# AIP-35 Add Signal Based Scheduling To Airflow

## Status

| | |
|---|---|
| **State** | Draft |
| **Discussion Thread** | [DISCUSS][AIP-35] Add Signal Based Scheduling To Airflow |
| **Slack Thread** | Proposal Discussion of Signal Based Scheduling |
| **Issue** | Add Signal Based Scheduling To Airflow |
| **Created** | $action.dateFormatter.formatGivenString("yyyy-MM-dd", $content.getCreationDate()) |

## Motivation

Airflow scheduler uses DAG definitions to monitor the state of tasks in the metadata database, and triggers the task instances whose dependencies have been met. It is based on state of dependencies scheduling.

However, the current design has the following caveats:

1. When the workflow contains streaming jobs, the scheduler can't work because the streaming job runs forever.
2. The communication between the operator and scheduler has a long latency of the database query interval.

In order to address the issues, we propose to add signal based scheduling to the scheduler.

The idea of signal based scheduling is to let the operators send signals to the scheduler to trigger a scheduling action, such as starting jobs, stopping jobs and restarting jobs. With this change, a streaming job can send signals to the scheduler to indicate the state change of the dependencies. This way, the workflow could support a mix of streaming and batch jobs.

Also, compared with the current state change retrieval mechanism, signal based scheduling allows the scheduler to know the change of the dependency state immediately without periodically querying the metadata database.

In addition to that, signal based scheduling allows potential support for richer scheduling semantics such as periodic execution and manual trigger at per operator granularity.

## Considerations

### Public Interfaces

#### Signal Operator

- Introduce a new class of SignalOperator for the operators that send signals.
- Let BaseOperator inherit SignalOperator.

**Signal Operator**

```
class SignalAction(str, Enum):
    """
    NONE: No action needs to be taken.
    START: Start to initialize the task instance.
    RESTART: Start to initialize the task instance or stop running the task instance and start the task
instance.
    STOP: Stop running the task instance.
    """
    NONE = "NONE"
    START = "START"
    RESTART = "RESTART"
    STOP = "STOP"

class Signal(object):
    """
    Signal describes an event in the workflow.
    """

    def __init__(self, key: str, value: str):
        self.key = key
        self.value = value

class SignalVersion(Signal):
    """
```

```
    SignalVersion represents the version, a.k.a. epoch, of a signal.
    """

    def __init__(self, key: str, value: str, version: int):
        super().__init__(key = key, value = value)
        self.version = version

class SignalOperator(Operator):
    """
    SignalOperator introduces a new attribute _signals to hold user-defined signals.
    """

    def __init__(self):
        """
        Attribute _signals represents signals defined on the operator.
        """
        self._signals: Set[Signal] = {}

    @abstractmethod
    def set_signal(self, signal: Signal):
        """
        Set the user-defined signal.

        :param: signal: Specific user-defined signal.
        """
        pass

    @property
    def signals(self) -> Set[Signal]:
        """
        Return the set of signals for the operator.

        :return: Set of signals defined on the operator.
        """
        return self._signals

    @abstractmethod
    def on_signal(self, signal: Signal) -> SignalAction:
        """
        Return corresponding action upon the specific signal is received.

        :param: signal: Specific user-defined signal.
        :return: Corresponding action of user-defined signal.
        """
        pass

class BaseOperator(SignalOperator, LoggingMixin):
    """
    BaseOperator contains the following attributes:
    _upstream_task_ids: maintain upstream relationships.
    _downstream_task_ids: maintain downstream relationships.
    _signals: represents signals defined on operator.
    dep(): return set of dependencies for operator.
    """

class PythonOperator(BaseOperator):
     """
     After inheriting BaseOperator, PythonOperator could use attribute _siganls to drive
     the execution of Python callable.
     """
```

### Signal Notification Service

- Introduce a new component of Signal Notification Service to receive and distribute signals.

