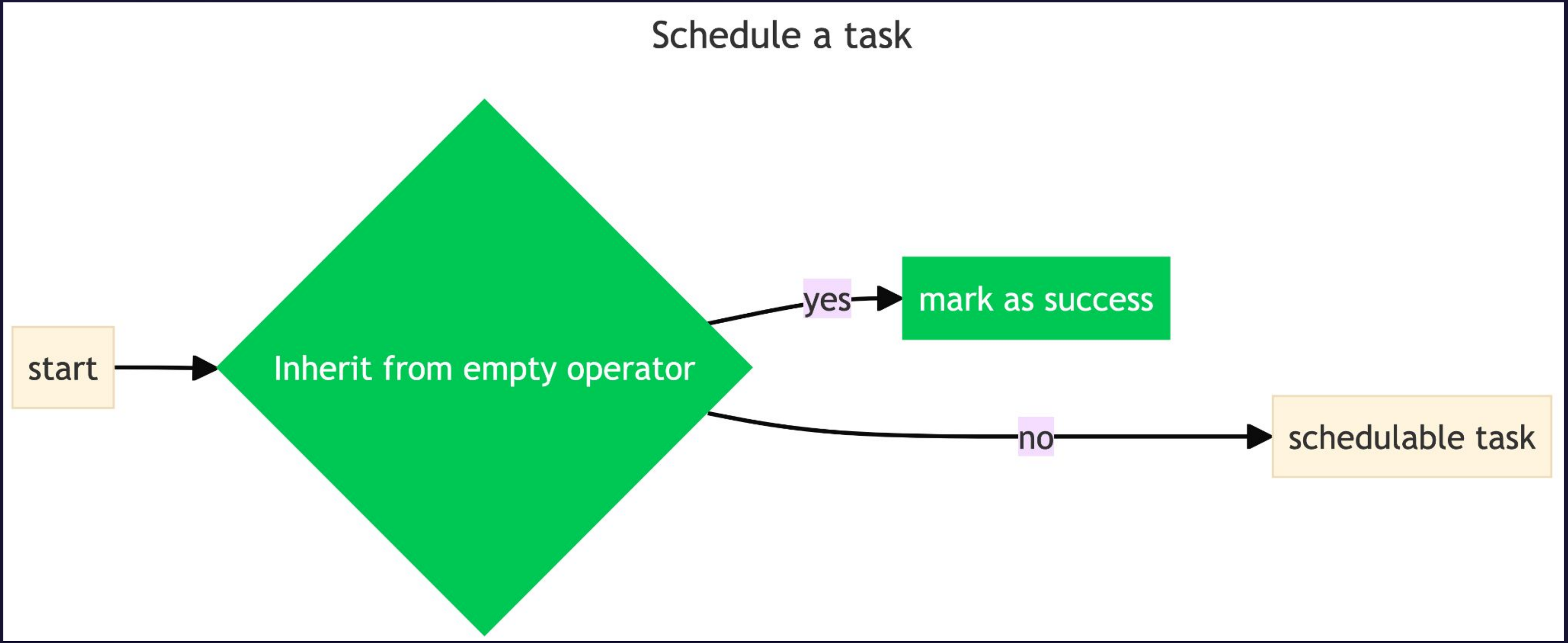
Start task execution in the triggerer

Under the hood (it used to be…)



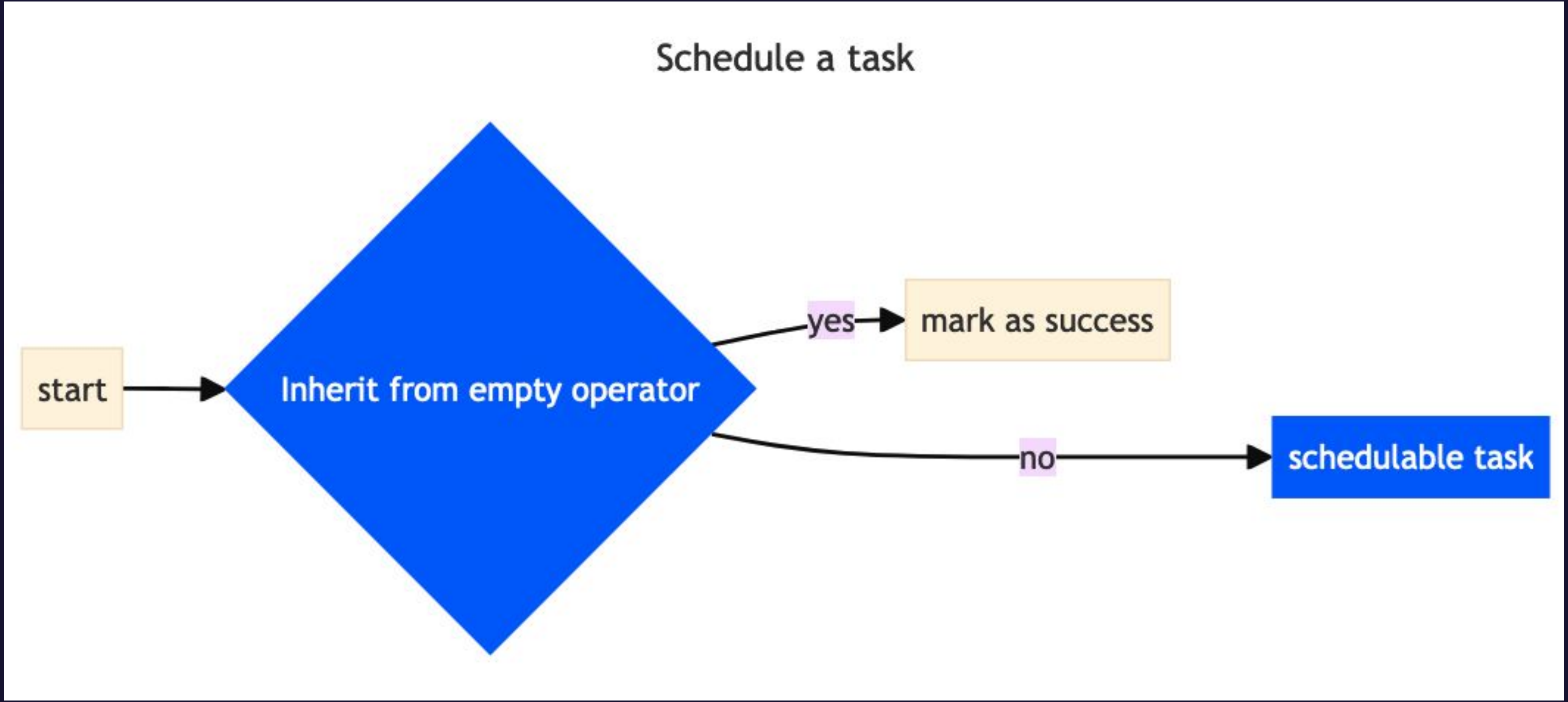
Start task execution in the triggerer

Under the hood (it used to be…)



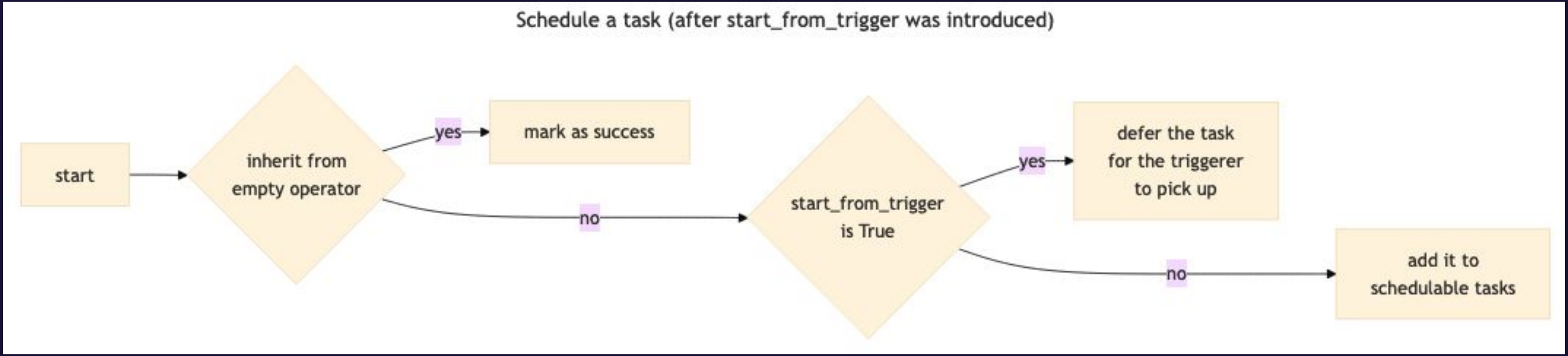
Start task execution in the triggerer

Under the hood (it used to be…)



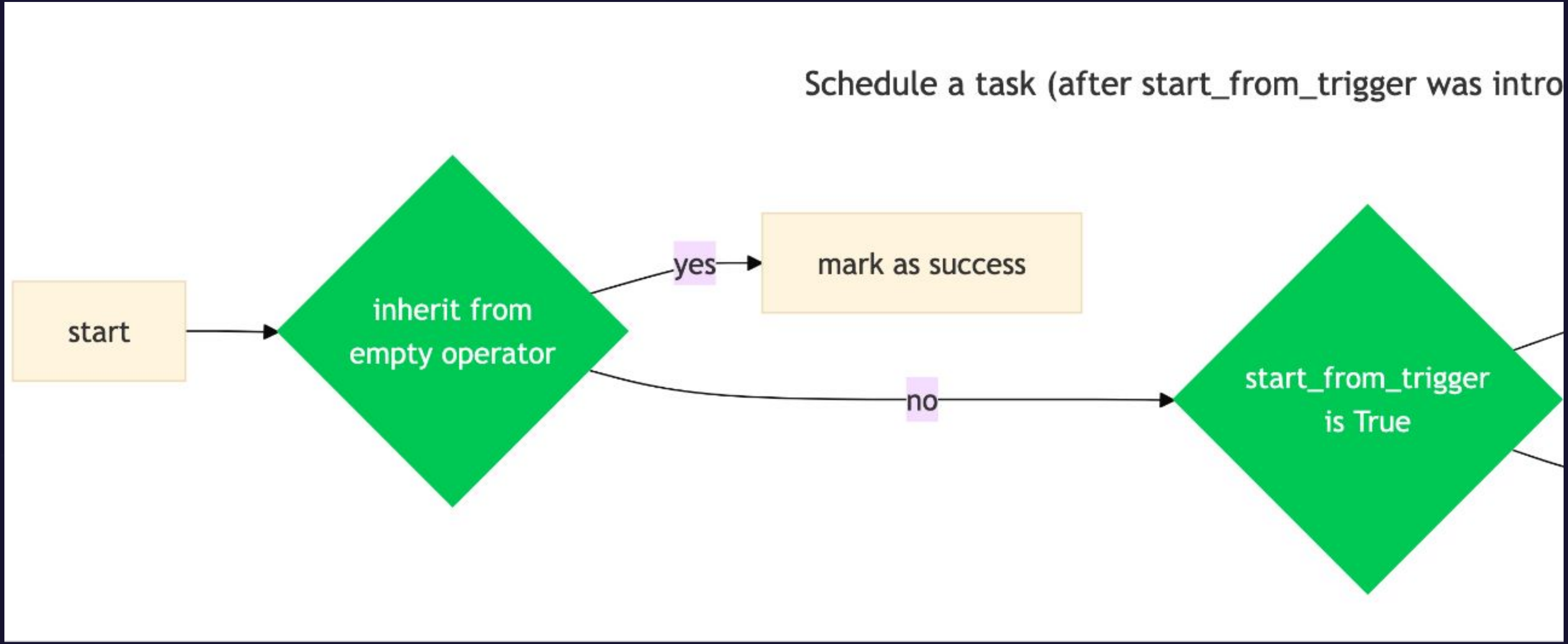
Start task execution in the triggerer

Under the hood (it’s now…)


Schedule a task (after start_from_trigger was introduced)
start
inherit from empty operator
yes
mark as success
no
start_from_trigger is True
yes
defer the task for the triggerer to pick up
no
add it to schedulable tasks

