PACC

Prefect Associate Certification Course

PREFECT

Norms reminder

Zoom

- Camera on
- Mute unless asking a question
- Use hand raise in Zoom to ask a question

Slack

- Use threads
- Emoji responses 🙂

104 - Work pool-based deployments

104 Agenda

- Create work pool-based deployments with *.deploy()*
- Flow code storage
- Prefect managed work pools
- Hybrid work pools with workers
- Push work pools

Why use a work pool-based deployment?

Infrastructure is a pain, Prefect makes it better. 🙂

- Run a deployment on a variety of infrastructure
- Provide a template for deployments
- Ability to prioritize work
- Options to scale infrastructure to 0 (serverless)

Create deployment with *.deploy()*

Very similar syntax to *.serve()*

Differences:

- need to specify a work pool
- doesn’t start a server

First work pool-based deployment

- create with .deploy()
- specify flow code stored in a GitHub repository
- specify an existing *Prefect Managed* workpool

Create deployment with *.deploy()*

```python
from prefect import flow


if __name__ == "__main__":
    flow.from_source(
        source="https://github.com/discdiver/pacc-2024.git",
        entrypoint="102/weather2-tasks.py:pipeline",
    ).deploy(
        name="my-first-managed-deployment",
        work_pool_name="managed1",
    )
```


Create deployment with *.deploy()*

Run the script

```
Successfully created/updated all deployments!

                    Deployments
```

Name	Status	Details
pipeline/my-first-managed-deployment	applied	

```
To schedule a run for this deployment, use the following command:

        $ prefect deployment run 'pipeline/my-first-managed-deployment'


You can also run your flow via the Prefect UI:
https://app.prefect.cloud/account/9b649228-0419-40e1-9e0d-44954b5c0ab6/workspace/d137367a-5055-44ff-b91c-6f7366c9e4c4/deployments/deployment/d448be8f-2092-47f9-8d0b-ee06ce182480
```