**Signal Notification Service**

```
class SignalService(metaclass=abc.ABCMeta):
     """
     SignalService receives and propagates the signals from the operators and
     other sources to the scheduler.
     """

    def send_signal(self, key: str, value: str):
        """
        Send signal with given key and value in Notification Service.

        :param key: Key of signal updated.
        :param value: Value of signal updated.
        :return: A single object of signal notification created in Notification service.
        """
        pass

    def list_signals(self, key: str, version: int = None) -> list:
        """
        List specific `key` or `version` of signals in Notification Service.

        :param key: Key of the signal for listening.
        :param version: (Optional) Version of the signal for listening.
        :return: Specific `key` or `version` signal notification list.
        """
        pass

     def listen_signal(self, listener_name: str, key: str,
                                                    watcher: SignalWatcher,
                                                    version: int = None):
        """
        Listen to specific `key` or `version` of signal in Notification Service.

        :param listener_name: Name of registered listener to listen signal notification.
        :param key: Key of the signal for listening.
        :param watcher: Watcher instance for listening to the signal.
        :param version: (Optional) Version of the signal for listening.
        """
        pass

class SignalWatcher(metaclass=abc.ABCMeta):
     """
     SignalWatcher is used to represent a standard event handler, which defines the
     logic related to signal notifications.
     """

    @abstractmethod
    def process(self, signals: Set[Signal]):
        pass
```