Start task execution in the triggerer

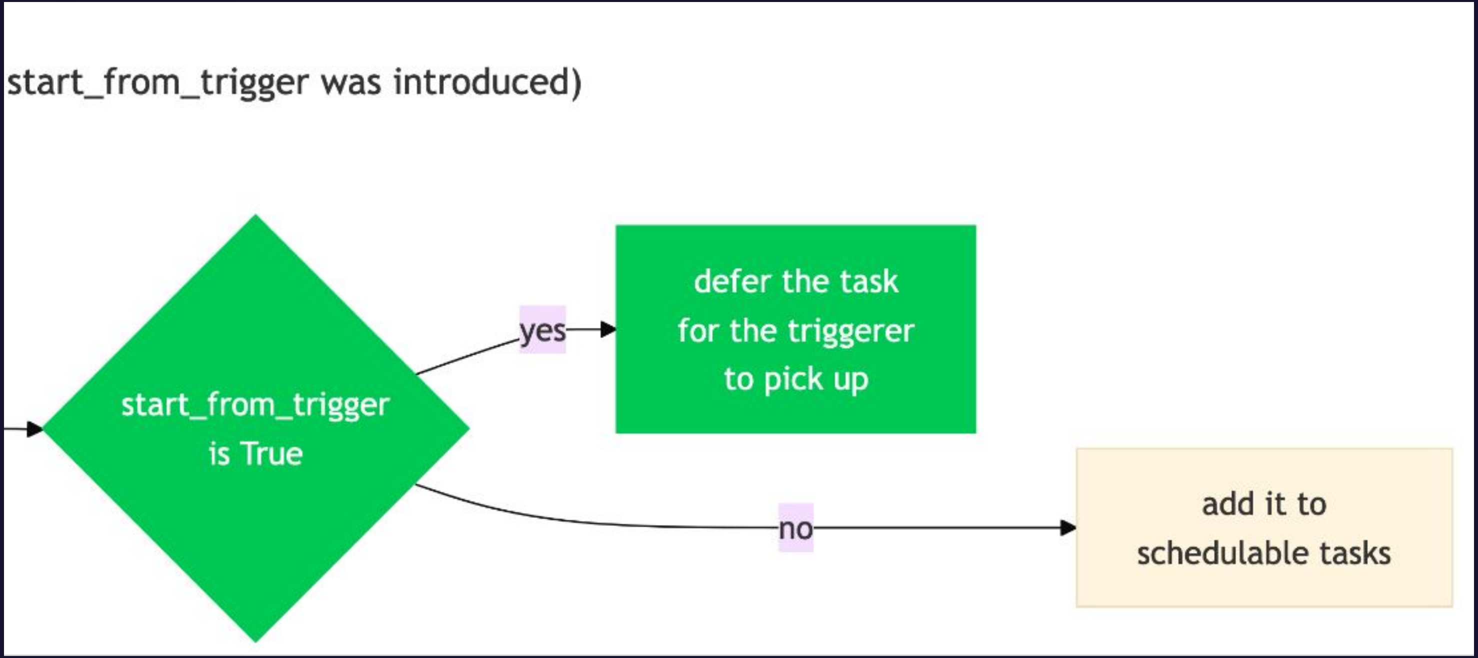
Under the hood (it's now…)



Schedule a task (after start_from_trigger was intro
start
inherit from empty operator
yes
mark as success
no
start_from_trigger is True

Start task execution in the triggerer

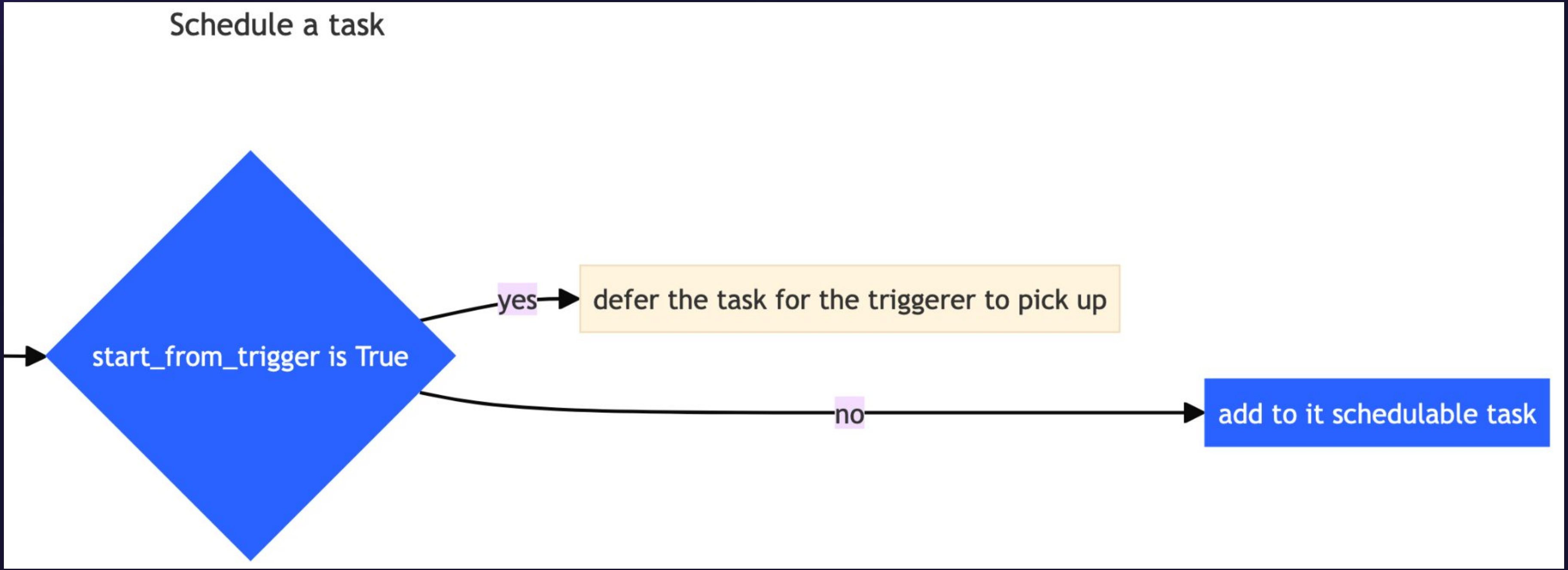
Under the hood (it's now…)


start_from_trigger was introduced)

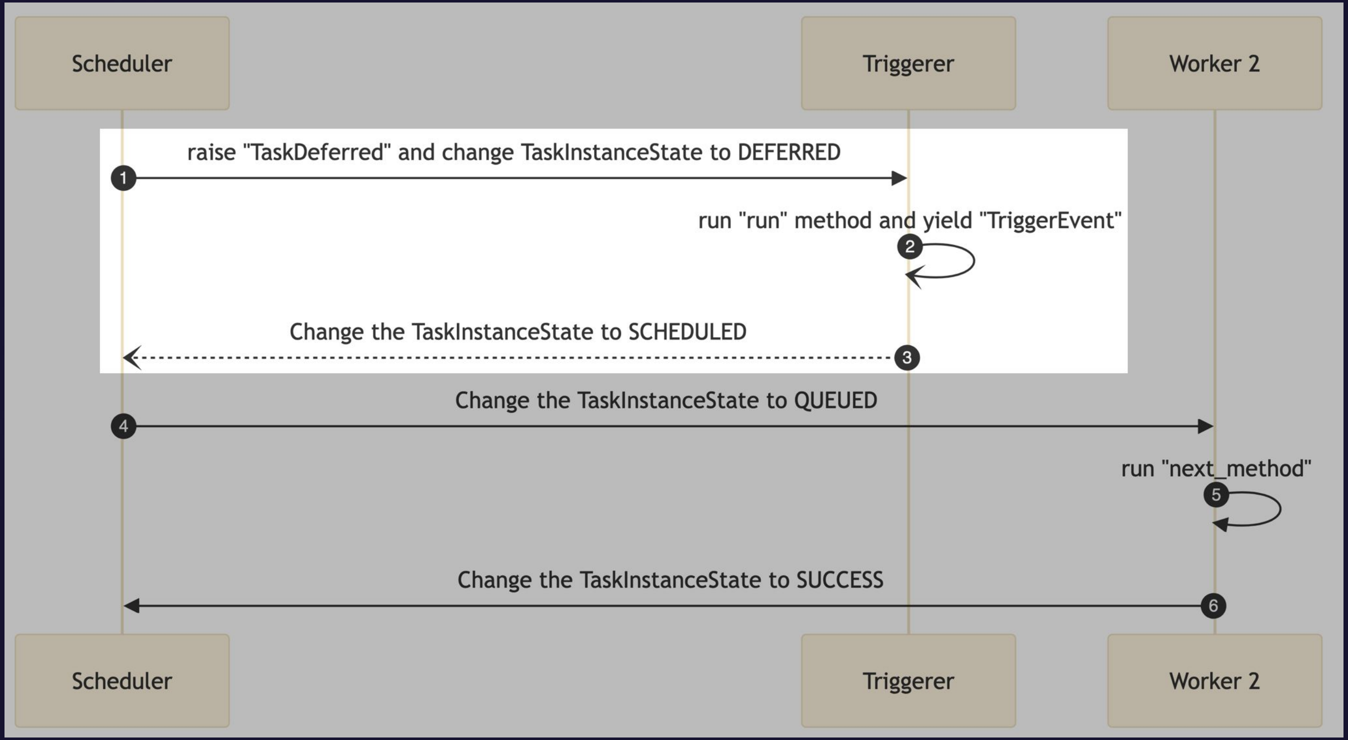
start_from_trigger is True
yes
defer the task for the triggerer to pick up
no
add it to schedulable tasks

Start task execution in the triggerer

Under the hood (it's now…)


Schedule a task
start_from_trigger is True
yes
defer the task for the triggerer to pick up
no
add to it schedulable task

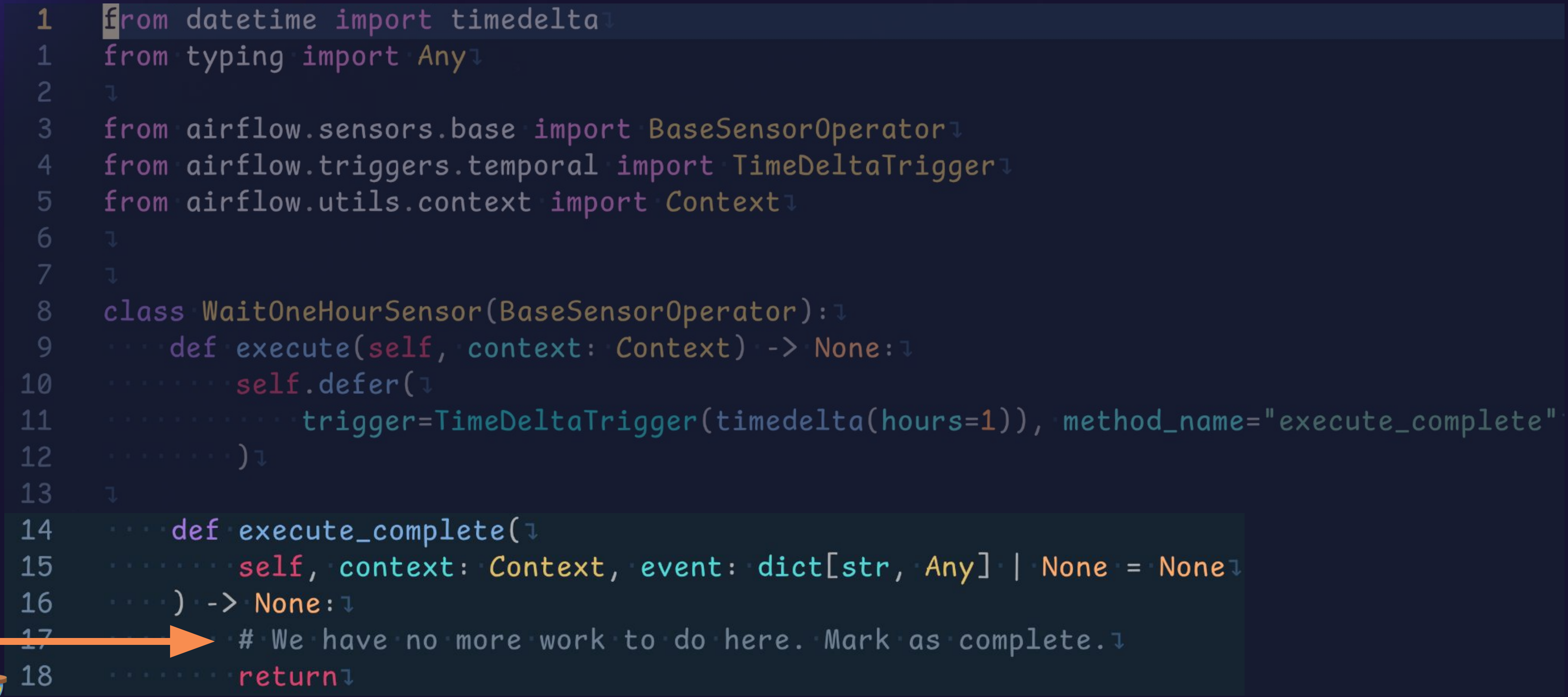
Start task execution in the triggerer


Scheduler
Triggerer
Worker 2
1 raise "TaskDeferred" and change TaskInstanceState to DEFERRED
2 run "run" method and yield "TriggerEvent"
3 Change the TaskInstanceState to SCHEDULED
4 Change the TaskInstanceState to QUEUED
5 run "next_method"
6 Change the TaskInstanceState to SUCCESS
Scheduler
Triggerer
Worker 2

WHAT IF...?