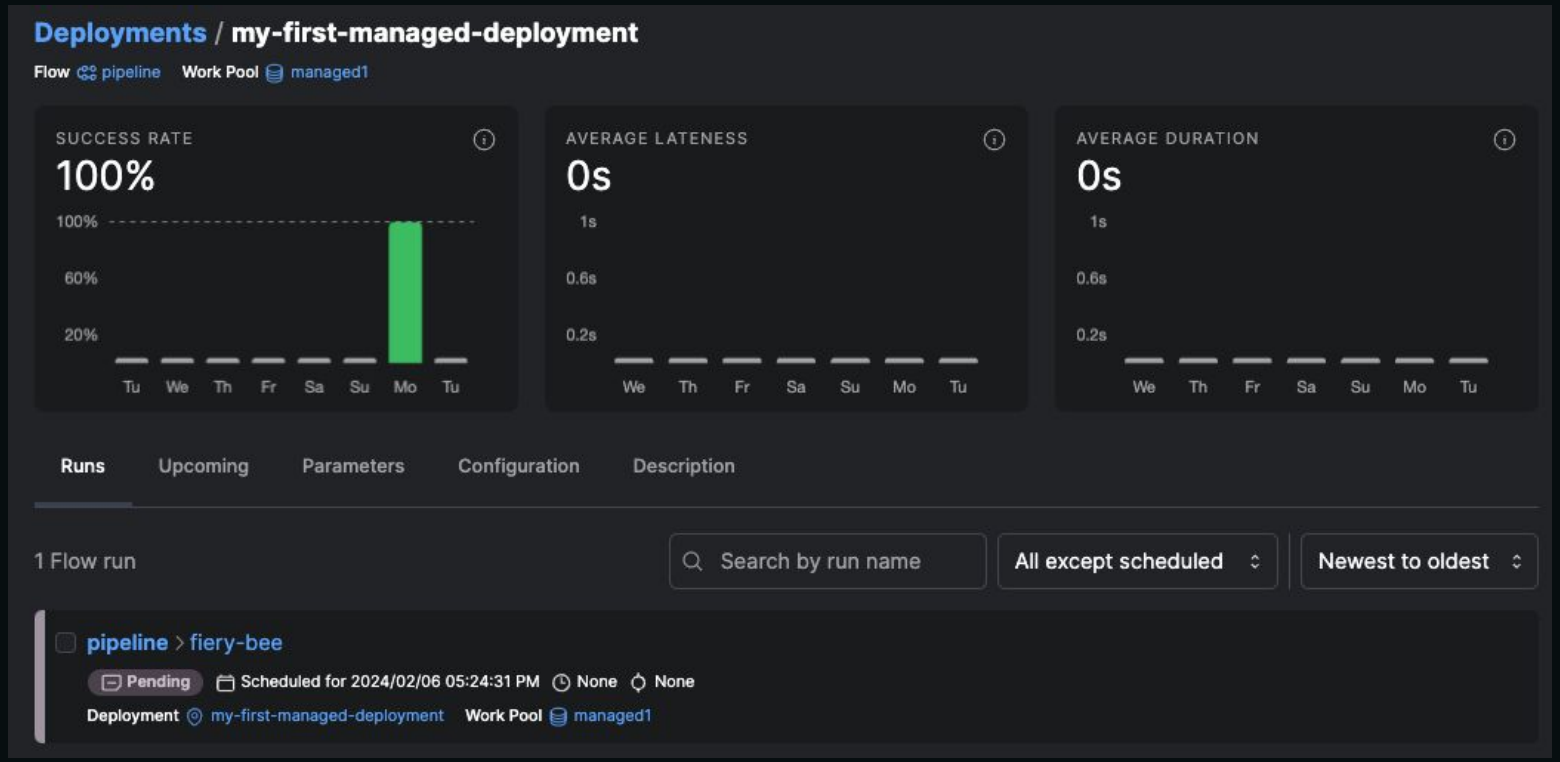

Create a deployment with *.deploy()*

Run the deployment from the UI or the CLI:

prefect deployment run 'pipeline/my-first-managed-deployment'

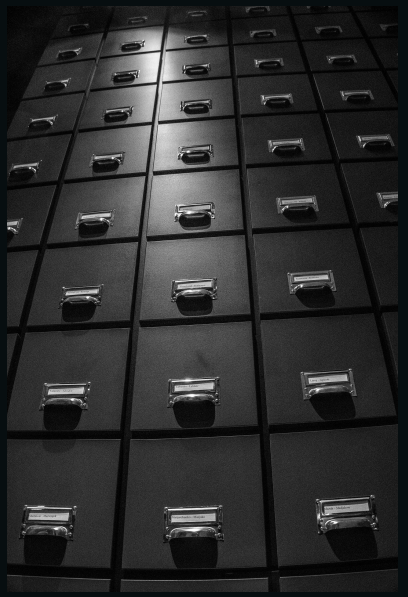
Takes a moment to start infra and pull base Docker image

See the deployment and flow run in the UI



Let’s break this down

Flow code storage



Flow code storage options

1. Git-based remote repository (e.g. GitHub)
2. Bake your code into a Docker image
3. Cloud provider storage

We specified a public GitHub repo with *.from_source()* class method.

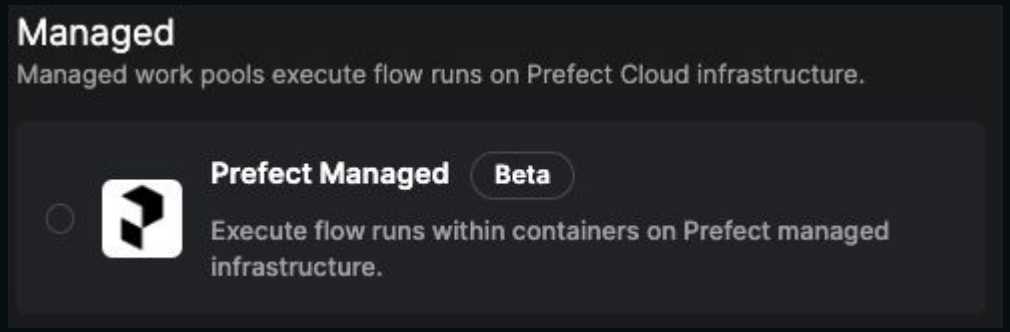
Provide the source URL to the repo and the entrypoint path:flow function name.