## Proposed Changes

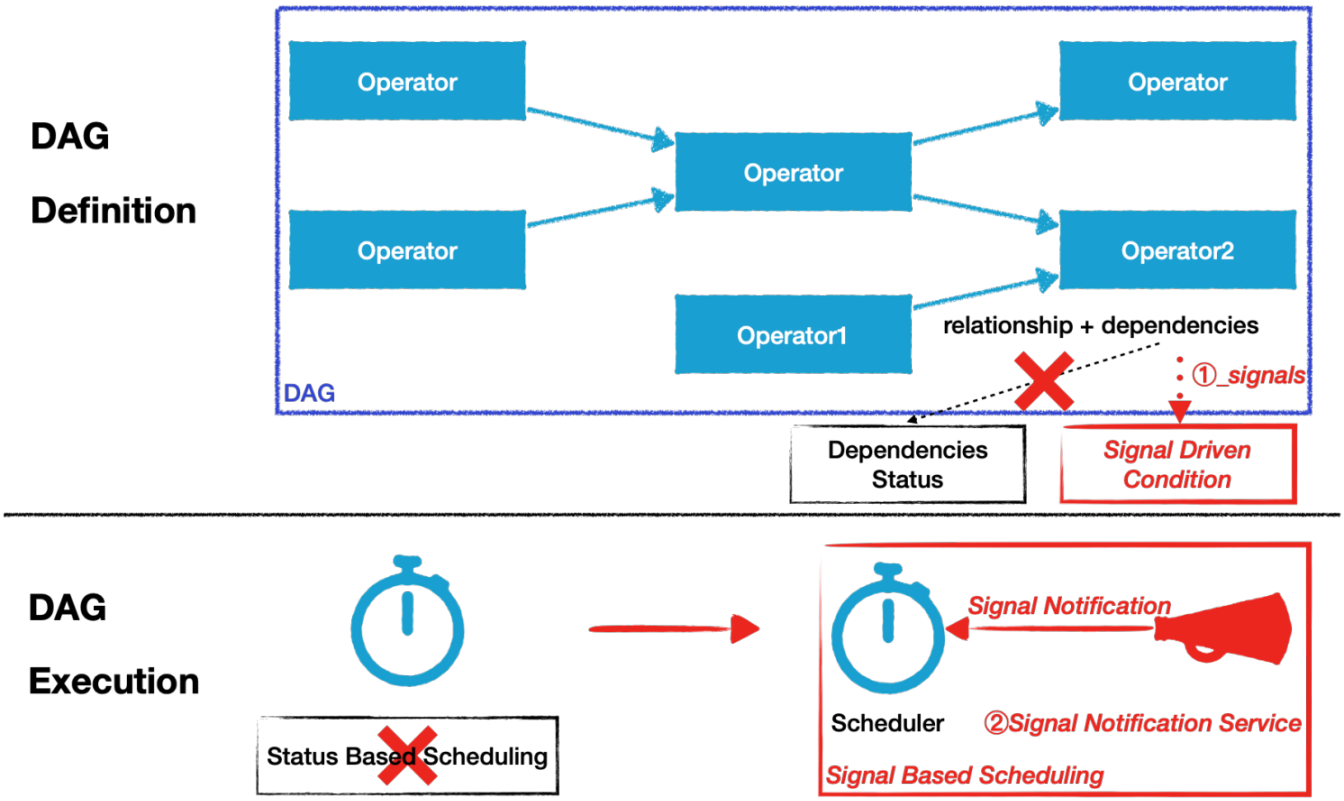


Fig 1. Changes of signal-based scheduling in Airflow Scheduler.

Changes of signal-based scheduling proposed to Airflow Scheduler involves two sections as Fig 1 shows:

1. Signal driven condition replaces the mechanism driven by dependencies status in DAG definition.
    a. For example, in Fig1, Operator2 is the downstream of Operator1. Whether to execute the corresponding task instance of Operator2 originally depends on the task's execution state of Operator1. Currently Operator2 determines the execution of the task instance according to the signal from Operator1.
2. Signal-based scheduler replaces original status-based mechanism of scheduling from perspective of DAG execution. Airflow's scheduler is a process that uses DAG definitions in conjunction with the state of tasks in the metadata database to decide which tasks need to be executed. Currently the scheduler replaces the communication between Scheduler and Worker with signal notification service, getting rid of directly periodically querying states of task instances in the database.

According to logic of the SchedulerJob involved, Airflow's scheduler currently is state-based scheduling which relies on the dependencies among operators. Compared with the state-based scheduling (STBS) mechanism, signal-based scheduling (SGBS) consists of two major parts:

- ***Signals*** are used to define conditions that must be met to run an operator. State change of the upstream tasks is one type of the signals. There may be other types of signals. The scheduler may take different actions when receiving different signals. To let the operators take signals as their starting condition, we propose to introduce ***SignalOperator*** which is mentioned in the public interface section.
- A ***notification service*** is necessary to help receive and propagate the signals from the operators and other sources to the scheduler. Upon receiving a signal, the scheduler can take action according to the predefined signal-based conditions on the operators. Therefore we propose to introduce a ***Signal Notification Service*** component to Airflow.

