106 - Get more out of your workflows: Interact with workflows, run tasks concurrently, and create advanced automation triggers

106 Agenda

- Interact with workflows: pause workflows for human input
- Prioritize and limit work
- Use more advanced triggers in automations
- Run tasks concurrently
- Test workflows

Interactive workflows

Interactive workflows

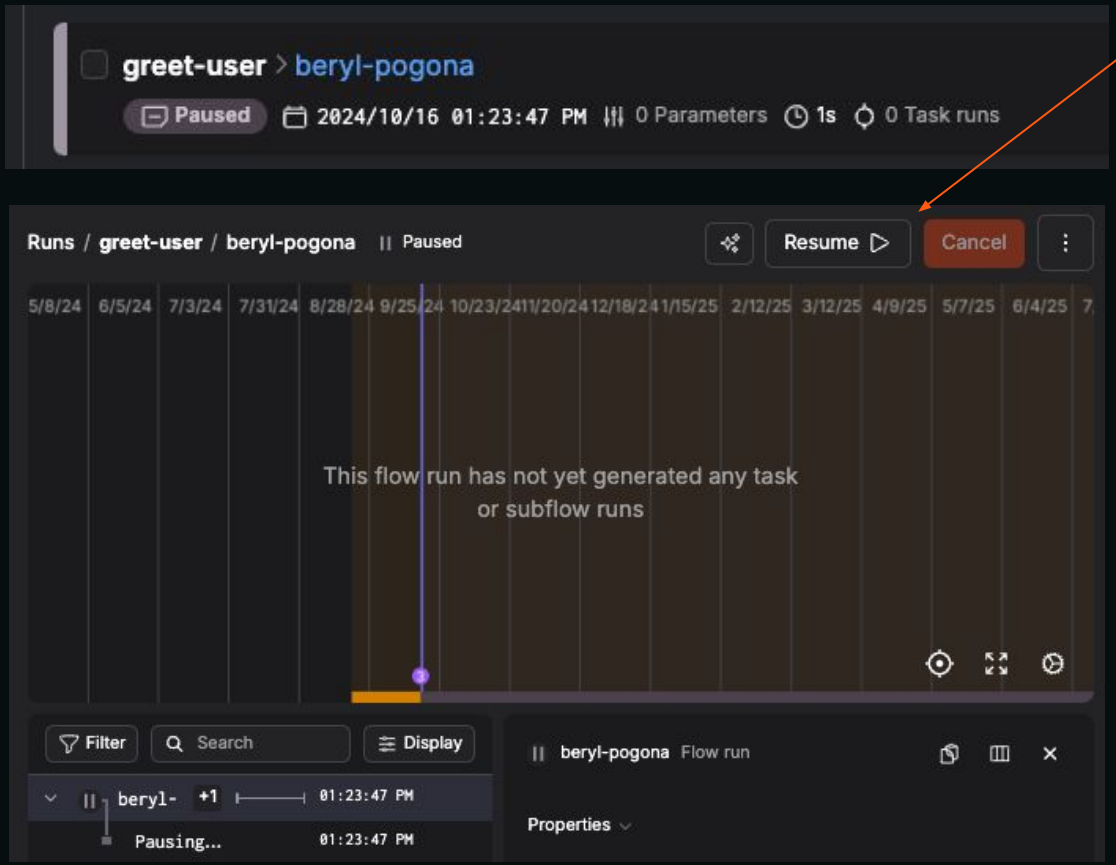
Pause a flow run to wait for input from a user via a web form (human-in-the-loop) 🙂