Work pools

Work pools

Provide default infrastructure configuration for deployments

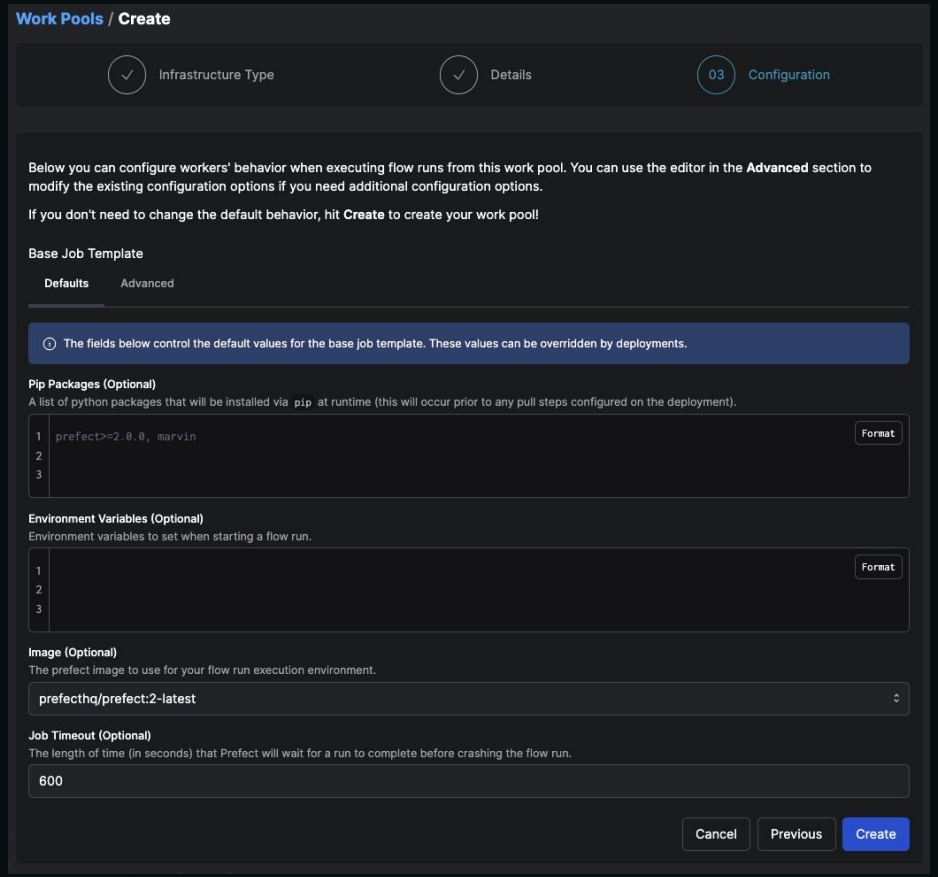
Create a work pool of type **Prefect Managed**



With a **Prefect Managed** pool, Prefect runs your flow code on our infrastructure in a Docker container.

👆Only available with Prefect Cloud

Create a Prefect Managed work pool



Create a Prefect Managed work pool

- Don’t modify the job template for now
- You can specify environment variables, packages to install at runtime, etc.
- All deployments that use this work pool inherit these settings

At runtime, Prefect:

1. Pulls the Docker image specified
2. Installs any specified packages
3. Pulls your flow code from GitHub
4. Runs your code in the container
5. Monitors state
6. Exits and cleans up 🧹