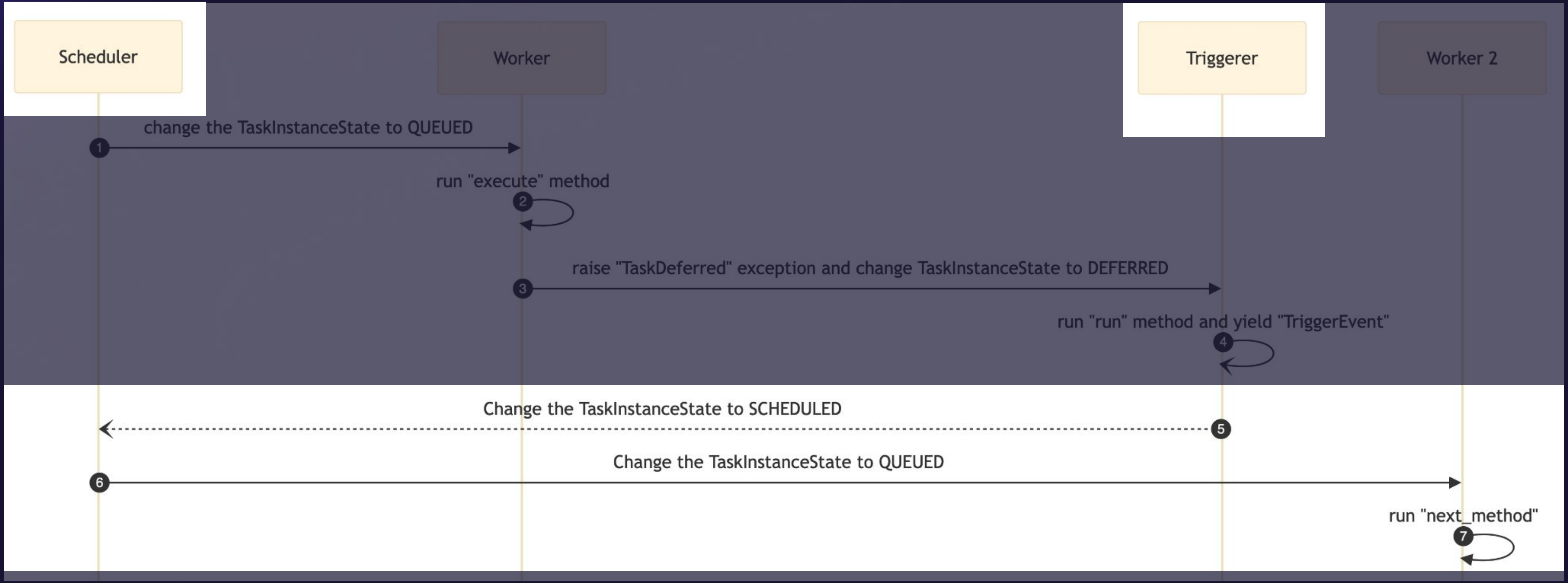
We don’t even have a next method to run after trigger finish its execution?

Especially when “next_method” does nothing



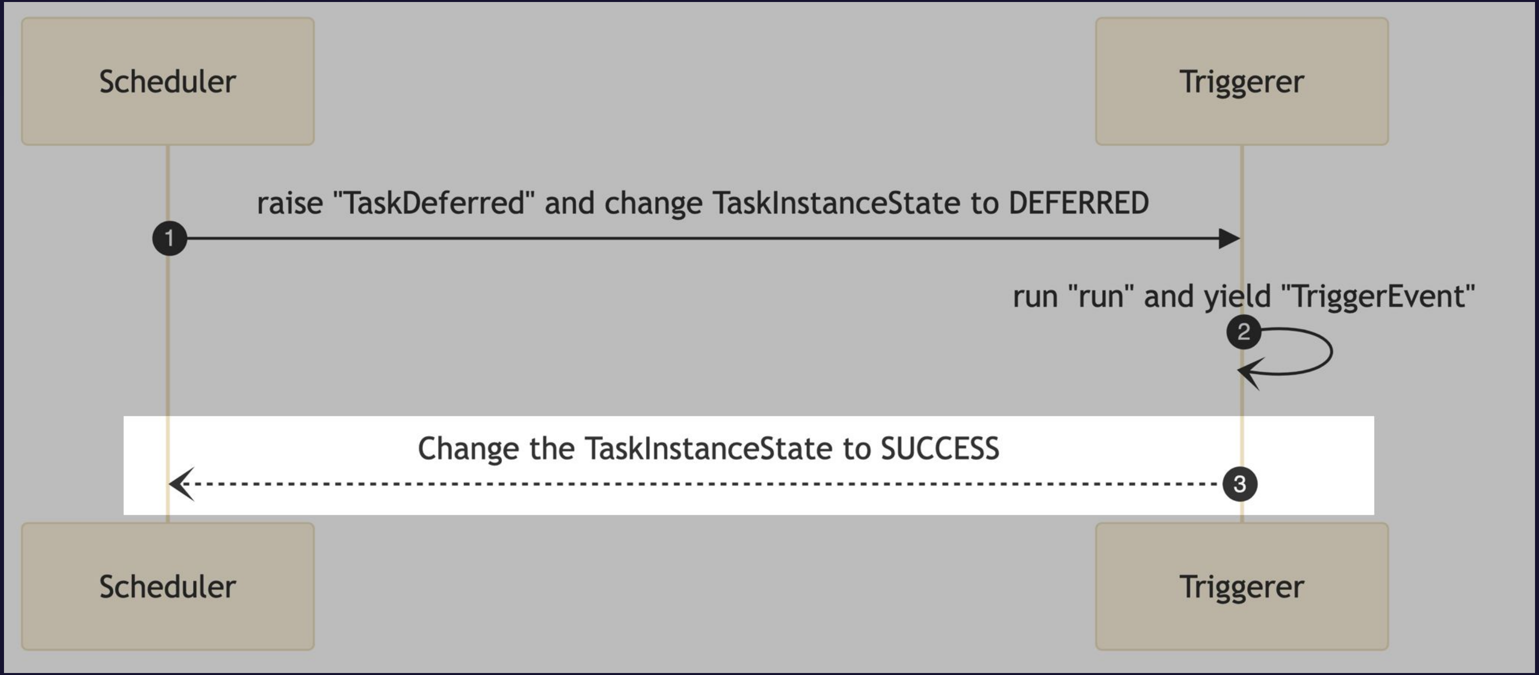
```python
from datetime import timedelta
from typing import Any

from airflow.sensors.base import BaseSensorOperator
from airflow.triggers.temporal import TimeDeltaTrigger
from airflow.utils.context import Context


class WaitOneHourSensor(BaseSensorOperator):
    def execute(self, context: Context) -> None:
        self.defer(
            trigger=TimeDeltaTrigger(timedelta(hours=1)), method_name="execute_complete"
        )

    def execute_complete(
        self, context: Context, event: dict[str, Any] | None = None
    ) -> None:
        # We have no more work to do here. Mark as complete.
        return
```

What can we do?

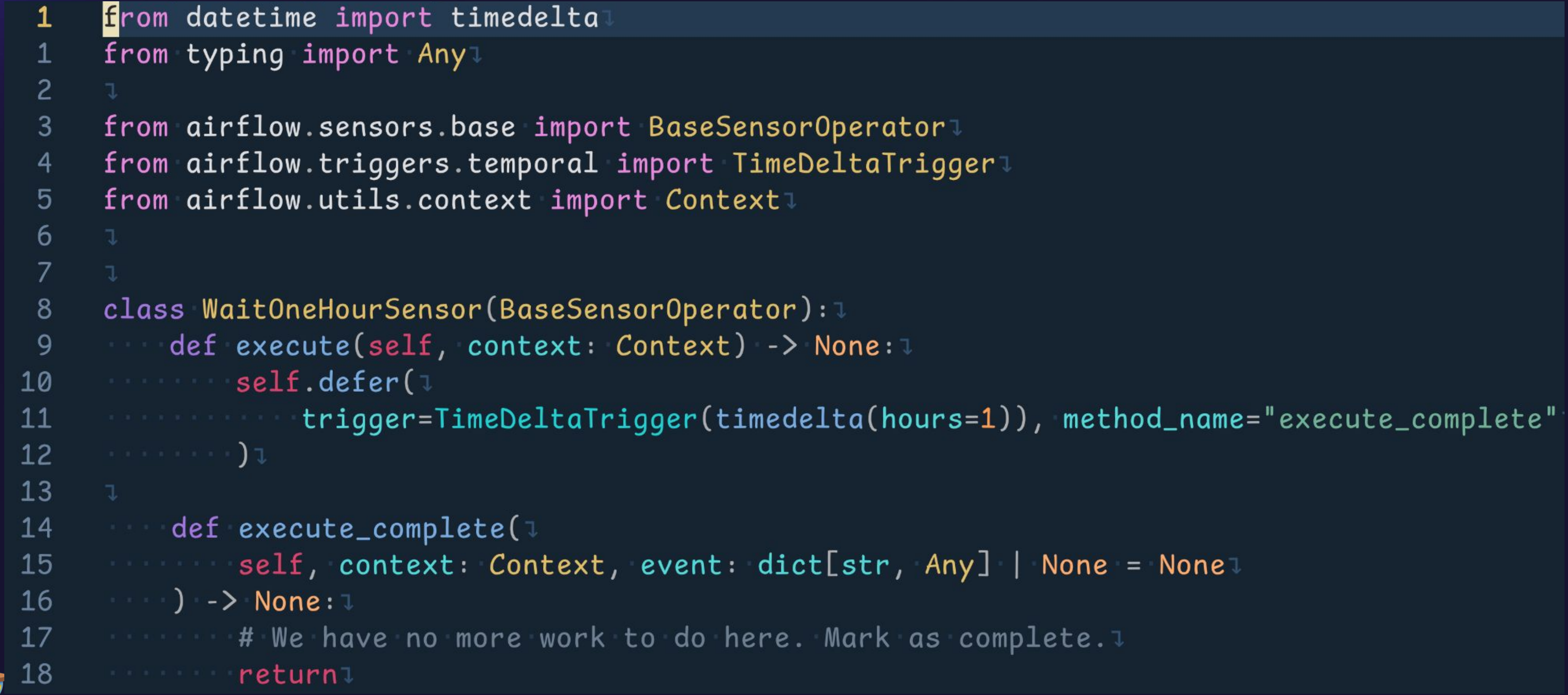

Scheduler
Worker
Triggerer
Worker 2
1 change the TaskInstanceState to QUEUED
2 run "execute" method
3 raise "TaskDeferred" exception and change TaskInstanceState to DEFERRED
4 run "run" method and yield "TriggerEvent"
5 Change the TaskInstanceState to SCHEDULED
6 Change the TaskInstanceState to QUEUED
7 run "next_method"

What’re the things we want to change?

End task execution in the triggerer


Scheduler
Triggerer
1
raise "TaskDeferred" and change TaskInstanceState to DEFERRED
run "run" and yield "TriggerEvent"
2
Change the TaskInstanceState to SUCCESS
3
Scheduler
Triggerer

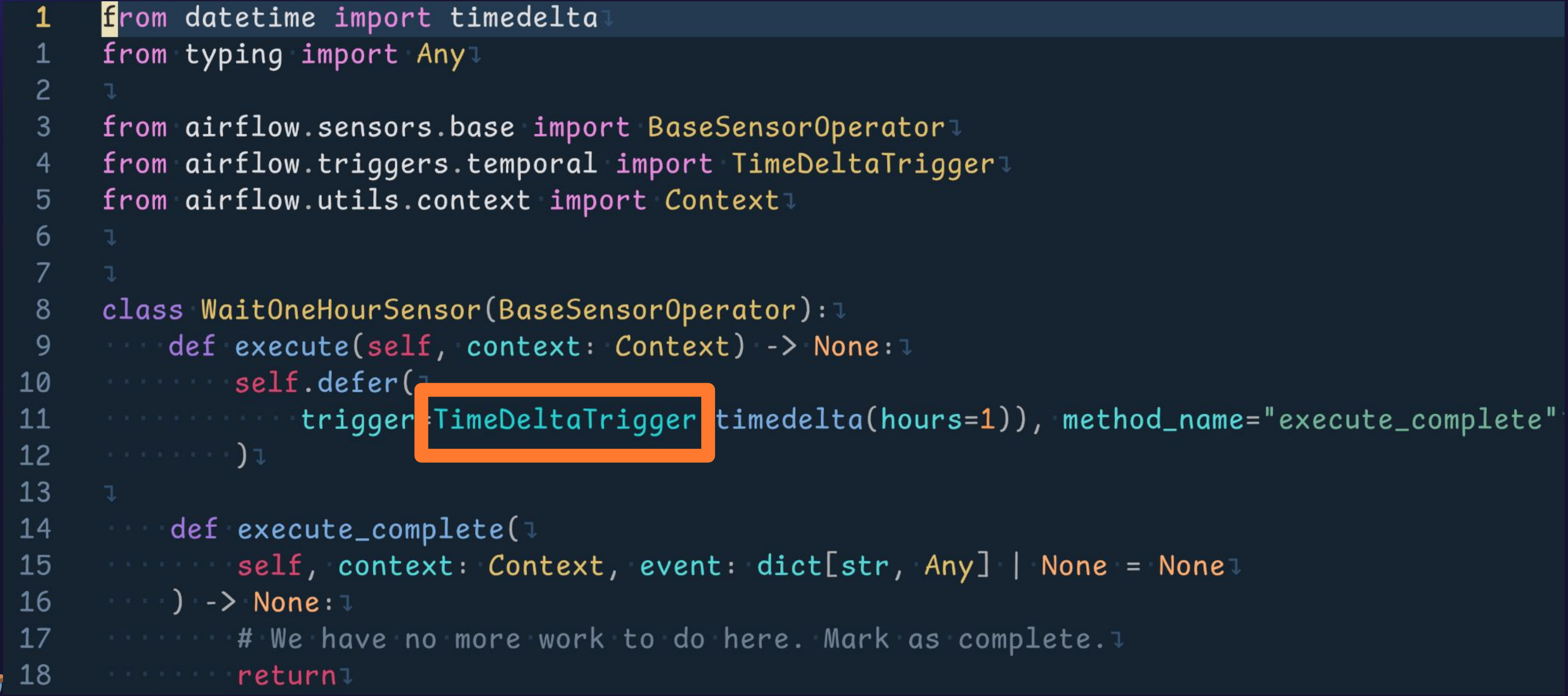
End task execution in the worker



```python
from datetime import timedelta
from typing import Any

from airflow.sensors.base import BaseSensorOperator
from airflow.triggers.temporal import TimeDeltaTrigger
from airflow.utils.context import Context


class WaitOneHourSensor(BaseSensorOperator):
    def execute(self, context: Context) -> None:
        self.defer(
            trigger=TimeDeltaTrigger(timedelta(hours=1)), method_name="execute_complete"
        )

    def execute_complete(
        self, context: Context, event: dict[str, Any] | None = None
    ) -> None:
        # We have no more work to do here. Mark as complete.
        return
```