pause_flow_run function

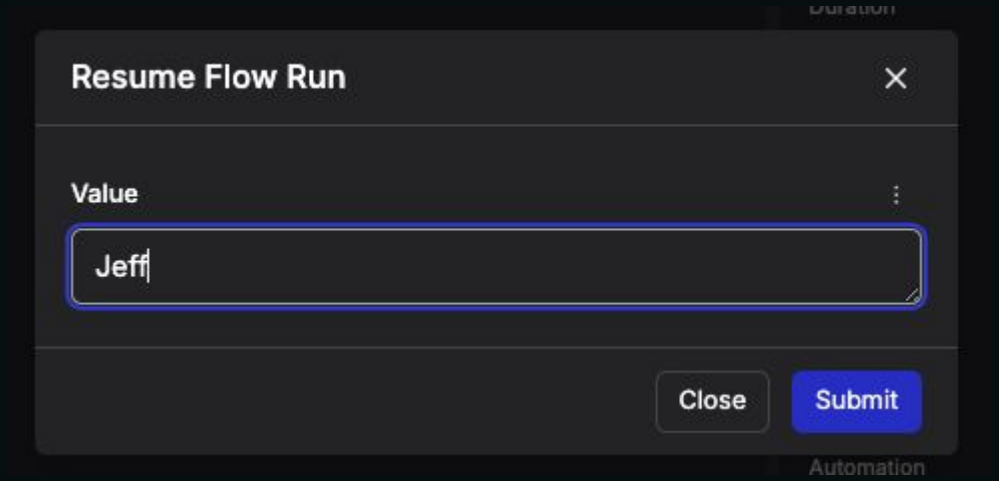
Human-in-the-loop: basic

```python
from prefect import flow, pause_flow_run


@flow(log_prints=True)
def greet_user():
    name = pause_flow_run(str)
    print(f"Hello, {name}!")


if __name__ == "__main__":
    greet_user()
```

Human-in-the-loop: basic



Human-in-the-loop: basic



Human-in-the-loop: basic

Runs / greet-user / beryl-pogona Completed

M 1:24:00 PM 1:24:30 PM 1:25:00 PM 1:25:30 PM 1:26:00 PM 1:26:30 PM

This flow run did not generate any task or subflow runs

Filter Search Display

beryl-pogona +4 01:23:47 PM

Pausing flow, execution will continue when this fl... 01:23:47 PM

Resuming flow run execution! 01:26:49 PM

Hello, Jeff! 01:26:49 PM

Finished in state Completed() 01:26:49 PM

beryl-pogona Flow run

Properties

State Completed

Scheduled start 2024/10/16 01:23:47 PM

Start 2024/10/16 01:23:47 PM

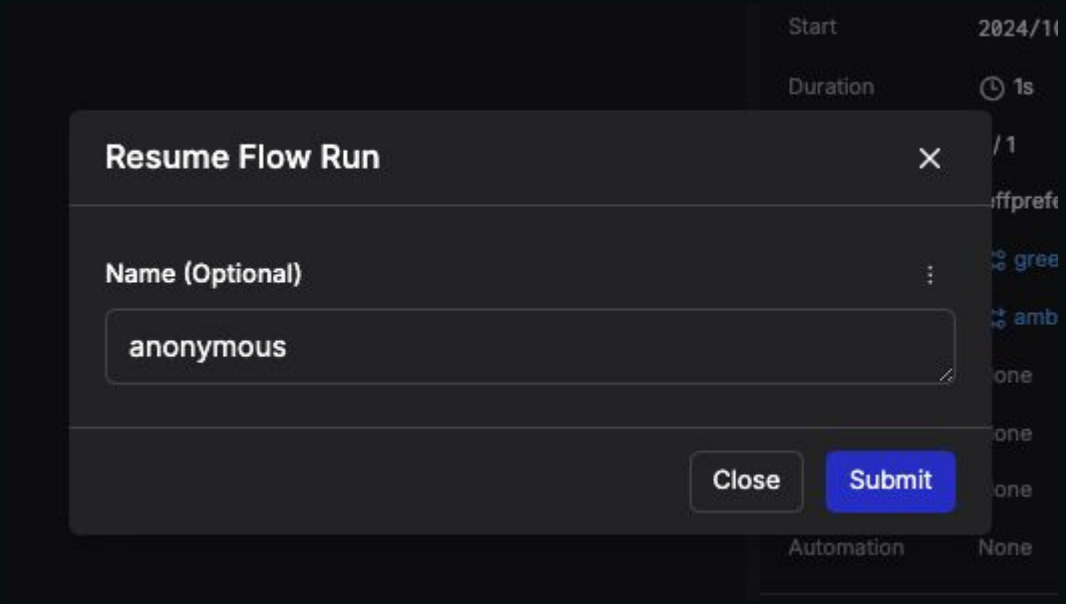
Human-in-the-loop: options

- Validate using *RunInput* class (subclass of Pydantic's *BaseModel*)
- Specify default value
- Create dropdown