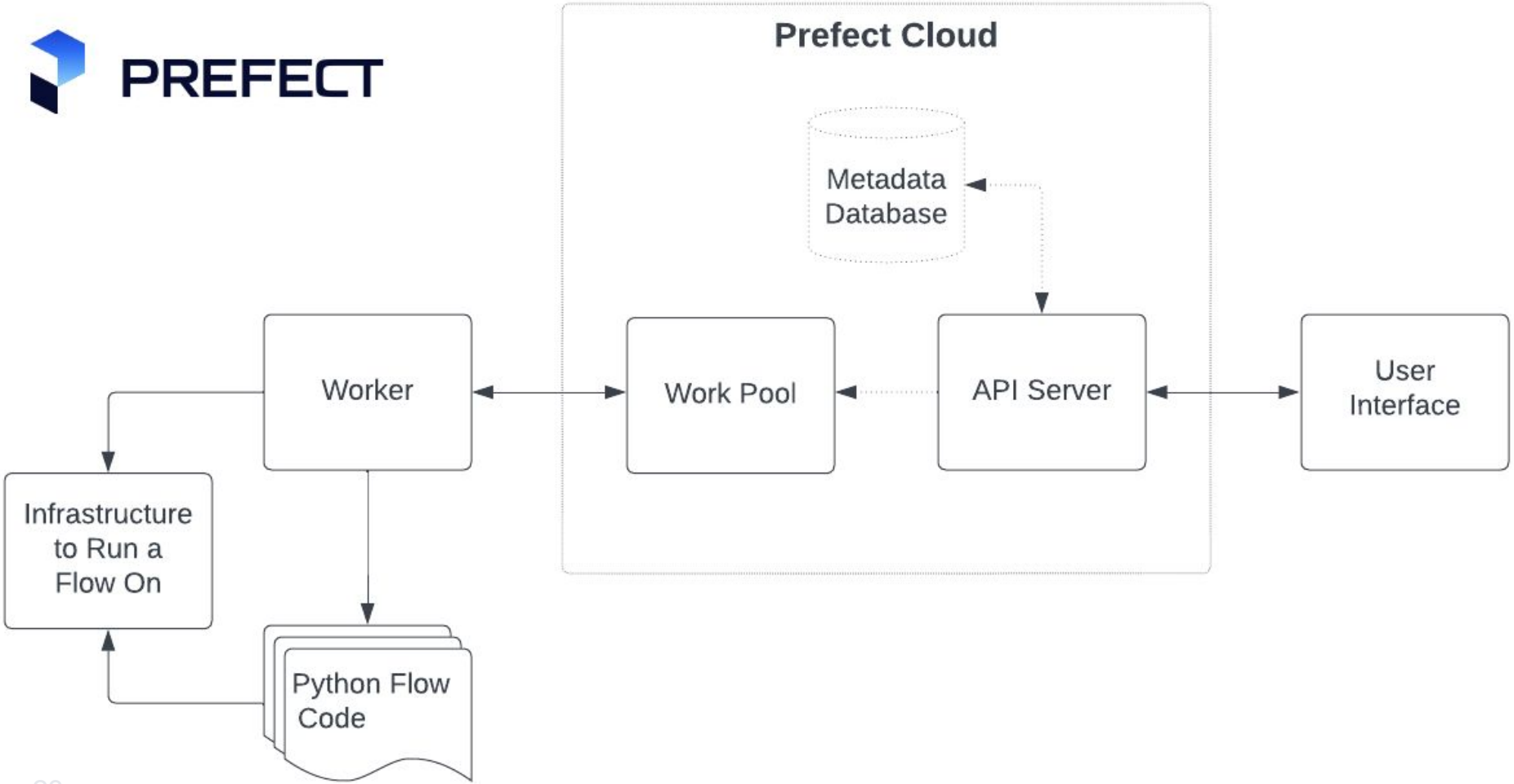
Hybrid model - hybrid work pools with workers

Hybrid model = separation

- Your flow code runs on your infrastructure
- Your flow code is stored on your storage (GitHub, AWS, Docker image, etc)
- Prefect Cloud stores metadata, logs, artifacts, etc.
- Data encrypted at rest
- Prefect Technologies, Inc. is SOC2 Type II compliant