ASTRONOMER

End task execution in the worker

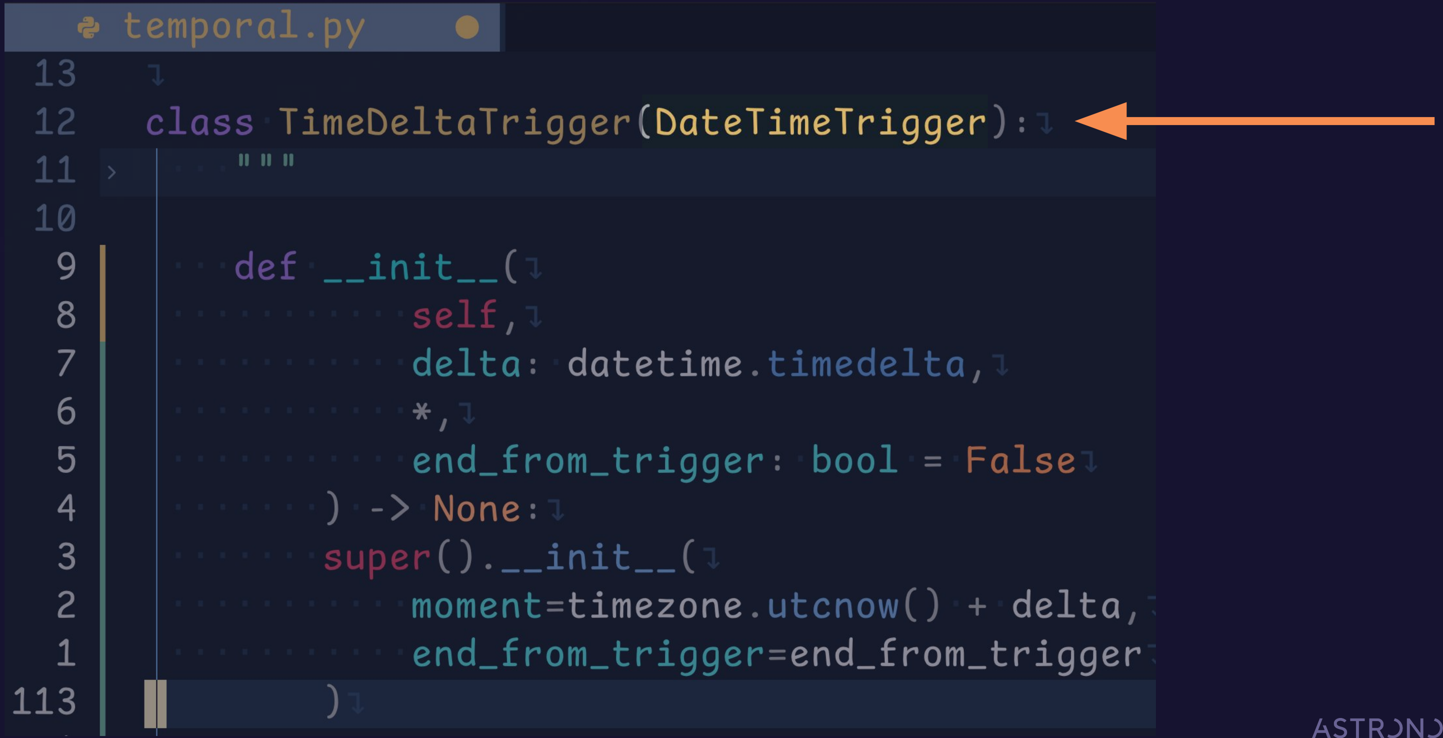


```python
from datetime import timedelta
from typing import Any

from airflow.sensors.base import BaseSensorOperator
from airflow.triggers.temporal import TimeDeltaTrigger
from airflow.utils.context import Context


class WaitOneHourSensor(BaseSensorOperator):
    def execute(self, context: Context) -> None:
        self.defer(
            trigger=TimeDeltaTrigger(timedelta(hours=1)), method_name="execute_complete"
        )

    def execute_complete(
        self, context: Context, event: dict[str, Any] | None = None
    ) -> None:
        # We have no more work to do here. Mark as complete.
        return
```

ASTRONOMER

End task execution in the triggerer

Well… we need to make some change in the trigger this time

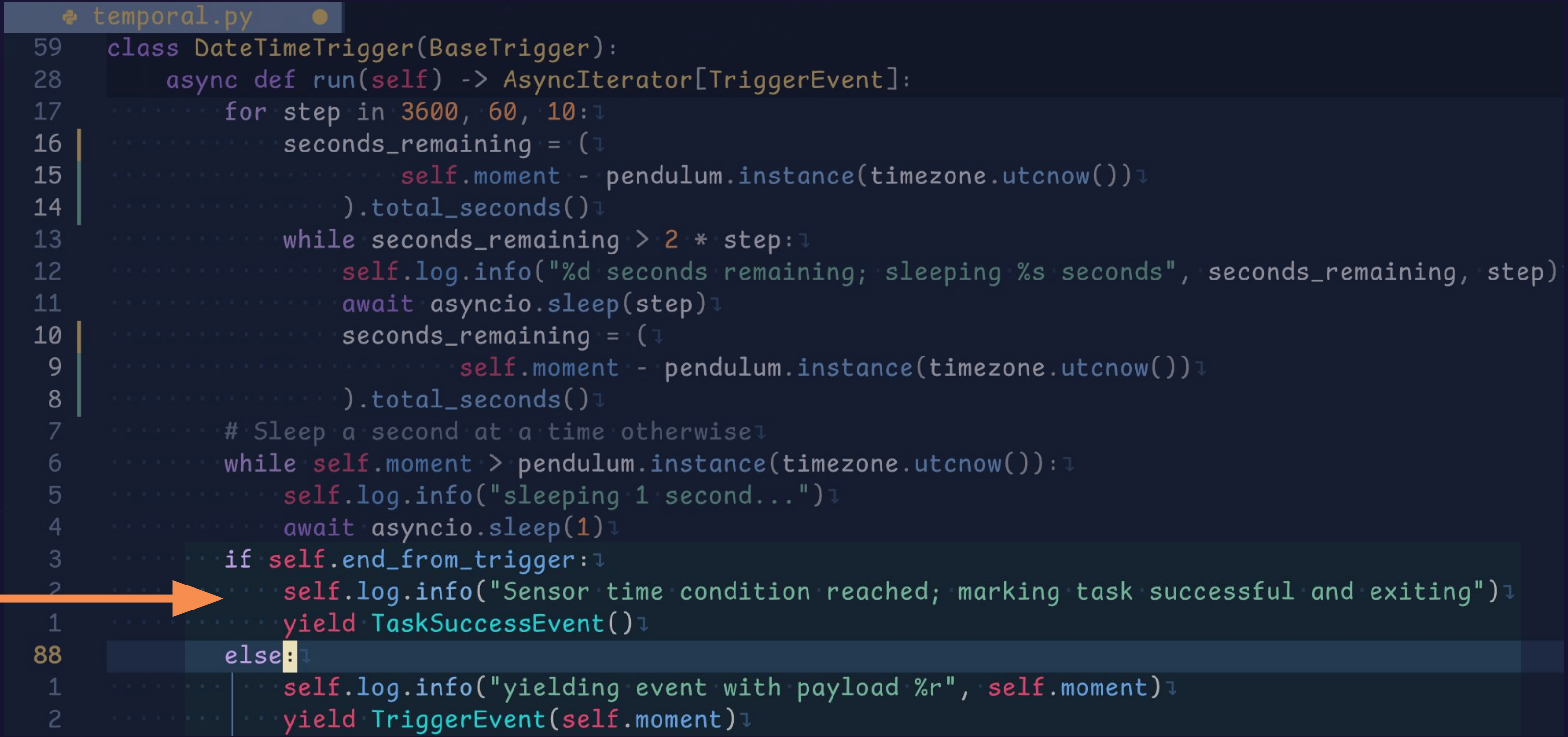


temporal.py

```
class TimeDeltaTrigger(DateTimeTrigger):
    """

    def __init__(
            self,
            delta: datetime.timedelta,
            *,
            end_from_trigger: bool = False
        ) -> None:
        super().__init__(
            moment=timezone.utcnow() + delta,
            end_from_trigger=end_from_trigger
        )
```

End task execution in the triggerer

Well… we need to make some change in the trigger



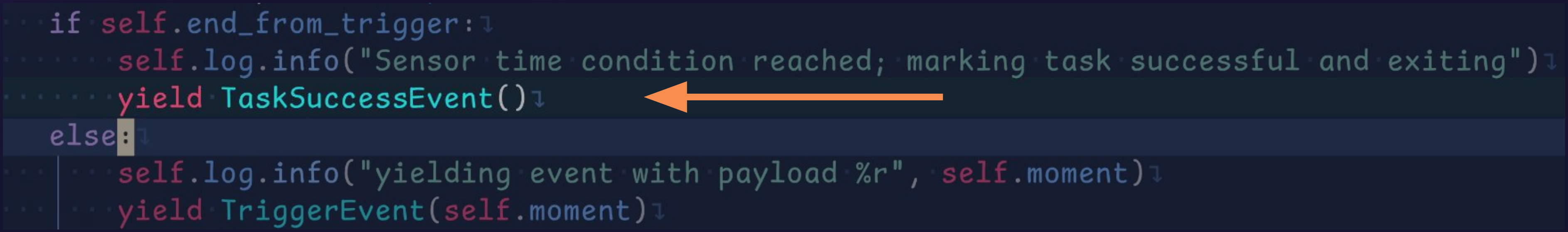
temporal.py

```python
class DateTimeTrigger(BaseTrigger):
    async def run(self) -> AsyncIterator[TriggerEvent]:
        for step in 3600, 60, 10:
            seconds_remaining = (
                    self.moment - pendulum.instance(timezone.utcnow())
                ).total_seconds()
            while seconds_remaining > 2 * step:
                self.log.info("%d seconds remaining; sleeping %s seconds", seconds_remaining, step)
                await asyncio.sleep(step)
                seconds_remaining = (
                        self.moment - pendulum.instance(timezone.utcnow())
                ).total_seconds()
        # Sleep a second at a time otherwise
        while self.moment > pendulum.instance(timezone.utcnow()):
            self.log.info("sleeping 1 second...")
            await asyncio.sleep(1)
        if self.end_from_trigger:
            self.log.info("Sensor time condition reached; marking task successful and exiting")
            yield TaskSuccessEvent()
        else:
            self.log.info("yielding event with payload %r", self.moment)
            yield TriggerEvent(self.moment)
```