Human-in-the-loop: default value

```python
import asyncio
from prefect import flow, pause_flow_run
from prefect.input import RunInput


class UserNameInput(RunInput):
    name: str


@flow(log_prints=True)
async def greet_user():
    user_input = await pause_flow_run(
        wait_for_input=UserNameInput.with_initial_data(name="anonymous")
    )

    if user_input.name == "anonymous":
        print("Hello, stranger!")
    else:
        print(f"Hello, {user_input.name}!")


if __name__ == "__main__":
    asyncio.run(greet_user())
```

Human-in-the-loop: default value



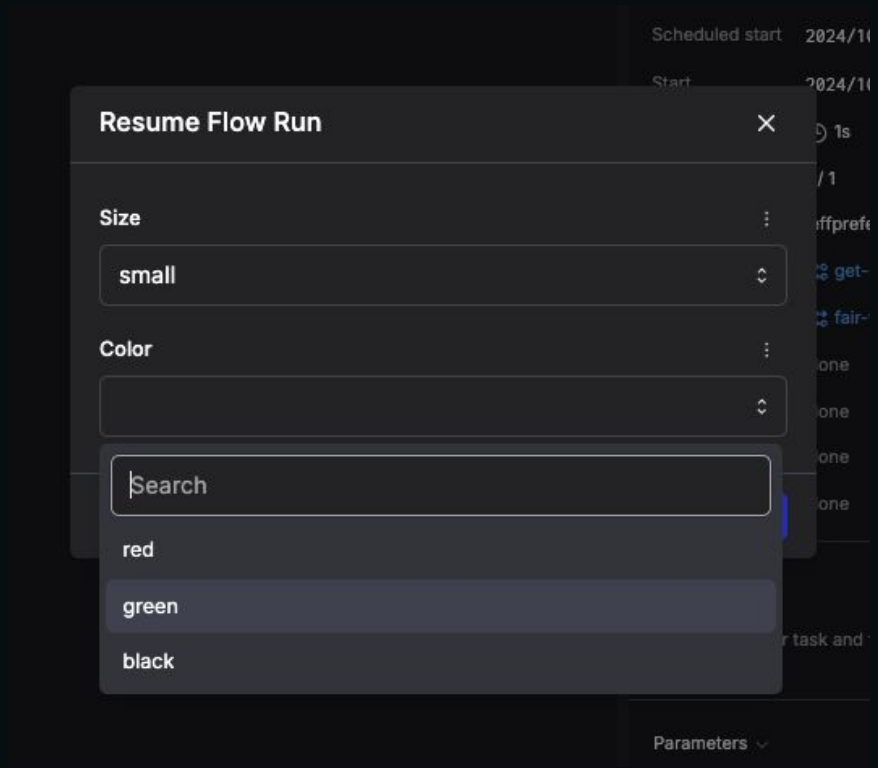
Human-in-the-loop: custom validation

```python
from typing import Literal
import pydantic
from prefect import flow, pause_flow_run
from prefect.input import RunInput


class ShirtOrder(RunInput):
    size: Literal["small", "medium", "large", "xlarge"]
    color: Literal["red", "green", "black"]

    @pydantic.validator("color")
    def validate_age(cls, value, values, **kwargs):
        if value == "green" and values["size"] == "small":
            raise ValueError("Green is only in-stock for medium, large, and XL sizes.")

        return value
```