https://www.prefect.io/security

PREFECT

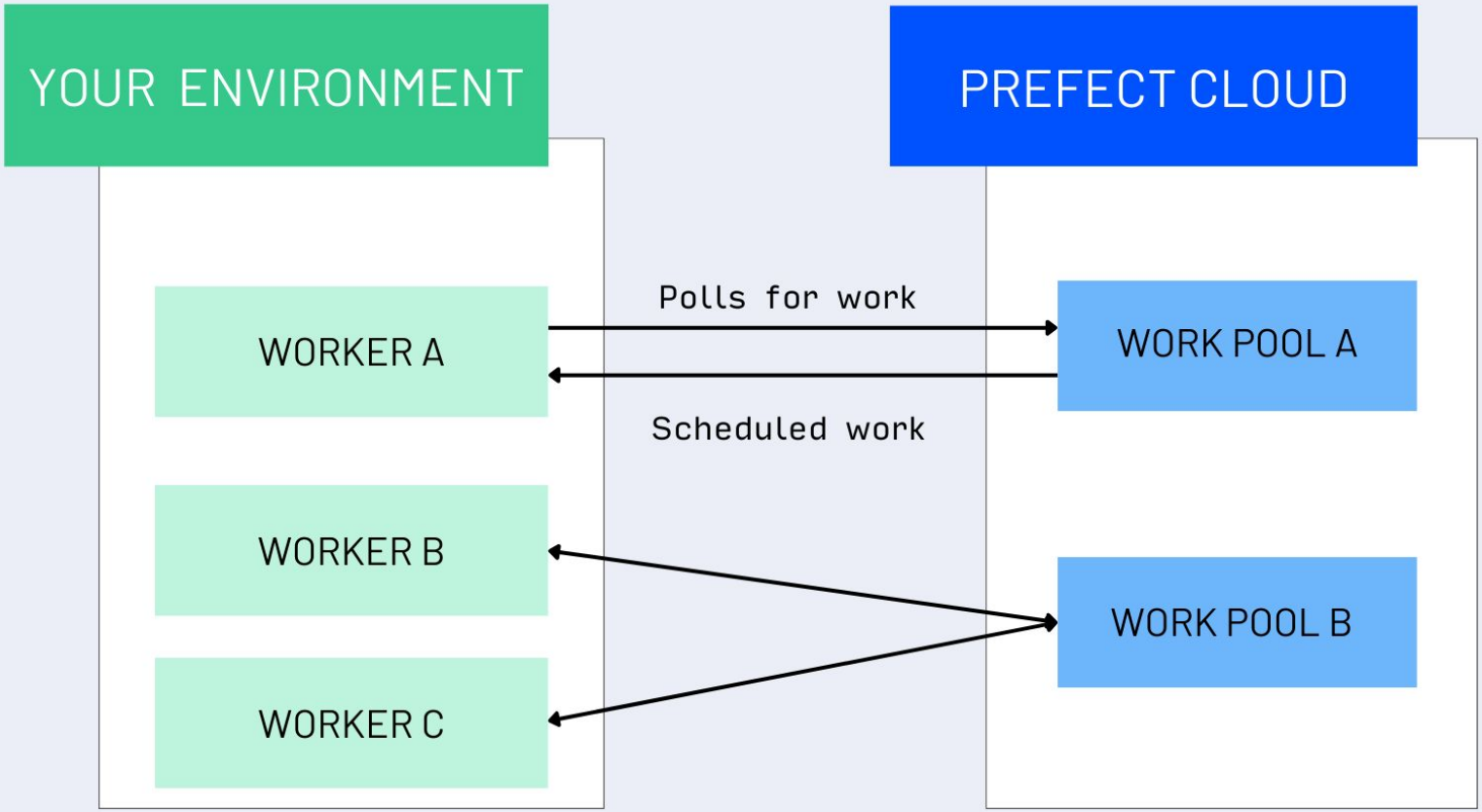

Prefect Cloud
Metadata Database
Worker
Work Pool
API Server
User Interface
Infrastructure to Run a Flow On
Python Flow Code

Workers

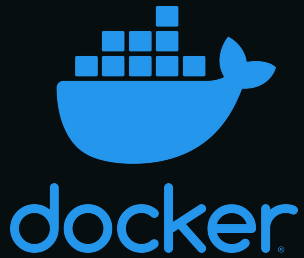
Workers

- Long-running process on the client
- Poll for scheduled flow runs from work pools
- Must match a work pool to pick up work

WORKERS & WORK POOLS



Docker work pool & worker



Why use Docker?