ASTRONOMER

End task execution in the triggerer

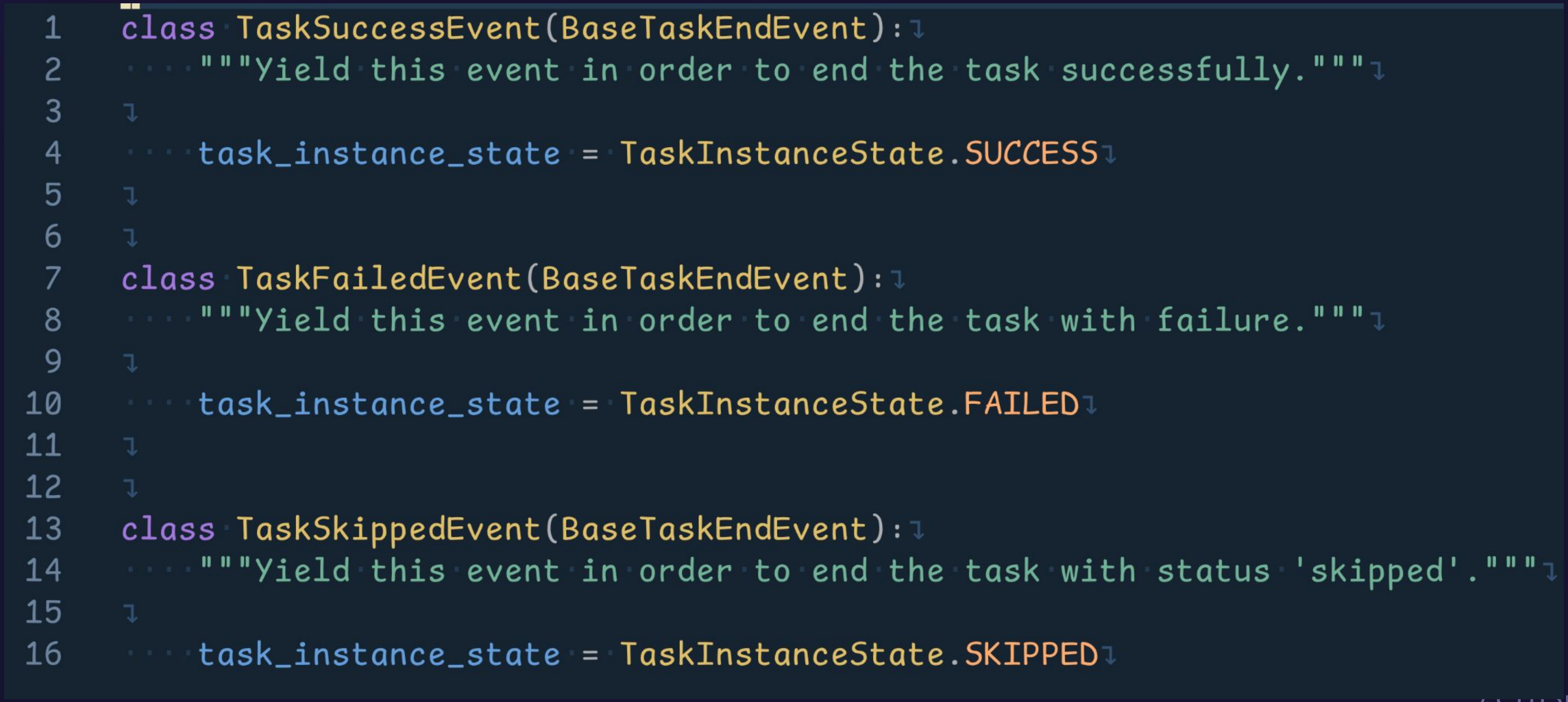
Yield a TaskSuccessEvent



```
if self.end_from_trigger:
    self.log.info("Sensor time condition reached; marking task successful and exiting")
    yield TaskSuccessEvent()
else:
    self.log.info("yielding event with payload %r", self.moment)
    yield TriggerEvent(self.moment)
```

End task execution in the triggerer

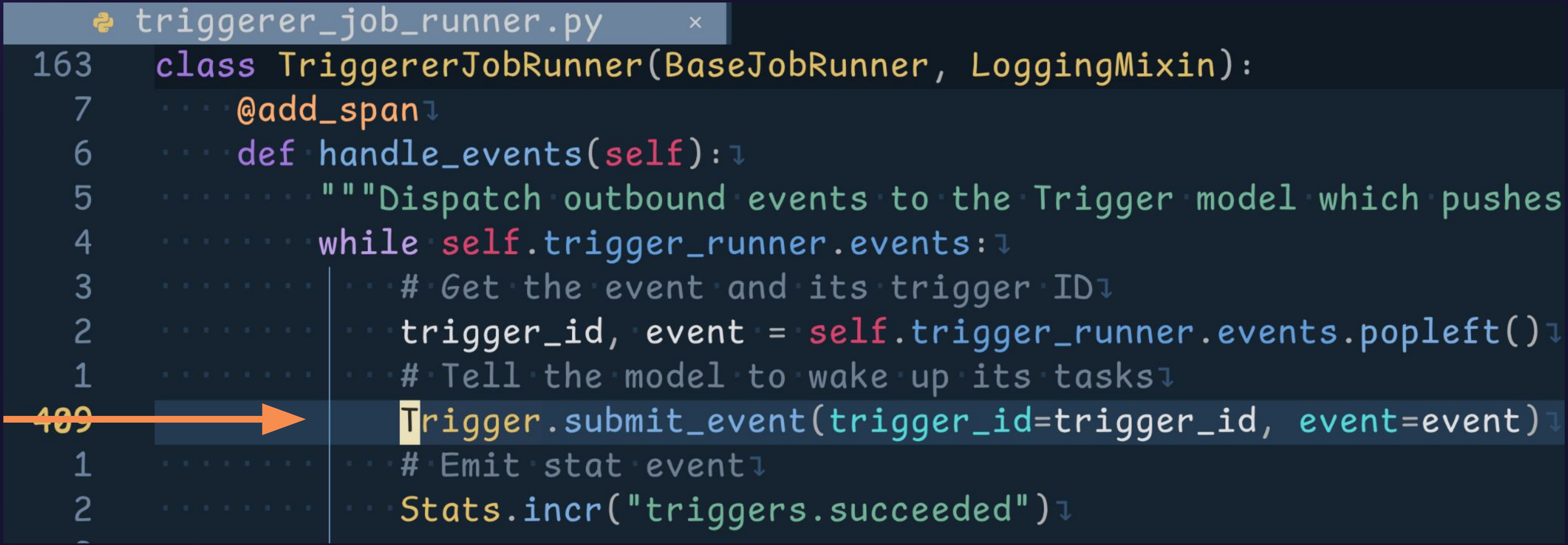
Newly supported TriggerEvents



```python
class TaskSuccessEvent(BaseTaskEndEvent):
    """Yield this event in order to end the task successfully."""

    task_instance_state = TaskInstanceState.SUCCESS


class TaskFailedEvent(BaseTaskEndEvent):
    """Yield this event in order to end the task with failure."""

    task_instance_state = TaskInstanceState.FAILED


class TaskSkippedEvent(BaseTaskEndEvent):
    """Yield this event in order to end the task with status 'skipped'."""

    task_instance_state = TaskInstanceState.SKIPPED
```

End task execution in the triggerer

Under the hook



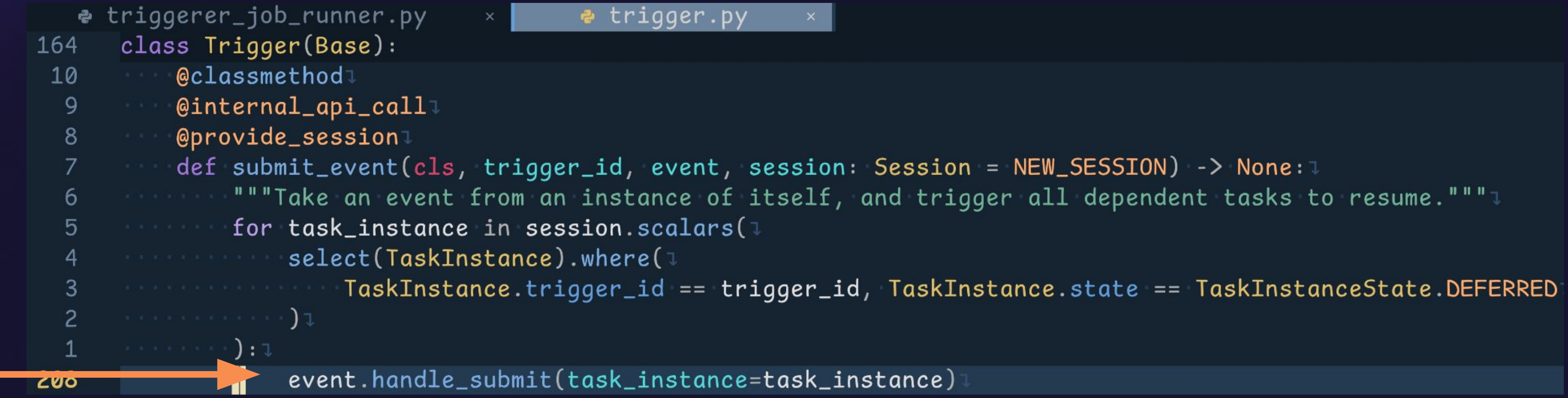
triggerer_job_runner.py

```
class TriggererJobRunner(BaseJobRunner, LoggingMixin):
    @add_span
    def handle_events(self):
        """Dispatch outbound events to the Trigger model which pushes
        while self.trigger_runner.events:
            # Get the event and its trigger ID
            trigger_id, event = self.trigger_runner.events.popleft()
            # Tell the model to wake up its tasks
            Trigger.submit_event(trigger_id=trigger_id, event=event)
            # Emit stat event
            Stats.incr("triggers.succeeded")
```

ASTRONOMER

End task execution in the triggerer

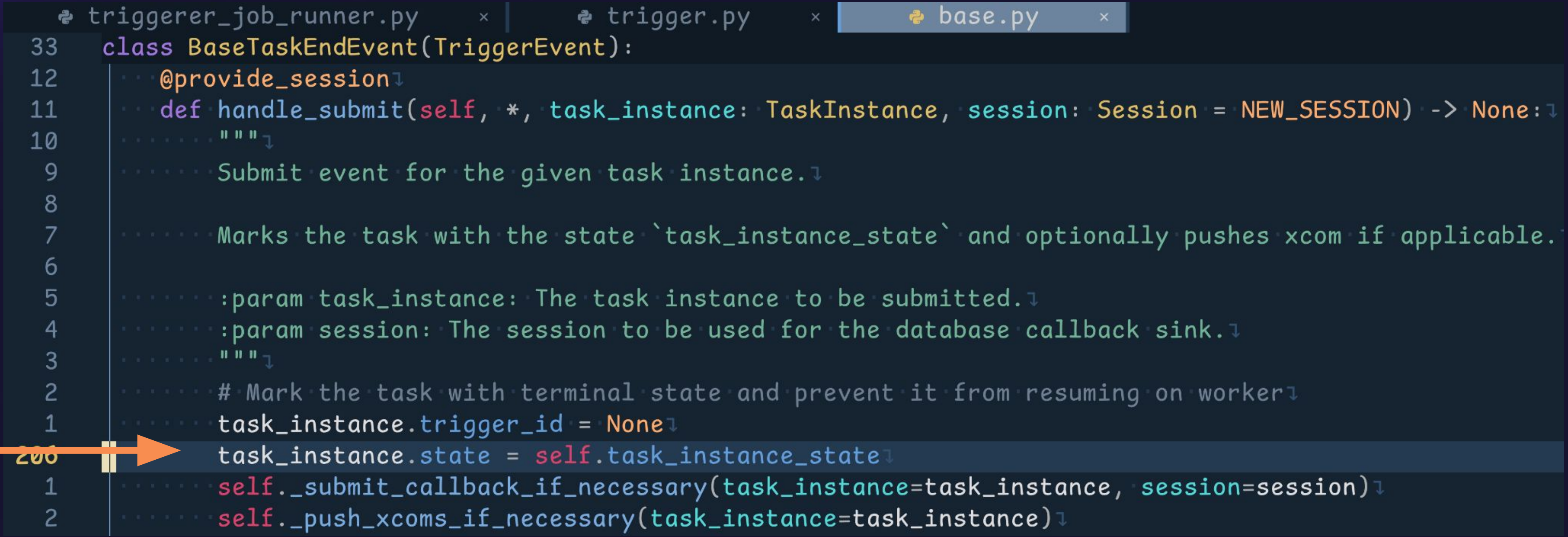
Under the hook



```python
class Trigger(Base):
    @classmethod
    @internal_api_call
    @provide_session
    def submit_event(cls, trigger_id, event, session: Session = NEW_SESSION) -> None:
        """Take an event from an instance of itself, and trigger all dependent tasks to resume."""
        for task_instance in session.scalars(
            select(TaskInstance).where(
                TaskInstance.trigger_id == trigger_id, TaskInstance.state == TaskInstanceState.DEFERRED
            )
        ):
            event.handle_submit(task_instance=task_instance)
```

End task execution in the triggerer

Under the hook, it updates the state...