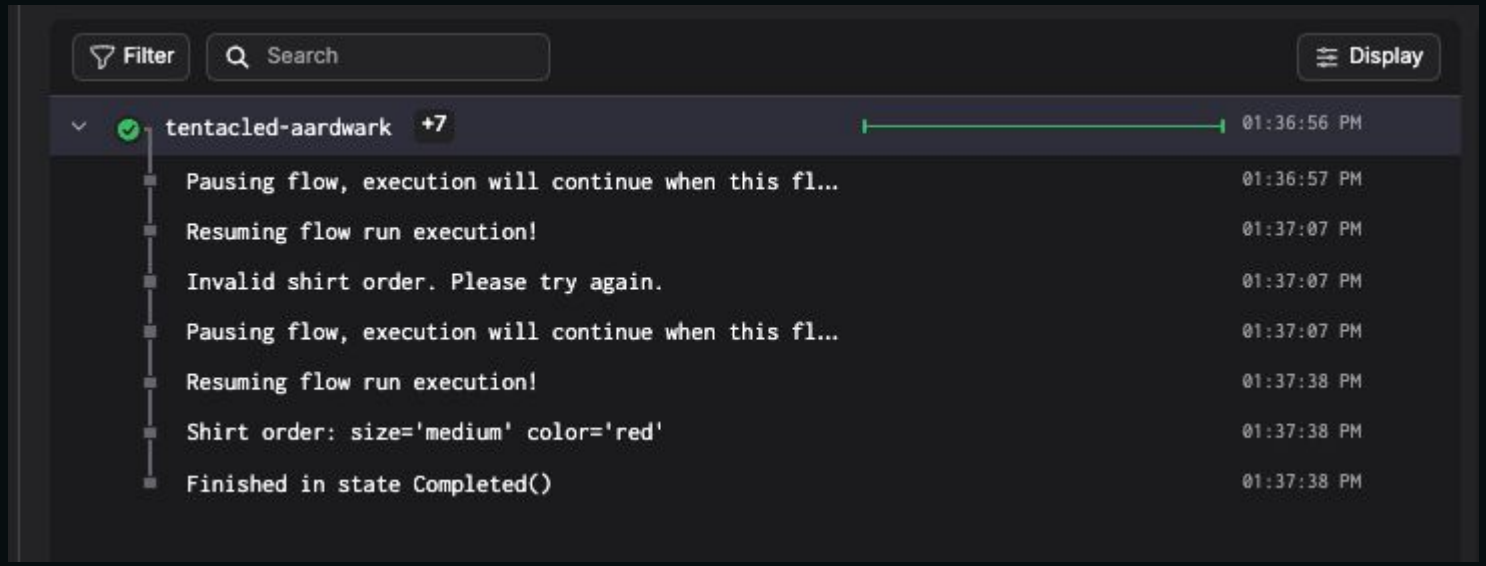
Human-in-the-loop: custom validation

```python
@flow(log_prints=True)
def get_shirt_order():
    shirt_order = None

    while shirt_order is None:
        try:
            shirt_order = pause_flow_run(wait_for_input=ShirtOrder)
            print(f"Shirt order: {shirt_order}")
        except pydantic.ValidationError:
            print("Invalid shirt order. Please try again.")
```

Human-in-the-loop: custom validation



Human-in-the-loop: custom validation



Human-in-the-loop

Potential uses:

- Finance approval prior to running an expensive workflow
- AI classifier labeling - request human intervention when low confidence in label

Task Runners for easier async

Concurrency & Parallelism

- **Sequential:** run to completion before next run starts
- **Concurrency:** non-blocking, single-thread, interleaving
- **Parallelism:** multiple operations at the same time

Your Prefect code runs **sequentially** by default

Concurrency

Concurrency

- Helpful when waiting for external systems to respond
- Allows other work to be done while waiting
- Use *ThreadPoolTaskRunner* to execute code concurrently

Task Runners

- Specify in *flow* decorator
- Creates a Prefect future object

⚠ Prefect future objects must be resolved explicitly before returning from a flow. Dependencies between futures will be automatically resolved whenever their dependents are resolved. This means that only *terminal* futures need to be resolved, either by:

- returning the terminal futures from your flow
- calling `.wait()` or `.result()` on each terminal future
- using one of the top level `wait` or `as_completed` utilities to resolve terminal futures

Not doing so may leave your tasks in an unfinished state.