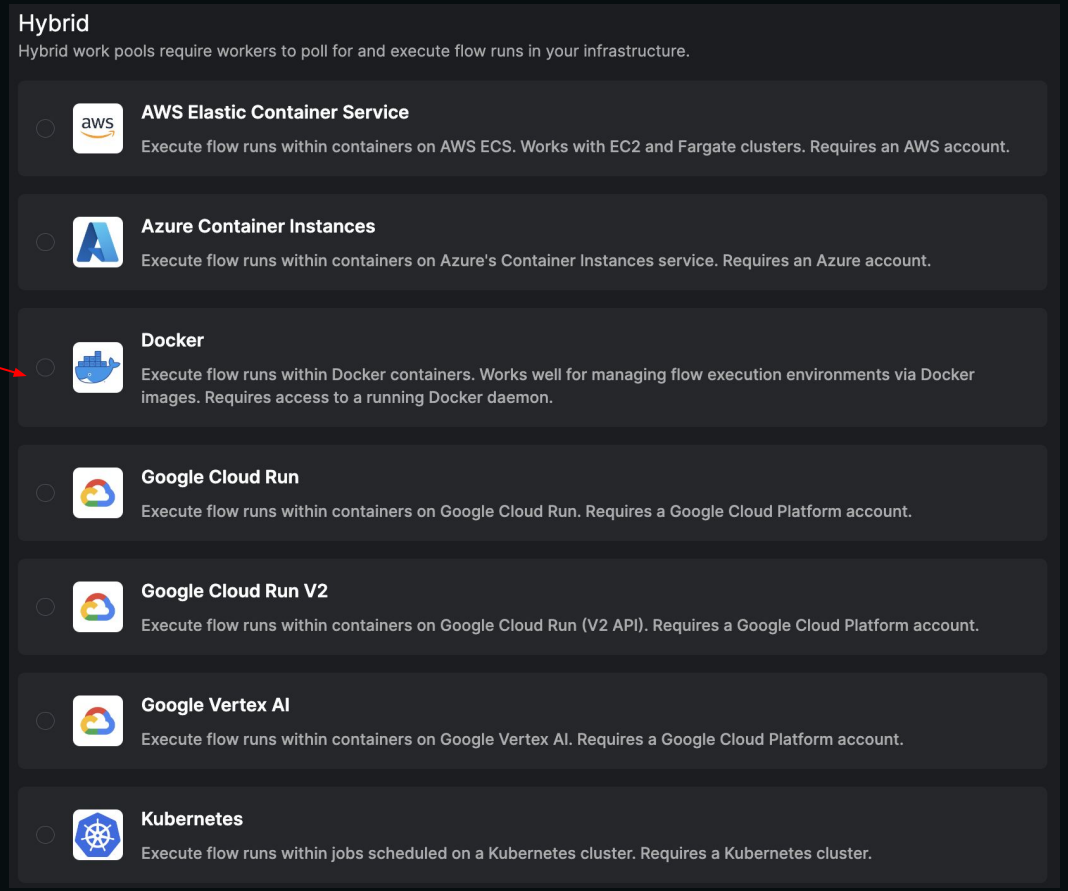
- Same operating environment everywhere
- Lighter weight than a VM
- Linux (generally)
- Portable
- Very popular
- Almost all Prefect work pools use it

Docker work pool

Run a flow in a Docker container

1. Install: *pip install -U prefect-docker*
2. Start Docker on your machine
3. Create a Docker type work pool
4. Start a worker that polls the work pool
5. Create a deployment that specifies the work pool
6. Run the deployment

Create a Docker work pool



Package flow code into a Docker image with *.deploy()*

```python
from prefect import flow


@flow(log_prints=True)
def buy():
    print("Buying securities")


if __name__ == "__main__":
    buy.deploy(
        name="my-code-in-an-image-deployment",
        work_pool_name="my-docker-pool",
        image="discdiver/local-image:1.0",
        push=False,
    )
```

.from_source() method not needed if baking flow code into image

.deploy() method

Creates a Docker image with your flow code baked in by default!

- specify the image name
- specify *push=False* to not push image to registry
- can create a *requirements.txt* file with packages to install into the image (or add package names to work pool or at deployment creation time)

Docker type worker

Start a Docker type worker to connect to a work pool named my-*docker-pool*

prefect worker start -p my-docker-pool

Dockerfile used to create your image (under the hood)

```
FROM prefecthq/prefect:2-latest
COPY requirements.txt /opt/prefect/104/requirements.txt
RUN python -m pip install -r requirements.txt
COPY . /opt/prefect/pacc-2024/
WORKDIR /opt/prefect/pacc-2024/
```