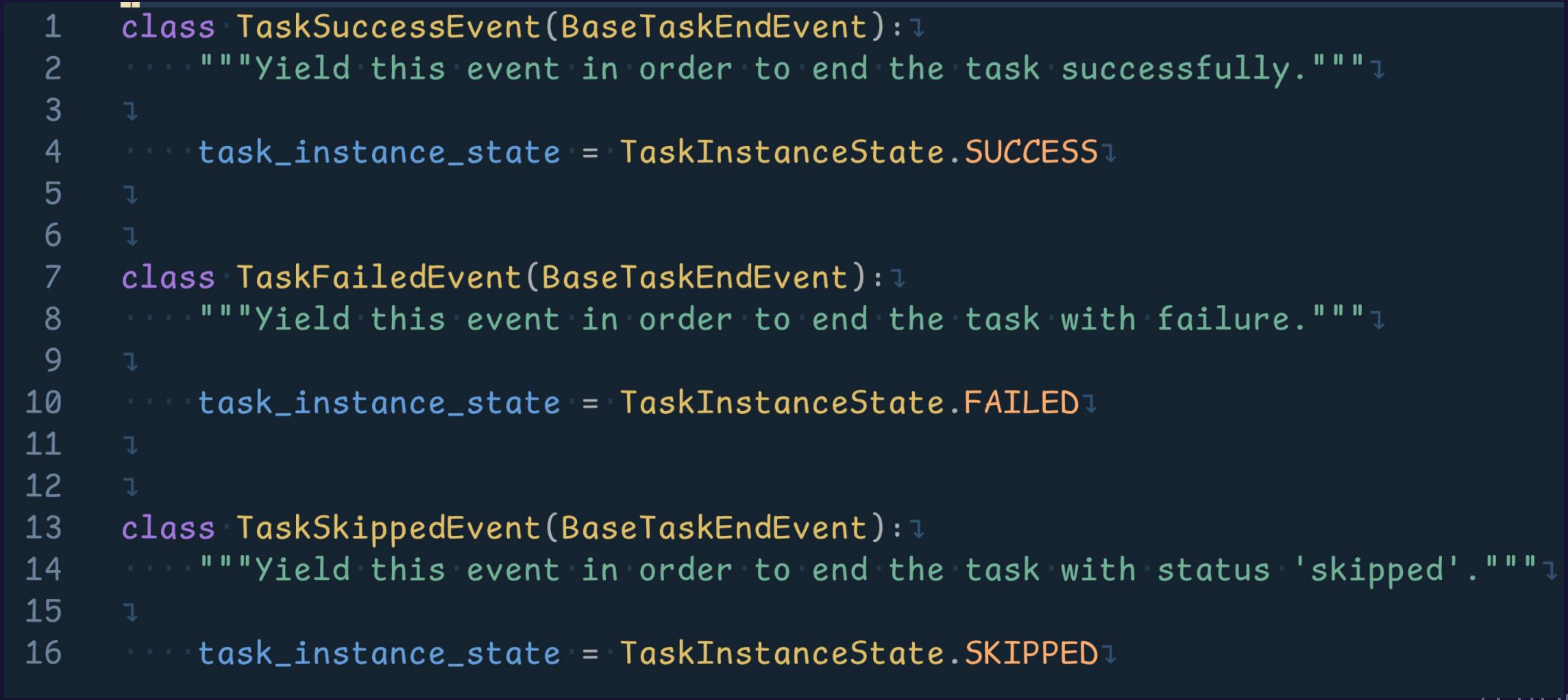
triggerer_job_runner.py × | trigger.py × | base.py ×

```python
class BaseTaskEndEvent(TriggerEvent):
    @provide_session
    def handle_submit(self, *, task_instance: TaskInstance, session: Session = NEW_SESSION) -> None:
        """
        Submit event for the given task instance.

        Marks the task with the state `task_instance_state` and optionally pushes xcom if applicable.

        :param task_instance: The task instance to be submitted.
        :param session: The session to be used for the database callback sink.
        """
        # Mark the task with terminal state and prevent it from resuming on worker
        task_instance.trigger_id = None
        task_instance.state = self.task_instance_state
        self._submit_callback_if_necessary(task_instance=task_instance, session=session)
        self._push_xcoms_if_necessary(task_instance=task_instance)
```

ASTRONOMER

End task execution in the triggerer

based on the TriggerEvent type

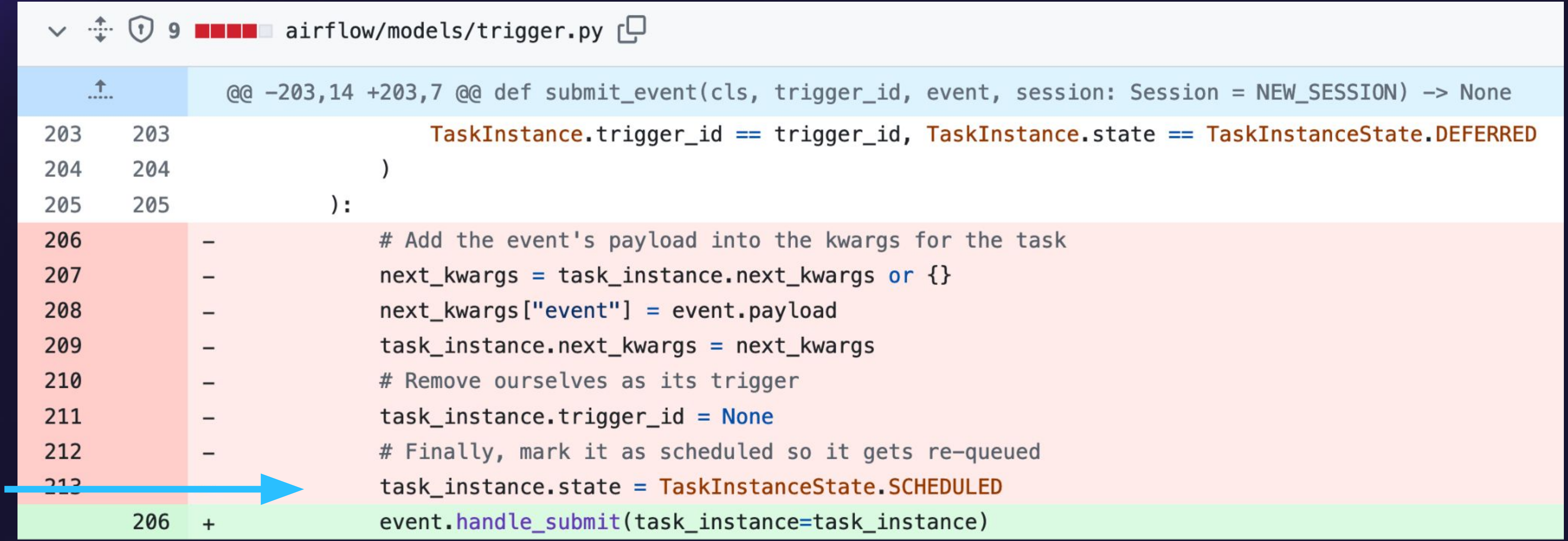


```python
class TaskSuccessEvent(BaseTaskEndEvent):
    """Yield this event in order to end the task successfully."""

    task_instance_state = TaskInstanceState.SUCCESS


class TaskFailedEvent(BaseTaskEndEvent):
    """Yield this event in order to end the task with failure."""

    task_instance_state = TaskInstanceState.FAILED


class TaskSkippedEvent(BaseTaskEndEvent):
    """Yield this event in order to end the task with status 'skipped'."""

    task_instance_state = TaskInstanceState.SKIPPED
```

ASTRONOMER

End task execution in the triggerer

which used to be always set as SCHEDULED

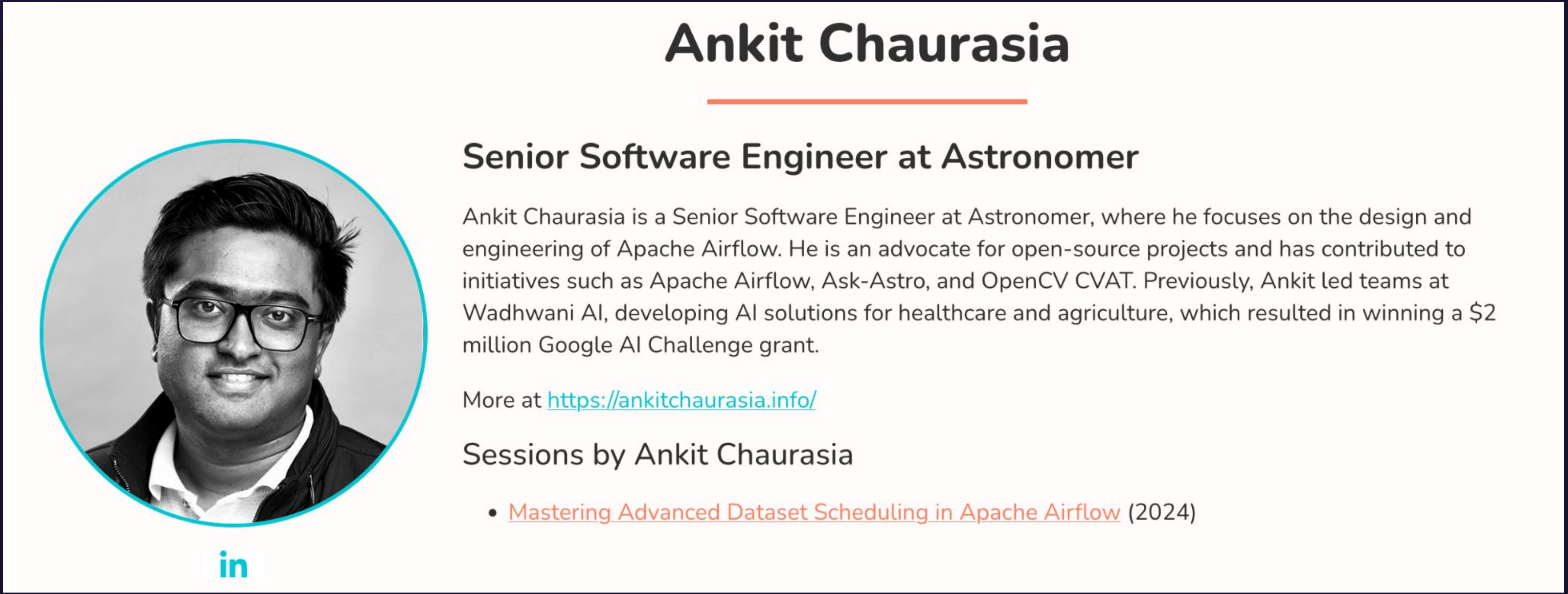


```
9 airflow/models/trigger.py

@@ -203,14 +203,7 @@ def submit_event(cls, trigger_id, event, session: Session = NEW_SESSION) -> None
                    TaskInstance.trigger_id == trigger_id, TaskInstance.state == TaskInstanceState.DEFERRED
                )
            ):
-               # Add the event's payload into the kwargs for the task
-               next_kwargs = task_instance.next_kwargs or {}
-               next_kwargs["event"] = event.payload
-               task_instance.next_kwargs = next_kwargs
-               # Remove ourselves as its trigger
-               task_instance.trigger_id = None
-               # Finally, mark it as scheduled so it gets re-queued
-               task_instance.state = TaskInstanceState.SCHEDULED
+               event.handle_submit(task_instance=task_instance)
```

ASTRONOMER

Credit

author of the end from trigger feature



Ankit Chaurasia

Senior Software Engineer at Astronomer

Ankit Chaurasia is a Senior Software Engineer at Astronomer, where he focuses on the design and engineering of Apache Airflow. He is an advocate for open-source projects and has contributed to initiatives such as Apache Airflow, Ask-Astro, and OpenCV CVAT. Previously, Ankit led teams at Wadhwani AI, developing AI solutions for healthcare and agriculture, which resulted in winning a $2 million Google AI Challenge grant.

More at https://ankitchaurasia.info/

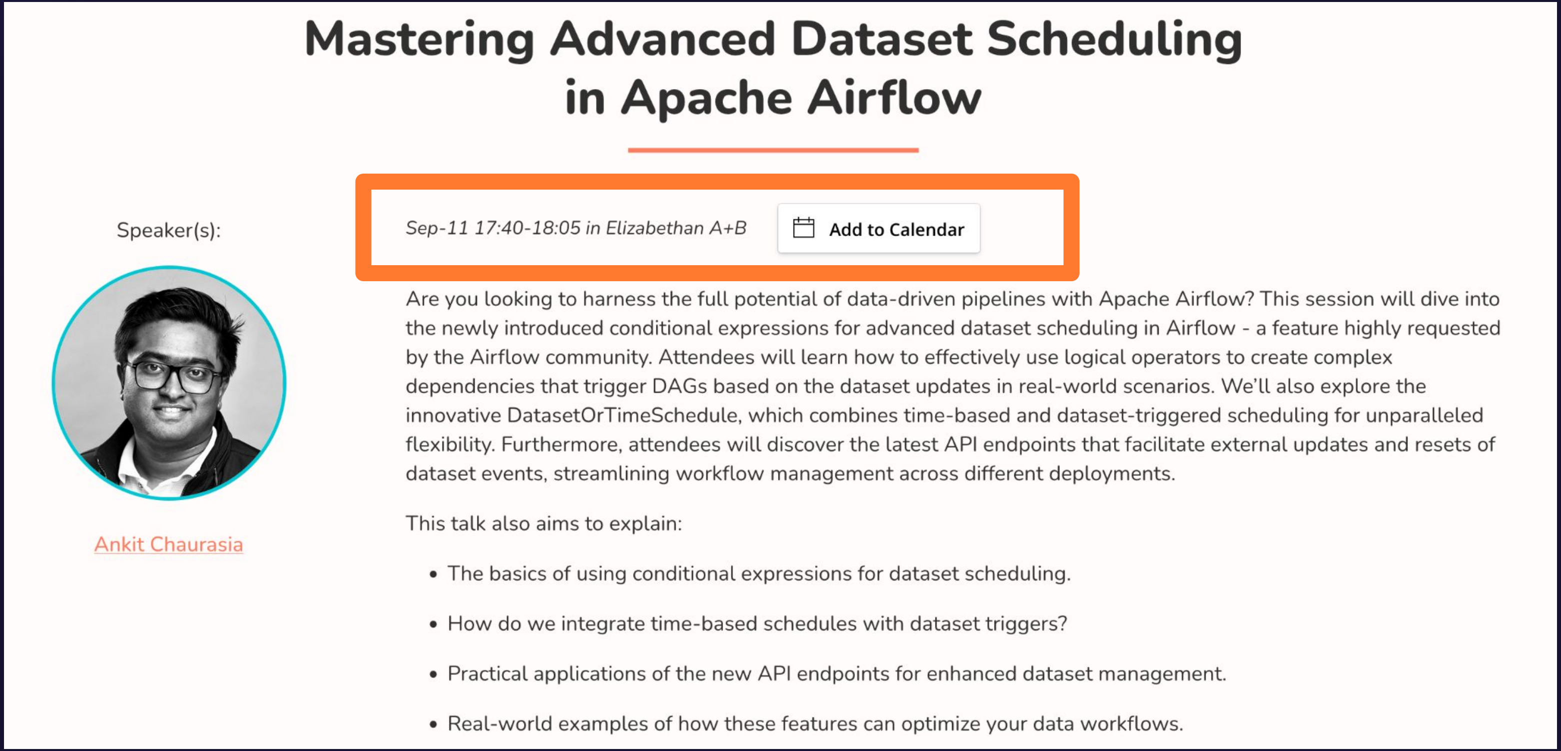
Sessions by Ankit Chaurasia

- Mastering Advanced Dataset Scheduling in Apache Airflow (2024)

in

ASTRONOMER

17:40 at the same room



Mastering Advanced Dataset Scheduling in Apache Airflow

Speaker(s):

Ankit Chaurasia

Sep-11 17:40-18:05 in Elizabethan A+B

Are you looking to harness the full potential of data-driven pipelines with Apache Airflow? This session will dive into the newly introduced conditional expressions for advanced dataset scheduling in Airflow - a feature highly requested by the Airflow community. Attendees will learn how to effectively use logical operators to create complex dependencies that trigger DAGs based on the dataset updates in real-world scenarios. We'll also explore the innovative DatasetOrTimeSchedule, which combines time-based and dataset-triggered scheduling for unparalleled flexibility. Furthermore, attendees will discover the latest API endpoints that facilitate external updates and resets of dataset events, streamlining workflow management across different deployments.