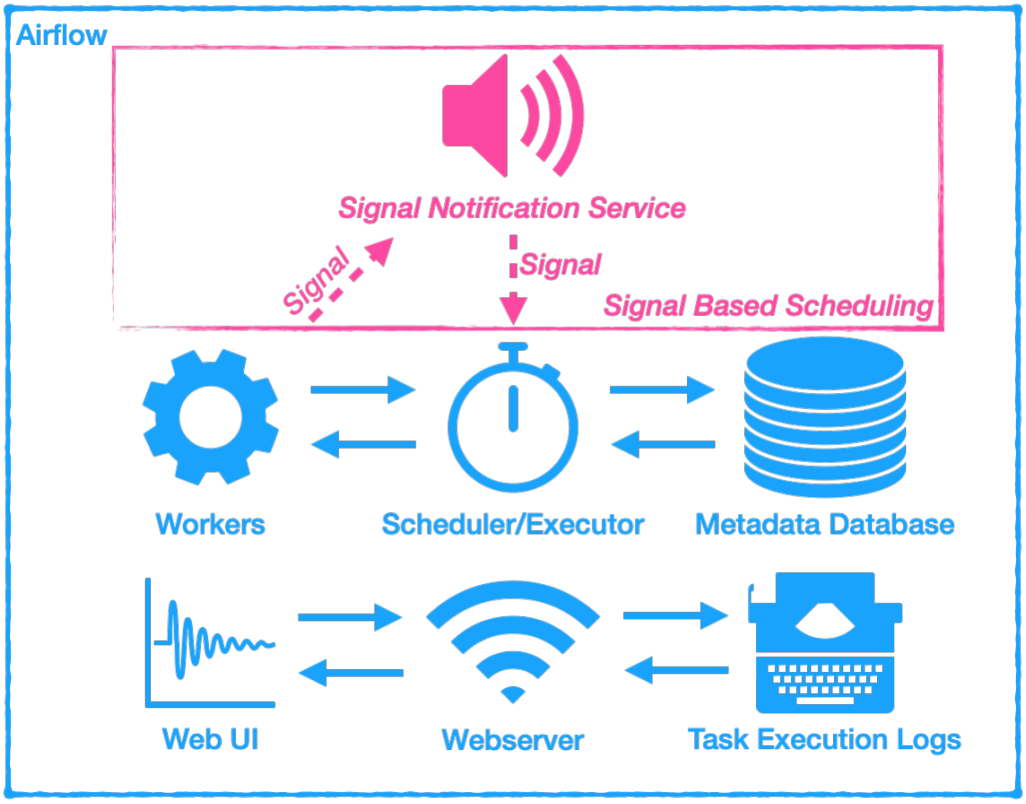


Fig 2. Signal and Signal Notification Service in Airflow

We discuss these two parts respectively in the following sections.

## What change do you propose to make?

### Signal Operator

Signal defines conditions that must be met for the action of the task instance. Scheduler switches to use signals and the corresponding condition definition to decide whether to run a task instance or not. Definitions of signal condition are sufficient and necessary.

- If it is a sufficient condition, the corresponding action is performed depending on the node matching signal success.
- If it is a necessary condition, all the dependent upstream requirements must be met before the corresponding action is performed.

Once the conditions are met, there are three types of signal actions: START, RESTART and STOP.