Concurrency

```python
from prefect import flow, task
from prefect.futures import wait
from prefect.task_runners import ThreadPoolTaskRunner
import time


@task
def stop_at_floor(floor):
    print(f"elevator moving to floor {floor}")
    time.sleep(floor)
    print(f"elevator stops on floor {floor}")


@flow(task_runner=ThreadPoolTaskRunner(max_workers=3))
def elevator():
    floors = []

    for floor in range(3, 0, -1):
        floors.append(stop_at_floor.submit(floor))

    wait(floors)
```

Concurrency

```
elevator moving to floor 3
elevator moving to floor 1
elevator moving to floor 2
elevator stops on floor 1
15:24:23.662 | INFO    | Task run 'stop_at_floor-599' - Finished in state Completed()
elevator stops on floor 2
15:24:24.661 | INFO    | Task run 'stop_at_floor-06f' - Finished in state Completed()
elevator stops on floor 3
15:24:25.662 | INFO    | Task run 'stop_at_floor-8ac' - Finished in state Completed()
```

Parallelism

Parallelism

- Two or more operations happening at the same time on one or more machines
- Helpful when operations limited by CPU
- Many machine learning algorithms parallelizable

Task Runners for parallelism

- DaskTaskRunner
- RayTaskRunner

Required integration packages:

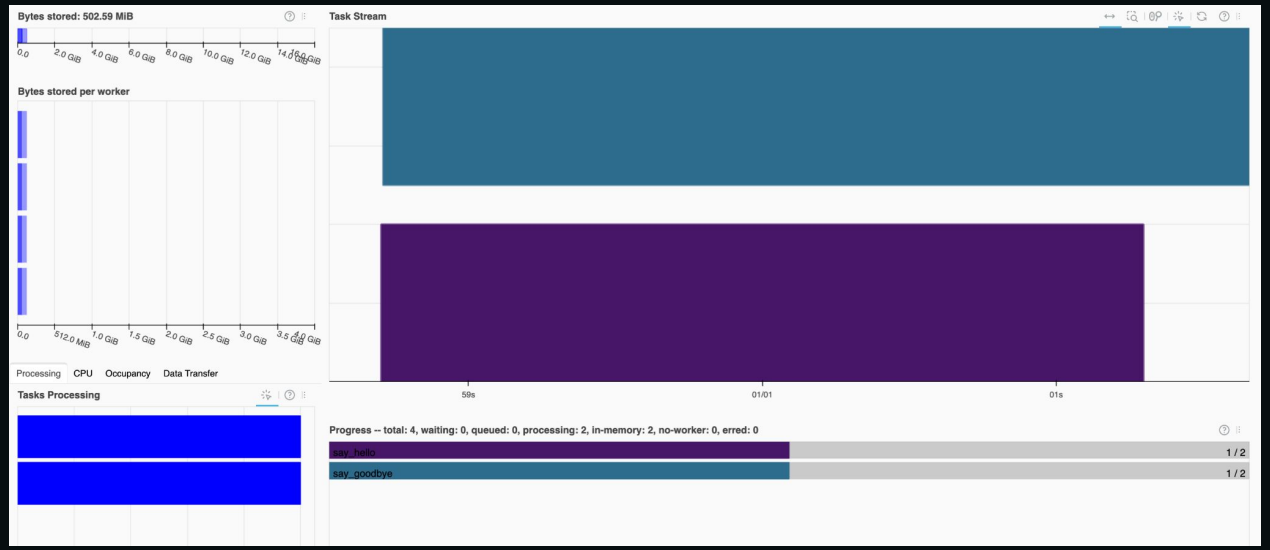
- *prefect-dask*
- *prefect-ray*

DaskTaskRunner for parallelism

```python
from prefect import flow, task
from prefect_dask.task_runners import DaskTaskRunner
import asyncio


@task
async def say_hello(name):
    print(f"hello {name}")


@task
async def say_goodbye(name):
    print(f"goodbye {name}")


@flow(task_runner=DaskTaskRunner())
async def greetings(names):
    for name in names:
        say_hello.submit(name)
        say_goodbye.submit(name)


if __name__ == "__main__":
    asyncio.run(greetings(["arthur", "trillian", "ford", "marvin"]))
```