This talk also aims to explain:

- The basics of using conditional expressions for dataset scheduling.
- How do we integrate time-based schedules with dataset triggers?
- Practical applications of the new API endpoints for enhanced dataset management.
- Real-world examples of how these features can optimize your data workflows.

How does it affect DAG authors?

- Release more worker slot
- Improve operators and sensors for efficiency
- Reduce resource usage, which indicates cost saving
- More new use cases to come after more operator authors apply this new feature

How does it affect operator authors?

- A new way to implement operators in an asynchronous manner
- Simplify operators / sensors by reducing unnecessary “execute” and “execute_complete” methods
(most applicable to sensors I think)

How does it affect Airflow?

Potential to run all tasks in async

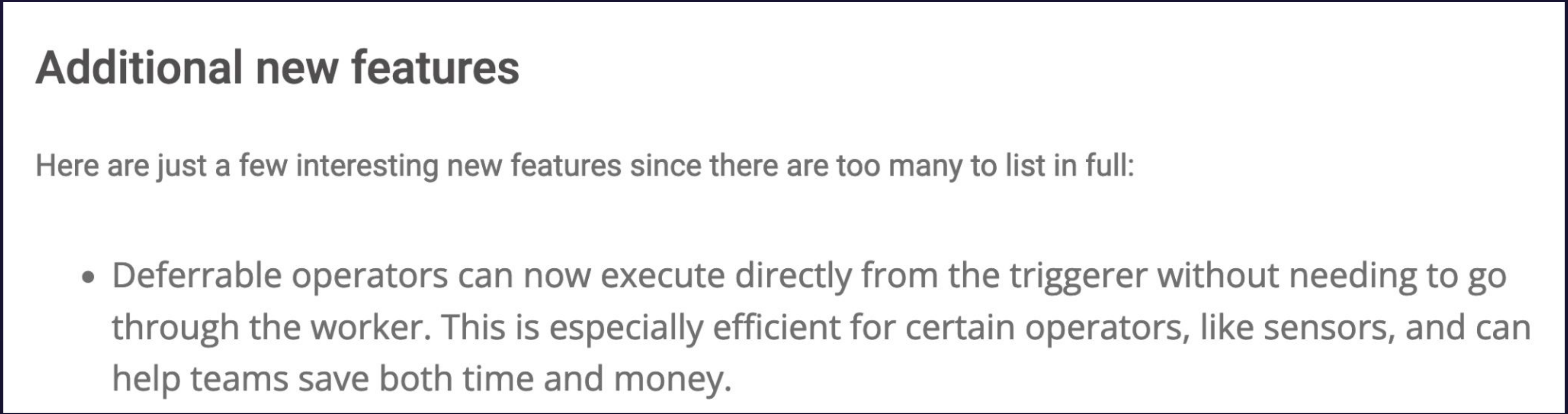
How does it affect Airflow?

Potential to run all tasks in async ?

ASTRONOMER

They're included in Airflow 2.10.0 🎉

You can find it if you scroll down to the end of Airflow 2.10.0 post



Additional new features

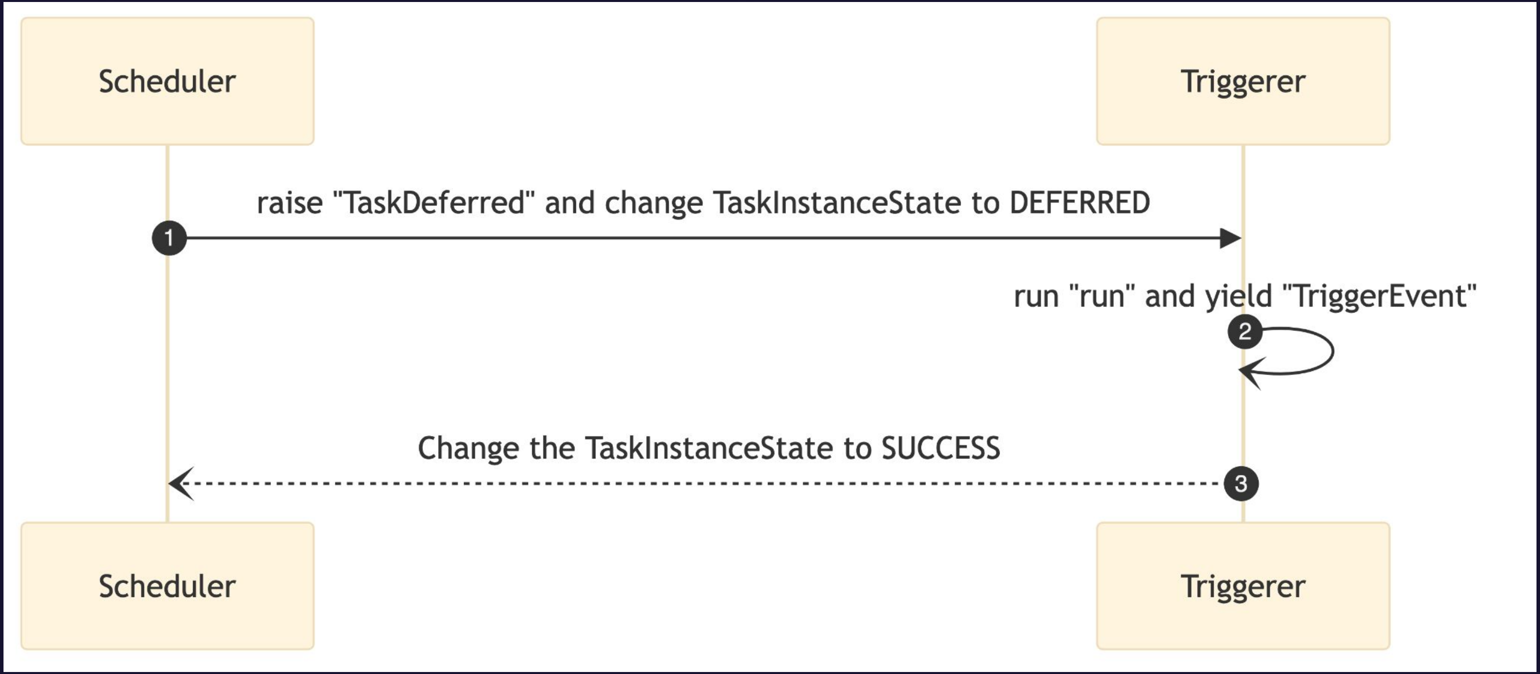
Here are just a few interesting new features since there are too many to list in full:

- Deferrable operators can now execute directly from the triggerer without needing to go through the worker. This is especially efficient for certain operators, like sensors, and can help teams save both time and money.

Airflow 2.10.0 post: https://airflow.apache.org/blog/airflow-2.10.0/

Running Airflow Tasks without the workers

There's no more "What if…"


Scheduler
Triggerer
raise "TaskDeferred" and change TaskInstanceState to DEFERRED
1
run "run" and yield "TriggerEvent"
2
Change the TaskInstanceState to SUCCESS
3
Scheduler
Triggerer

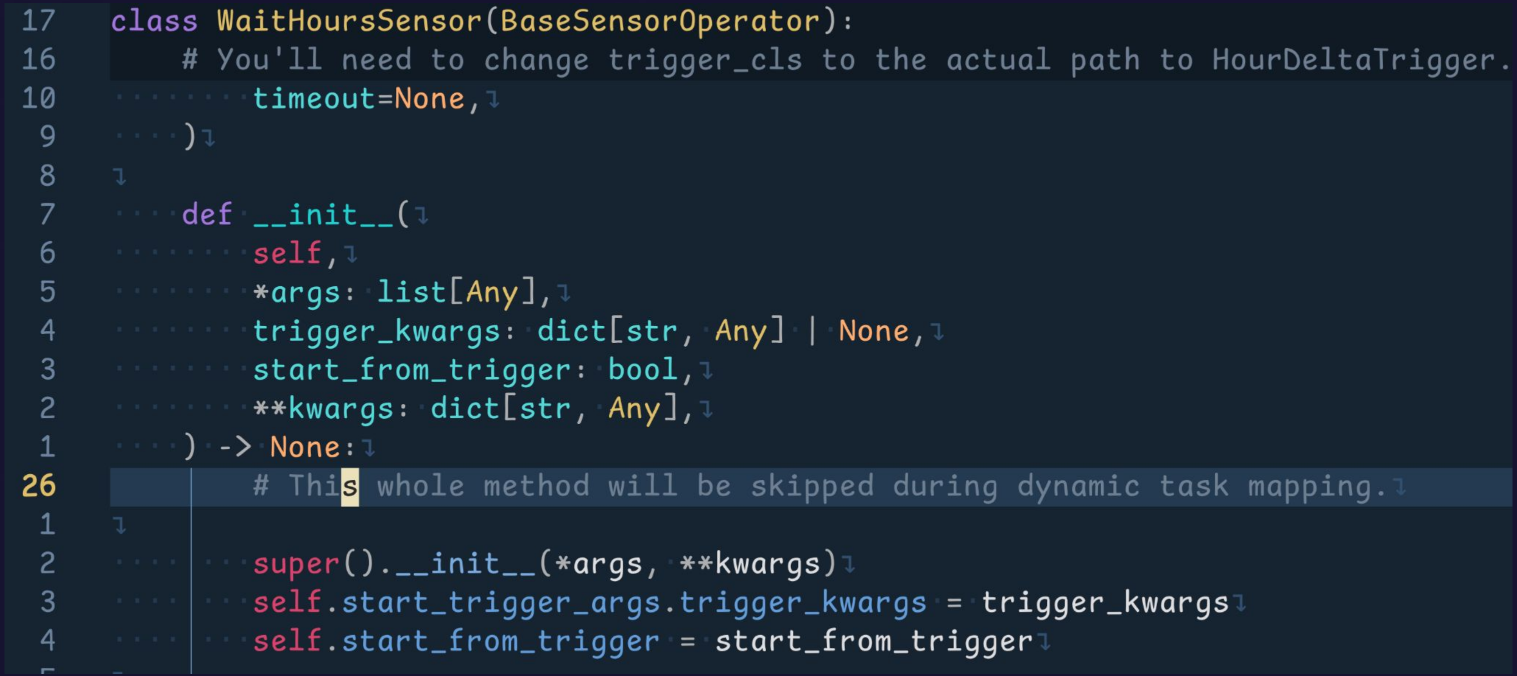
THE CIRCLE IS NOW COMPLETE.