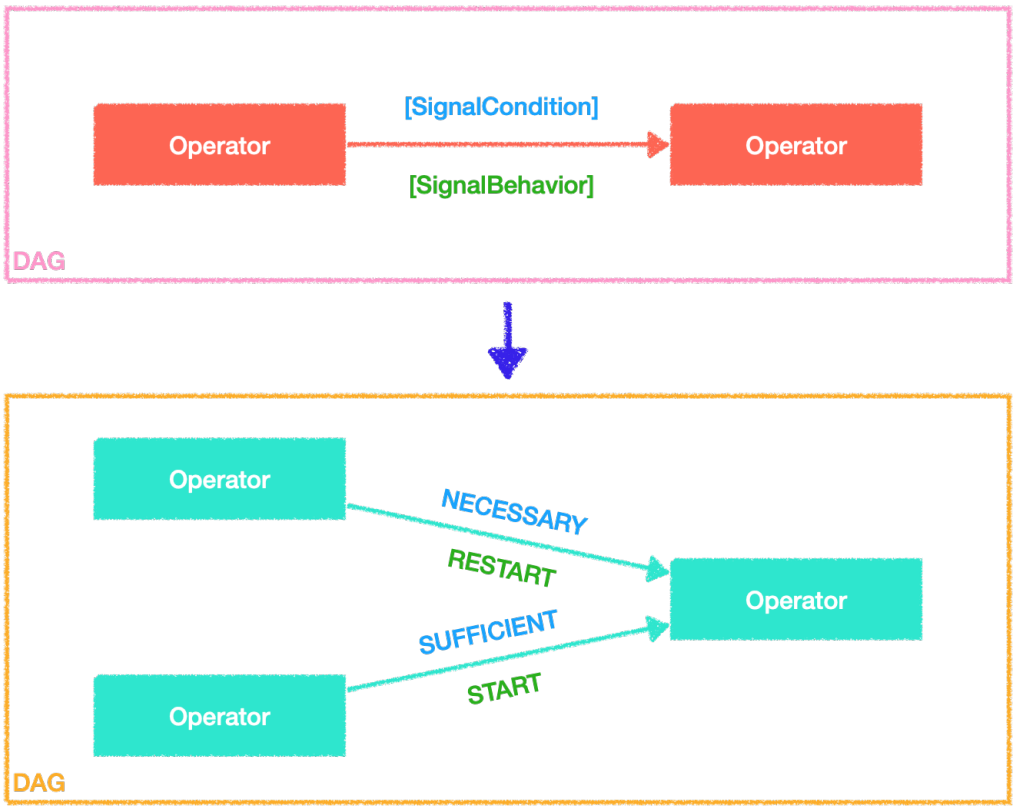


Fig 3. SignalCondition and SignalAction in Signal Driven Condition

SignalOperator introduces a new attribute ***_signals*** to hold user-defined signals. The SignalOperator maintains the epoch of signals of current and the last task instance running from operators individually. Taking compatibility into consideration, BaseOperator, the current implementation of operator that contains recursive methods for dag crawling behavior, should inherit from SignalOperator. Thus BaseOperator may contain the following attributes and properties:

- _upstream_task_ids - represents ids of upstream task instances.
- _downstream_task_ids - represents ids of downstream task instances.
- _signals - represents all signals defined on this operator, including
    - user-defined signals.
    - system-defined signals, e.g. the signals representing the task state changes of the upstream tasks.
- dep() - returns a set of dependencies for operators.

The system-defined signals are passed as an argument to the constructor of a SignalOperator. User-defined signals can be set via ***SignalOperator#set_signal()*** interface.

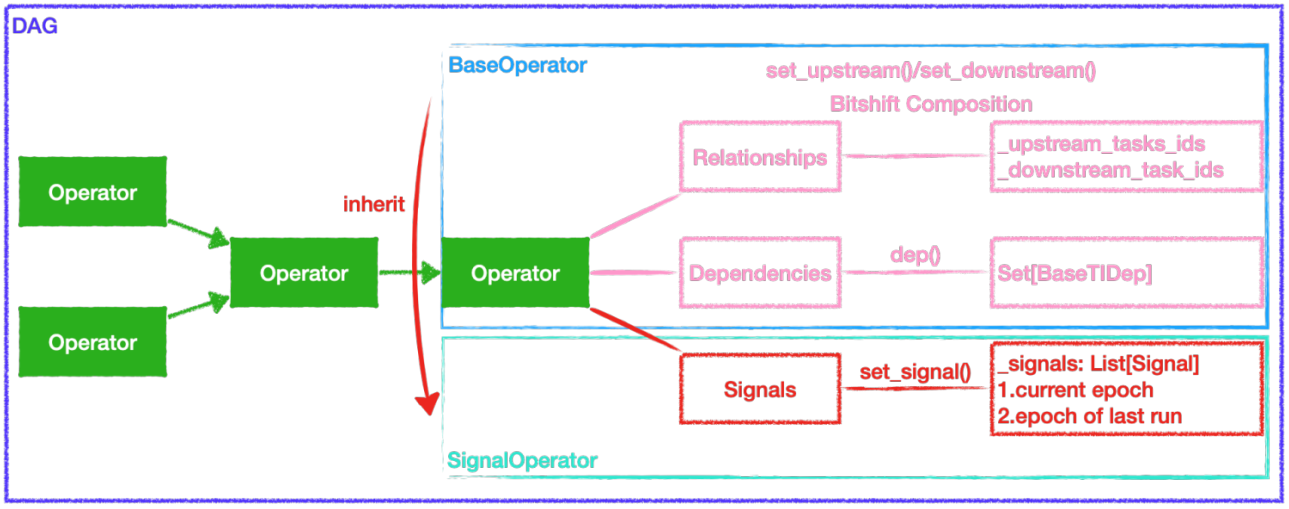


Fig 4. BaseOperator inherits SignalOperator

### Signal Notification Service

The signal notification service is a pluggable component to transmit signals from the operators to the scheduler. It allows the scheduler to register SignalListeners. Each signal listener is registered on a signal key. Once the signal notification service receives a signal of a certain key, it will notify the signal listener of the new signal arrival. In Airflow Scheduler,  Executor is a message queuing process that is tightly bound to the Scheduler and carries out the scheduling decision made by the scheduler. Integration of Scheduler and Worker with signal notification service includes mainly two aspects:

- Scheduler listens to notifications of signals and determines the worker executions of each scheduled task based on condition and action of received signal notifications.
- Worker sends signals to the notification service which will then notify the Scheduler of the new signals.