`DaskTaskRunner` for parallelism

- Can see the Dask UI if have bokeh package installed: *pip install -U bokeh*
- UI will be linked in the terminal at run time



Only use concurrency or parallelism if needed

- Concurrency for heavy IO
- Parallelism for heavy compute

Prioritize and limit work

What's a work queue for?

- Prioritize work
- Limit concurrent runs
- A work pool can have many work queues

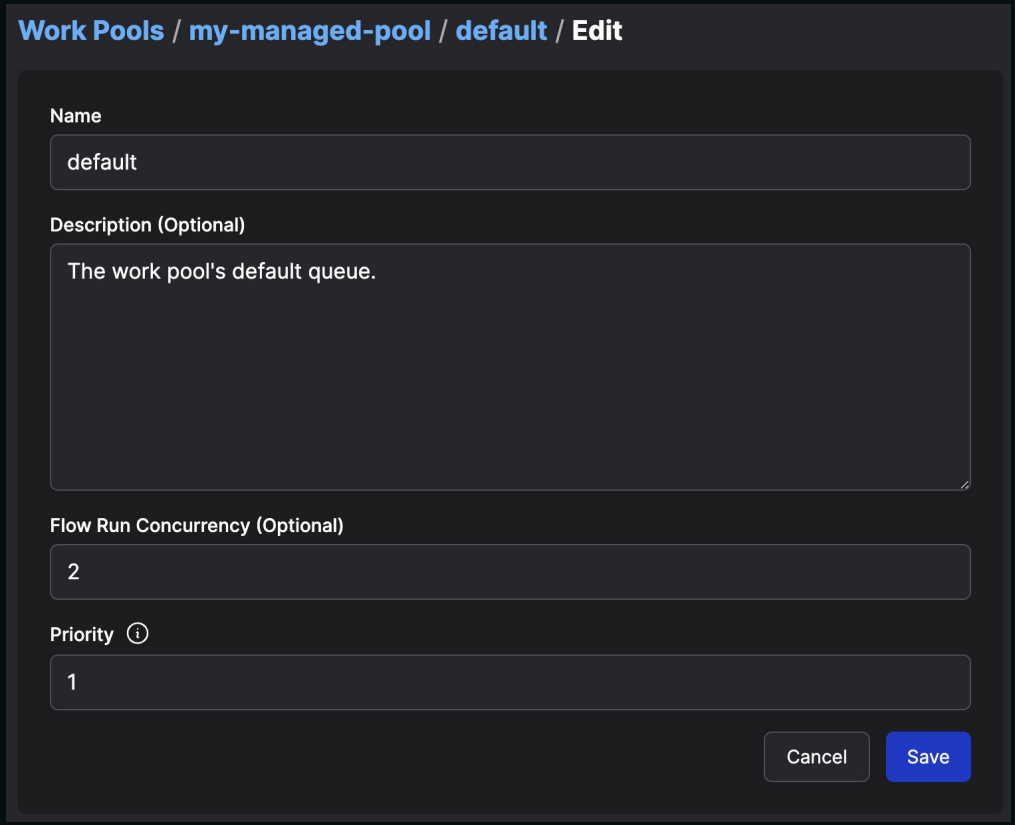
1 Work Queue

Search

Name	Concurrency Limit	Priority	Status
default		1	Not Ready

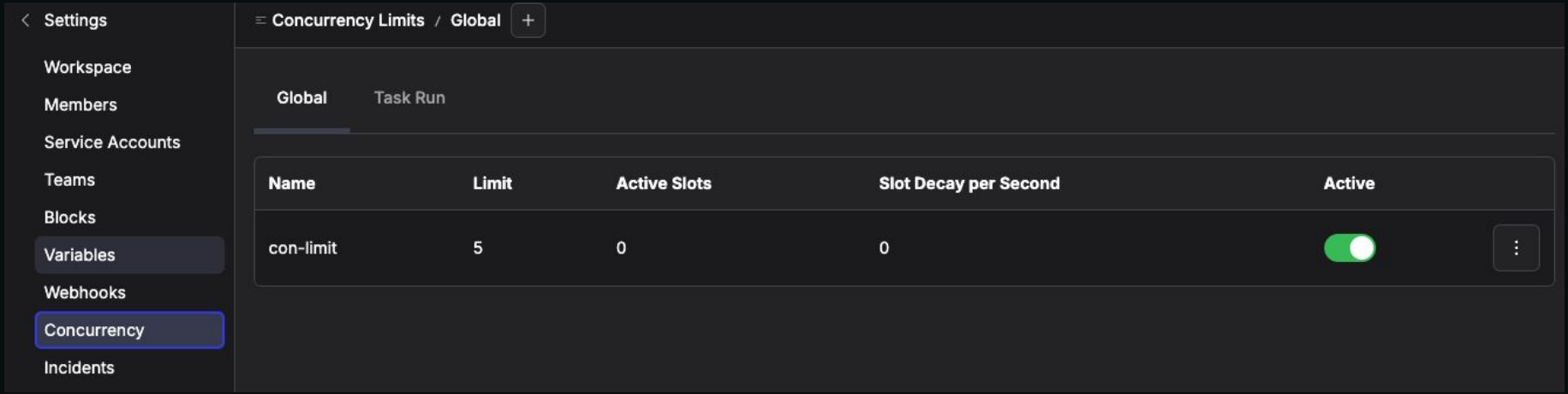
⚠️ *default* work queue created automatically

What’s a work queue for?



Other places to limit concurrency

- Globally (workspace level)
- Deployment
- Task run
- Work pool



Advanced triggers

Advanced triggers

- Composite triggers
- Metric triggers

Composite triggers

Composite trigger types

Compound trigger

- Event B
- Event A
- Event C

Sequential trigger