ASTRONOMER

Well yes, but actually no

Limitation (Start execution from the trigger)

Limited dynamic task mapping support

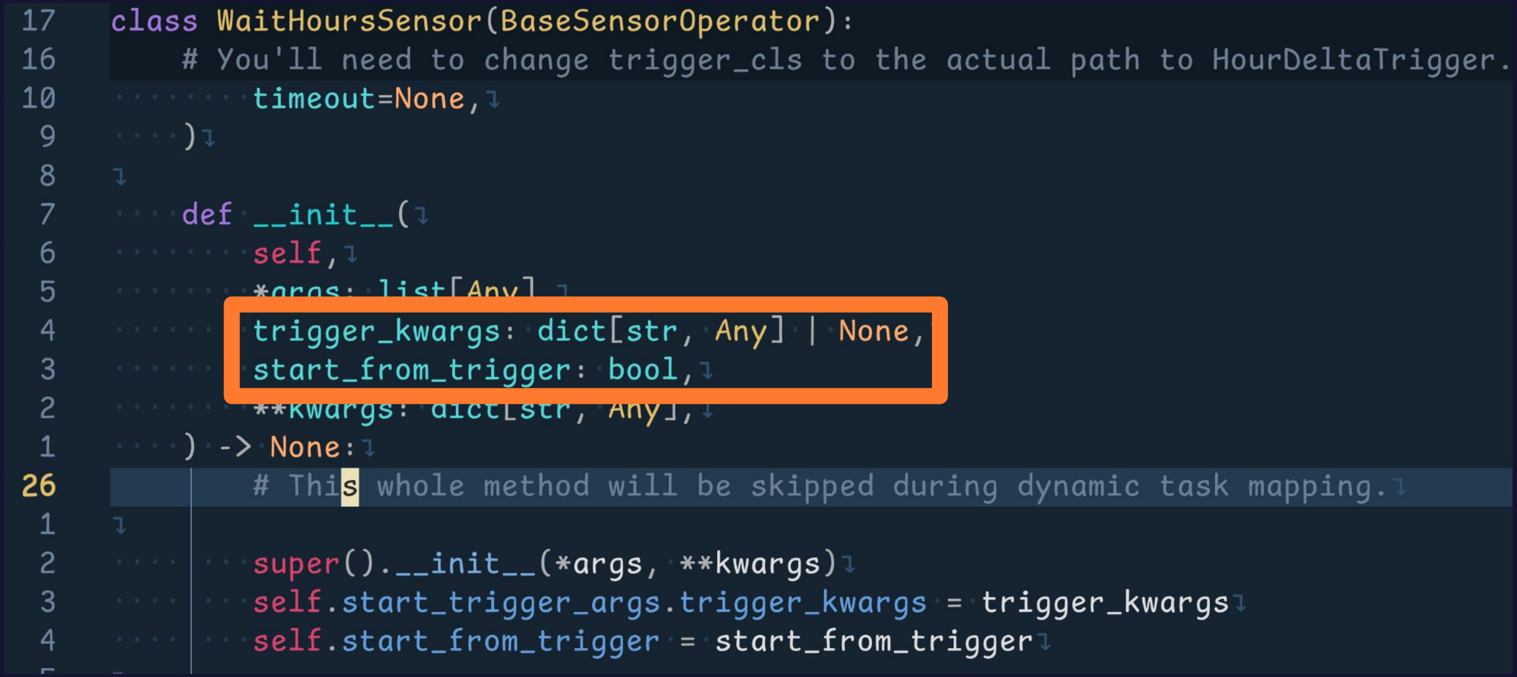


```python
class WaitHoursSensor(BaseSensorOperator):
    # You'll need to change trigger_cls to the actual path to HourDeltaTrigger.
        timeout=None,
    )

    def __init__(
        self,
        *args: list[Any],
        trigger_kwargs: dict[str, Any] | None,
        start_from_trigger: bool,
        **kwargs: dict[str, Any],
    ) -> None:
        # This whole method will be skipped during dynamic task mapping.

        super().__init__(*args, **kwargs)
        self.start_trigger_args.trigger_kwargs = trigger_kwargs
        self.start_from_trigger = start_from_trigger
```

ASTRONOMER

Limitation (Start execution from the trigger)

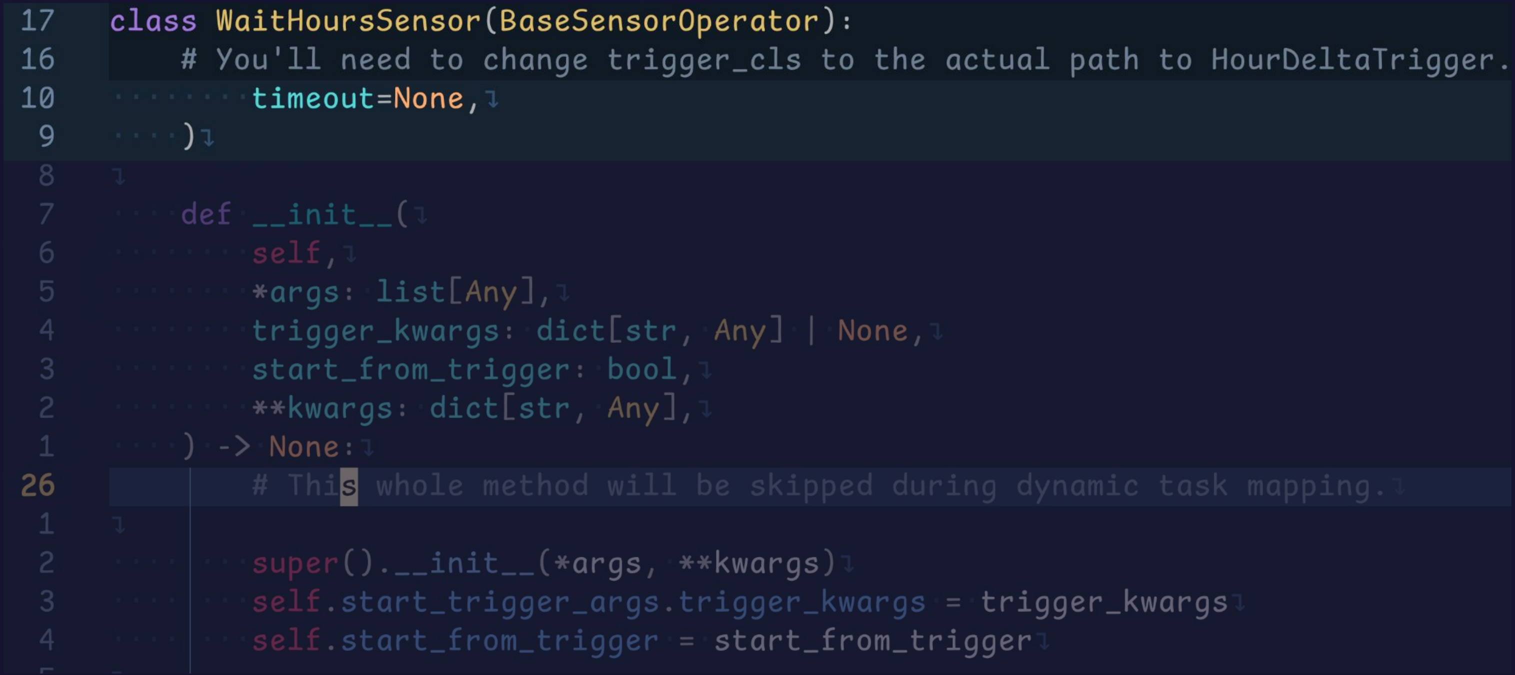
trigger_kwargs, start_from_trigger required in __init__



```
class WaitHoursSensor(BaseSensorOperator):
    # You'll need to change trigger_cls to the actual path to HourDeltaTrigger.
        timeout=None,
    )

    def __init__(
        self,
        *args: list[Any],
        trigger_kwargs: dict[str, Any] | None,
        start_from_trigger: bool,
        **kwargs: dict[str, Any],
    ) -> None:
        # This whole method will be skipped during dynamic task mapping.

        super().__init__(*args, **kwargs)
        self.start_trigger_args.trigger_kwargs = trigger_kwargs
        self.start_from_trigger = start_from_trigger
```

Limitation (Start execution from the trigger)

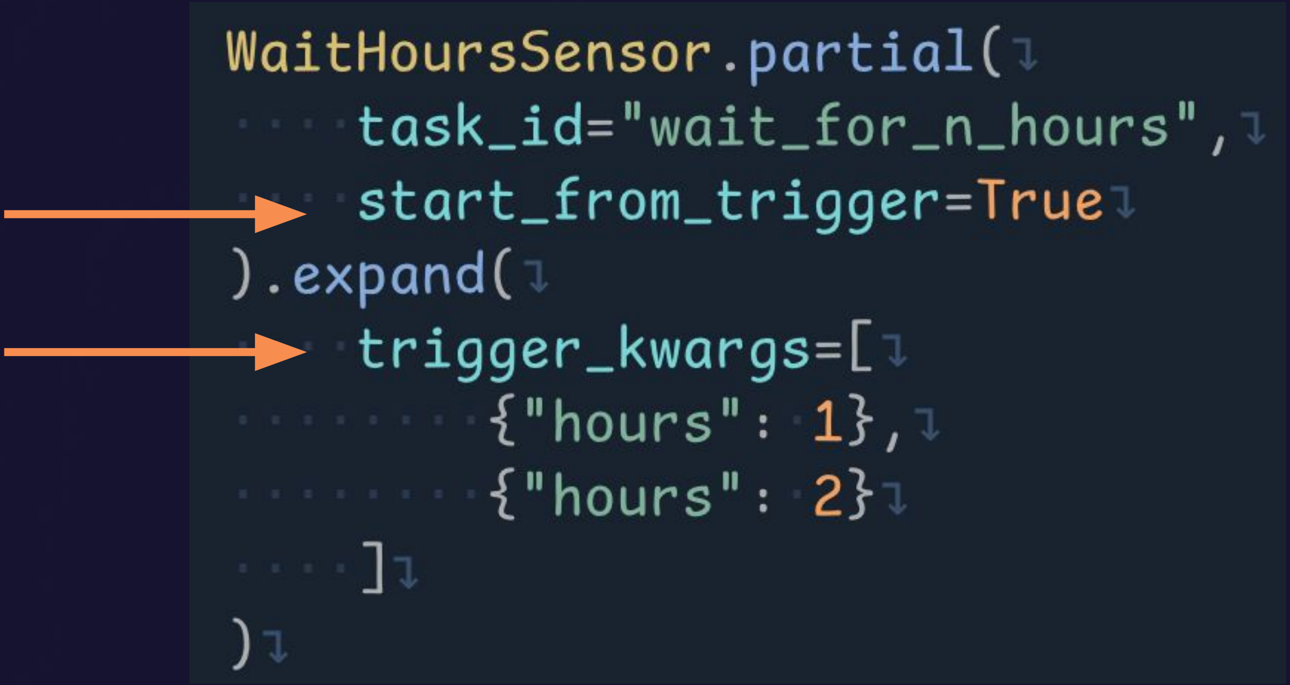
the whole **__init__ method skipped before execution**



```
class WaitHoursSensor(BaseSensorOperator):
    # You'll need to change trigger_cls to the actual path to HourDeltaTrigger.
        timeout=None,
    )

    def __init__(
        self,
        *args: list[Any],
        trigger_kwargs: dict[str, Any] | None,
        start_from_trigger: bool,
        **kwargs: dict[str, Any],
    ) -> None:
        # This whole method will be skipped during dynamic task mapping.

        super().__init__(*args, **kwargs)
        self.start_trigger_args.trigger_kwargs = trigger_kwargs
        self.start_from_trigger = start_from_trigger
```

Limitation (Start execution from the trigger)

slightly different syntax

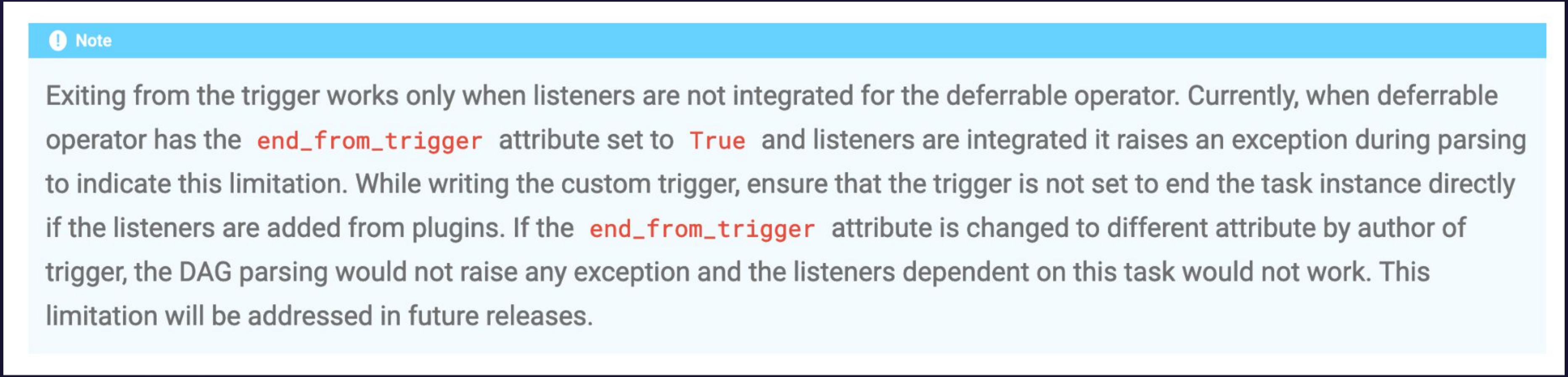


```
WaitHoursSensor.partial(
    task_id="wait_for_n_hours",
    start_from_trigger=True
).expand(
    trigger_kwargs=[
        {"hours": 1},
        {"hours": 2}
    ]
)
```

If you don’t know if you need to combine this feature with dynamic task mapping

Limitation (End execution from the trigger)

Doesn't support listeners



Note

Exiting from the trigger works only when listeners are not integrated for the deferrable operator. Currently, when deferrable operator has the `end_from_trigger` attribute set to `True` and listeners are integrated it raises an exception during parsing to indicate this limitation. While writing the custom trigger, ensure that the trigger is not set to end the task instance directly if the listeners are added from plugins. If the `end_from_trigger` attribute is changed to different attribute by author of trigger, the DAG parsing would not raise any exception and the listeners dependent on this task would not work. This limitation will be addressed in future releases.

QR Code links to my posts related to this talk

Thank you!
Any questions?

ASTRONOMER