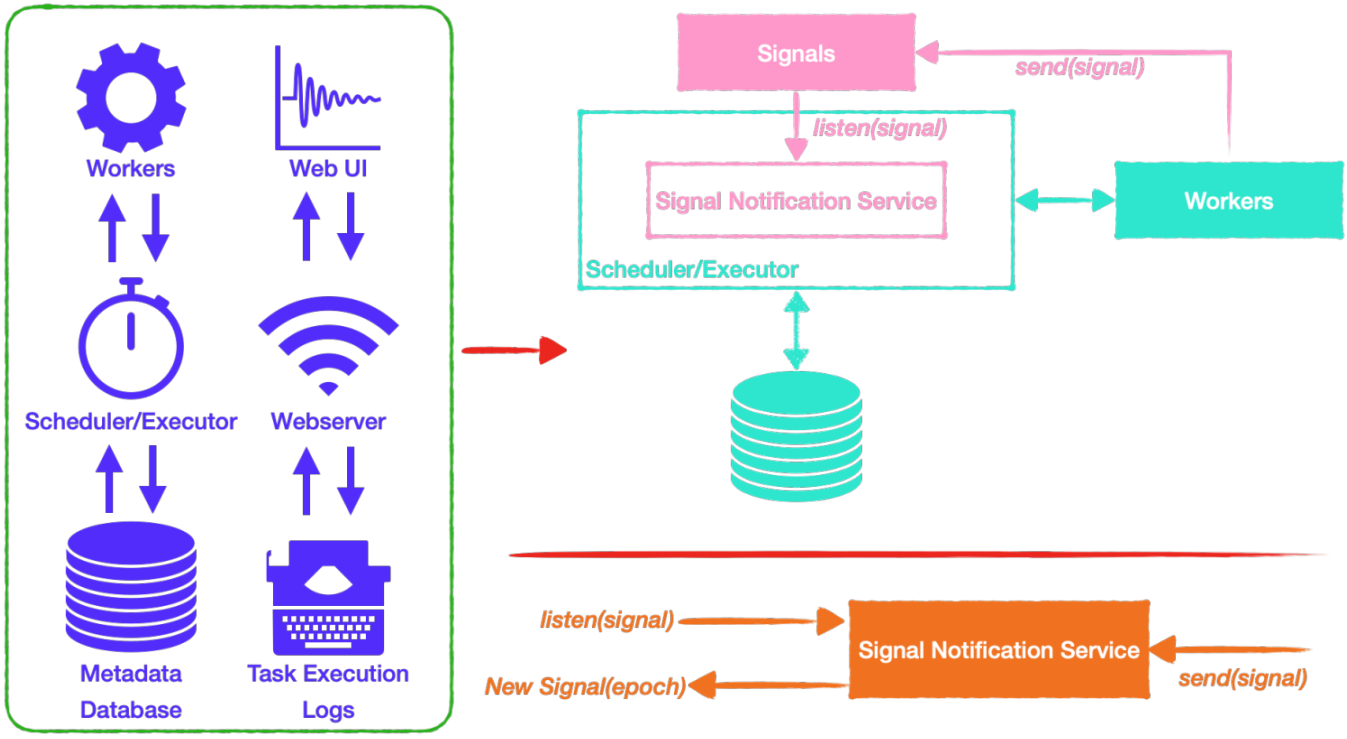


Fig 5. Signal Notification Service in Scheduler

Signal notification service mainly provide the following interfaces:

- send_signal - update specific key and value of the signal notification.
- list_signals - query required `key` or `version` of the signal notifications.
- listen_signal - listen to required `key` or `version` of the signal notification.

Default implementation of NotificationService is long polling for notifications of signals stored in the metadata database, and directly updating signal notification metadata in order to notify the listener that signal for dependencies status of operator has already been updated. Otherwise, signal notification component provides user-defined online signal notification service implementation(for example GRPC) based on SPI mechanism.