1. Event A
2. Event B
3. Event C

Composite triggers

An automation trigger made of more than one event

- **Compound:** any order
- **Sequential:** must occur in prescribed order

Optional: set a time period for events to occur

AND or OR

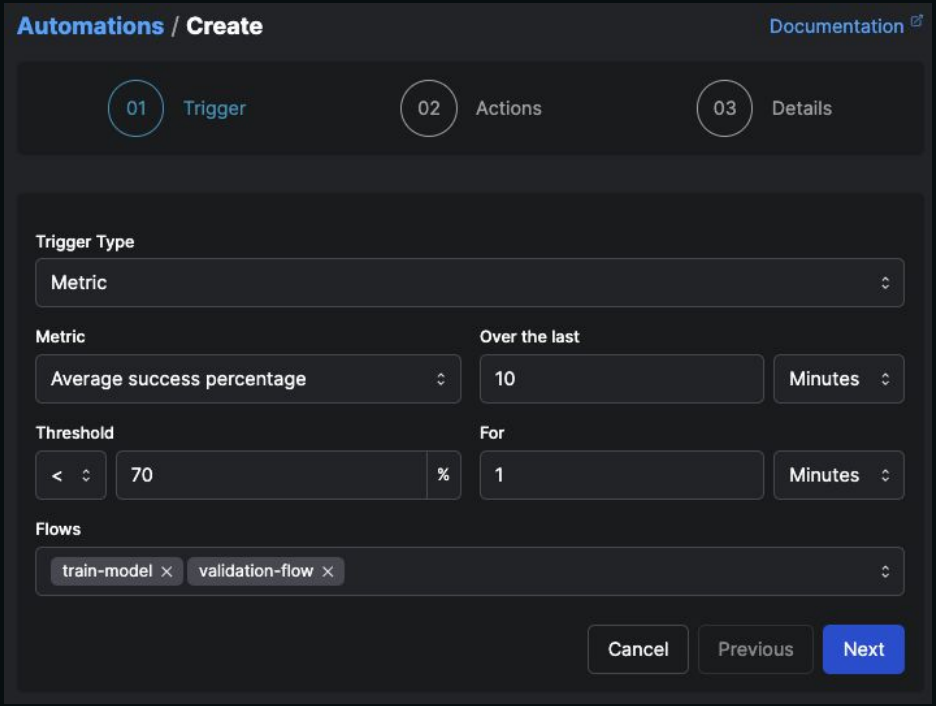
Composite triggers - example JSON

```
{
  "type": "compound",
  "require": "all",
  "within": 3600,
  "triggers": [
    {
      "type": "event",
      "posture": "Reactive",
      "expect": ["prefect.block.remote-file-system.write_path.called"],
      "match_related": {
        "prefect.resource.name": "daily-customer-export",
        "prefect.resource.role": "flow"
      }
    },
    {
      "type": "event",
      "posture": "Reactive",
      "expect": ["prefect.block.remote-file-system.write_path.called"],
      "match_related": {
        "prefect.resource.name": "daily-revenue-export",
        "prefect.resource.role": "flow"
      }
    },
```