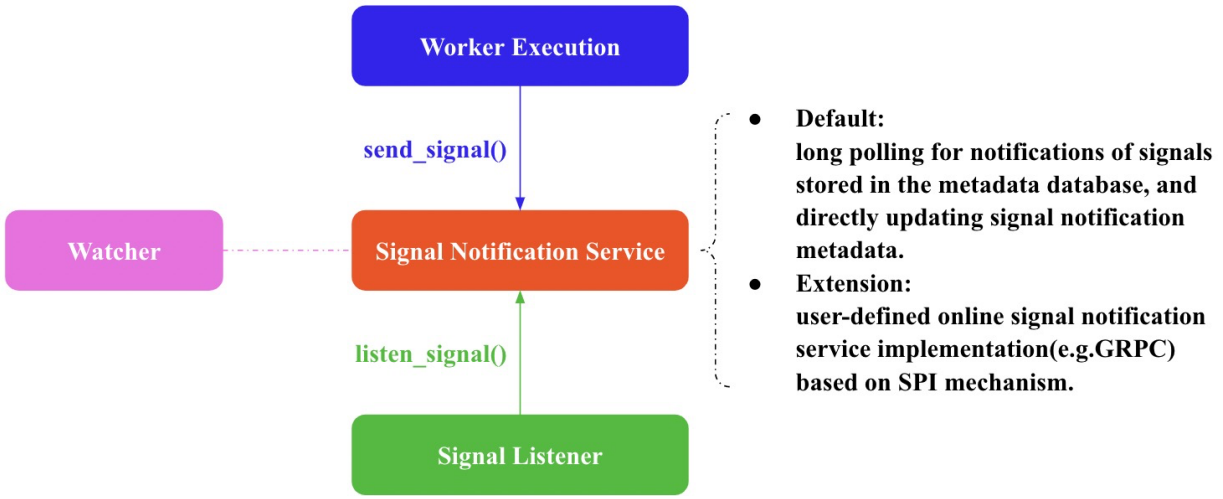


Fig 6. Implementation of Signal Notification Service

Introduced by notification service module, basic operations of signal based scheduling for SchedulerJob in pseudocode:

Step 0. Load available DAG definitions from disk (fill DagBag)

While the scheduling loop is running:

Step 1. The scheduling loop uses the DAG definitions to identify and/or initialize any DagRuns in the metadata db.

Step 2. The scheduling loop listens to the signal notifications from the active DagRuns. Upon receiving a signal notification, the scheduler loop checks with the SignalOperators to determine the action and then carries out that action. More specifically, it identifies TaskInstances that need to be executed and adds them to a worker queue.

Step 3. Each available worker pulls a TaskInstance from the queue and executes it, sends a signal to the notification service a signal indicating the task state changes from “queued” to “running”. The scheduling loop will then update the task state in the database when receiving this signal..

Step 4. Once a TaskInstance has finished running, the associated worker sends a signal of TaskInstance termination. (e.g. "finished", "failed", etc.)

Step 5. The scheduler updates the states of all active DagRuns ("running", "failed", "finished") according to the states of all completed associated TaskInstance.

Step 6. Repeat Steps 1-5

## How are users affected by the change? (e.g. DB upgrade required?)

- About signal operator implementation: SignalOperator won’t affect the existing functionality of BaseOperator and corresponding implementations. SignalOperator doesn’t change the relationship and dependencies between the corresponding task instances. In other words, adding SignalOperator is fully backwards-compatible.
- About signal notification service component: Signal based scheduling maintains the original long poll states of task instances in a metadata database with the default implementation of notification service, not affecting the original scheduling process. We do not expect any user sensible changes.
- About Scheduler and Worker module integration: Signal based scheduling doesn’t change the original process of SchedulerJob, and only adds signal listeners for change of signals in the Scheduler module and sends signals when Worker finishes to execute a task instance. This is essentially a change of underlying state change propagation mechanism, which should be transparent to the end users.

## What defines this AIP as "done"?

- Implement signal operator and signal notification service.
- Integrate Scheduler and Worker module with signal operator and signal notification service.
- Support periodic execution and manually triggered jobs semantics.