Metric triggers

Metric triggers

Create an automation that uses a metric as a trigger



Metric triggers

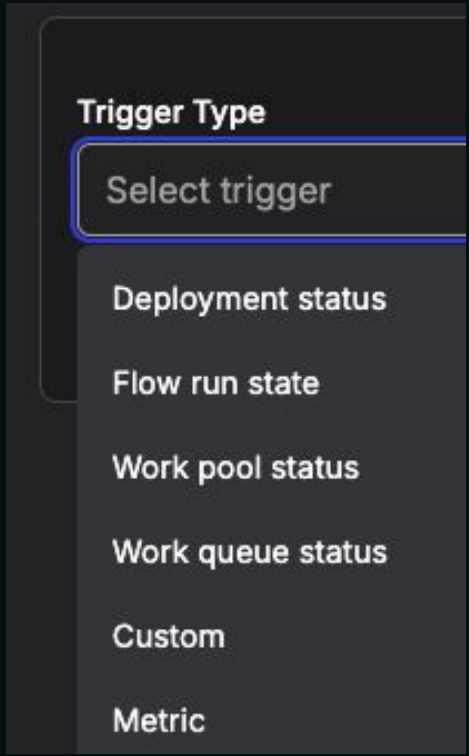
When a pattern is detected, then take an action.

Examples:

- Send a notification
- Toggle on a work pool
- Run a deployment
- Resume a flow run

Other trigger types

- Can use status of many Prefect objects as trigger



Testing

Testing

- Context manager for unit tests provided
- Run flows against temporary local SQLite DB database

```
from prefect import flow
from prefect.testing.utilities import prefect_test_harness

@flow
def my_favorite_flow():
    return 42

def test_my_favorite_flow():
    """basic test running the flow against a temporary testing database"""
    with prefect_test_harness():
        assert my_favorite_flow() == 42
```

Testing

Use in a Pytest fixture

```
from prefect import flow
import pytest
from prefect.testing.utilities import prefect_test_harness


@pytest.fixture(autouse=True, scope="session")
def prefect_test_fixture():
    with prefect_test_harness():
        yield
```

106 Recap

You've seen how to:

- Pause a flow run for human input
- Prioritize and limit work
- Create automations with compound and metric triggers
- Run tasks concurrently and in parallel
- Test workflows

Lab 106

106 Lab

- Create an interactive workflow that pauses a flow run for input from a user.
- Stretch 1: Run tasks concurrently with *ThreadPoolTaskRunner*
- Stretch 2: Use a compound trigger in an automation.
- Stretch 3: Use a metric trigger in an automation.