Docker

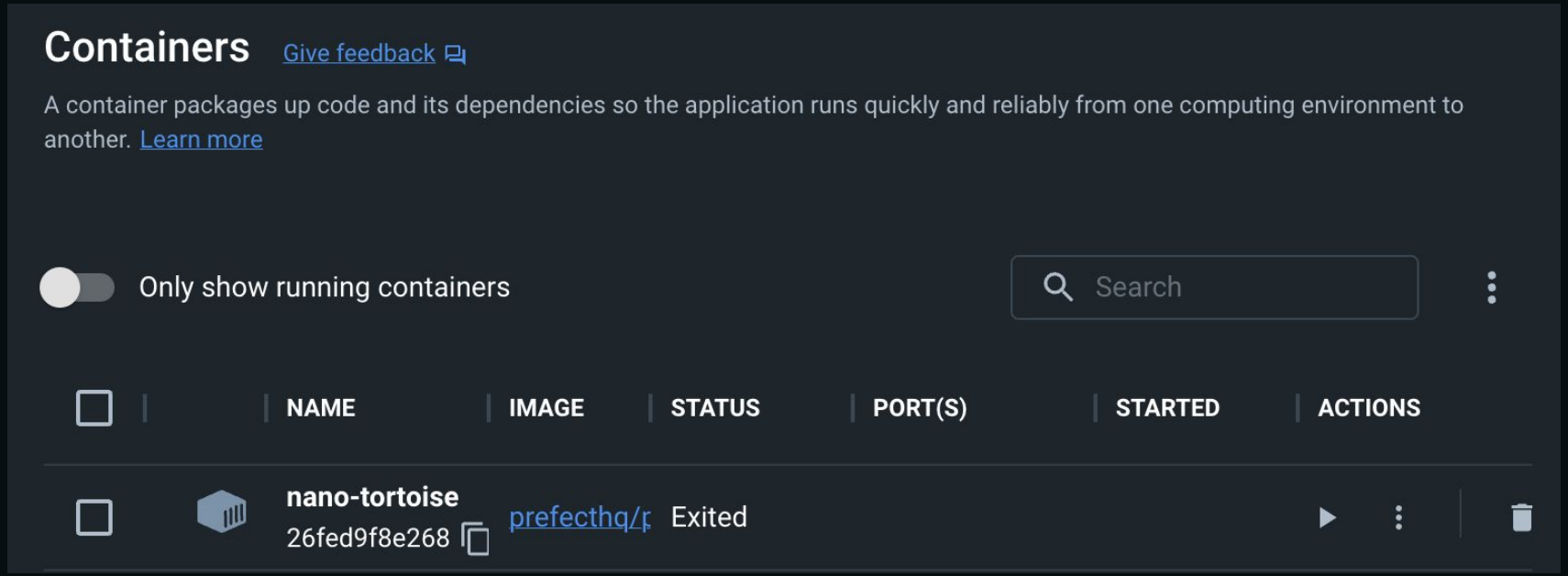
- Prefect provides base Docker images
- Can customize base image
- Read about choosing images at docs.prefect.io/concepts/infrastructure/#standard-python

Docker

- Run your deployment
- Worker pulls image and spins up Docker container
- Flow code runs in Docker container and exits 🚀

Docker

See container in Docker Desktop if running locally



Docker

Prerequisites reminder:

- Docker *installed* & **running**
- *prefect-docker* package installed

Hybrid work pool types

1. Kubernetes
2. Docker
3. Serverless options such as ECS, ACI, GCR, VertexAI
4. Process (local subprocess)

* Worker required for all

Push work pools

Push work pools

Serverless options with no worker required

Options:

- AWS ECS, Google Cloud Run, Azure Container Instances

Create from CLI:

prefect work-pool create --type modal:push --provision-infra my-modal-pool

Push work pools

Prefect will create everything for you with *--provision-infra*

Prerequisites to use:

- Cloud provider account
- CLI tool installed
- Authenticated locally

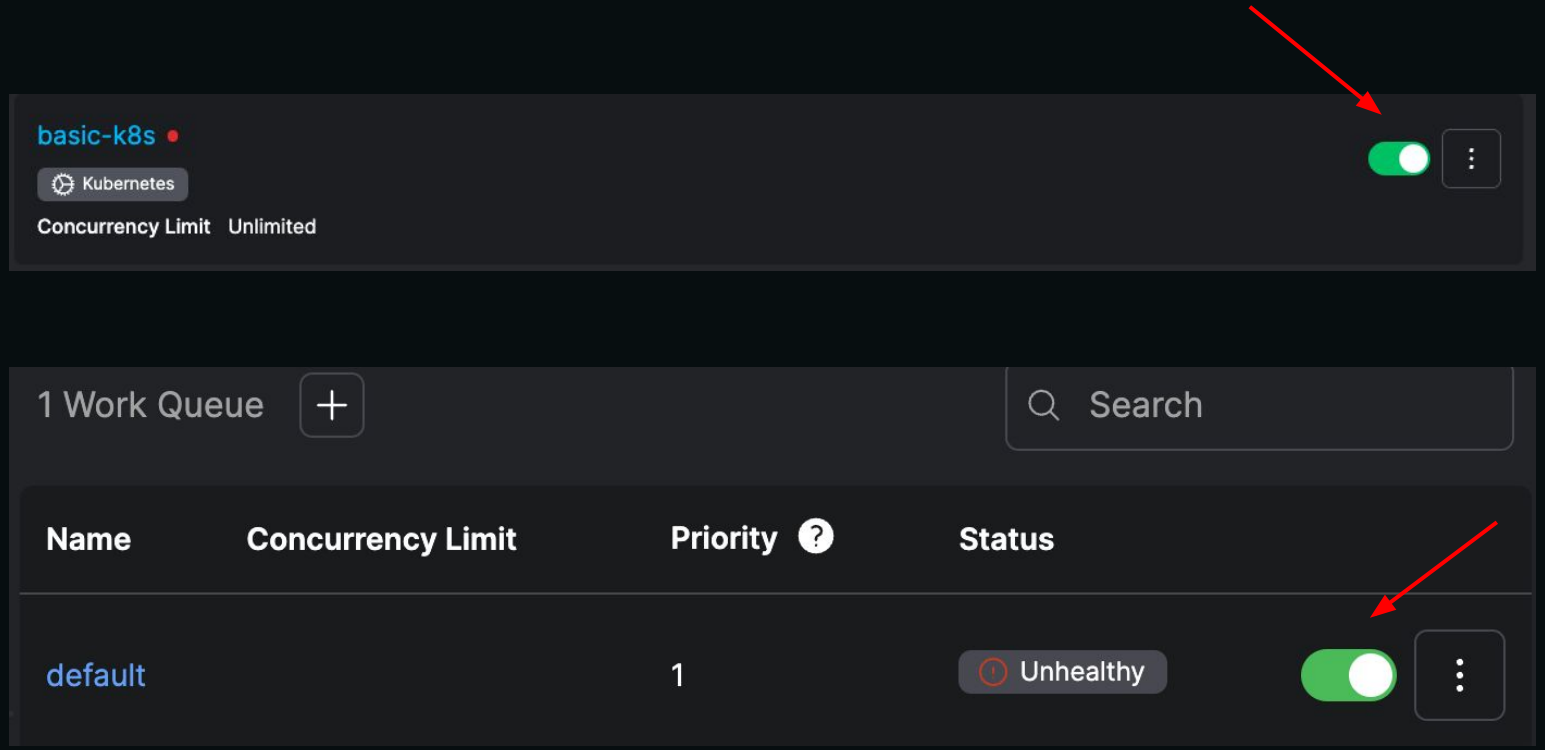
prefect work-pool create --type modal:push --provision-infra my-modal-pool

What’s a work queue for?

- Prioritize work
- Limit concurrent runs

⚠️ default work queue created automatically

Pause work pools or work queues



104 Recap

You've seen how to

- Create work-pool based deployments! 🎉
- Create a deployment that uses a Prefect managed work pool and flow code stored on GitHub
- Use the hybrid model with workers
- Bake flow code into Docker images
- Create push work pools with a single command
- Pause and resume work pools and work queues

Lab 104

Reminder: breakout room norms

1. 🙂 Introduce yourselves
2. 🎥 Camera on (if possible)
3. 💻 One person shares screen
4. 👩‍💻 Everyone codes
5. 🙋 Each person talks
6. 😌 Low-pressure, welcoming environment: lean in

Breakout rooms with lots of participation = more fun + more learning! 😎

104 Lab

- Create a Prefect Managed work pool.
- Create and run a deployment that uses the work pool.
- Use flow code stored in your own GitHub repository with a deployment.
- Pause and resume the work pool from the UI.

104 Lab Extensions

- Stretch 1: bake your flow code into a Docker image with *.deploy()*.
- Don’t push the image to a remote repository (or do log in and push it to DockerHub).

  Don’t forget to:

  - Start Docker on your machine
  - *pip install -U prefect-docker*
  - Make a Docker work pool
  - Start a Docker type worker that polls the pool
- Stretch 2: create a push work pool with *provision-infra* and use it in a deployment.
- Stretch 3: add an environment variable to a work pool and use it.

Lab 104: a solution

One person from each group, share your code in Slack 🧵

Discuss

Questions?

If you give an engineer a job…

Could you just fetch this data and save it? Oh, and …

1. set up logging?
2. do it every hour?
3. visualize the dependencies?
4. automatically retry if it fails?
5. create an artifact for human viewing?
6. add caching?
7. add collaborators to run and view - who don't code?
8. send me a message when it succeeds?
9. run it in a Docker container-based environment?
10. pause for input?
11. automatically declare an incident when a % of workflows fail?
12. automatically run a workflow in response to an event?

105 - Interactive workflows & incidents

105 Agenda

- Interactive workflows
  - Human in the loop
- Incidents
- Metric triggers
- Prefect Runtime
- State change hooks

Interactive workflows

Interactive workflows

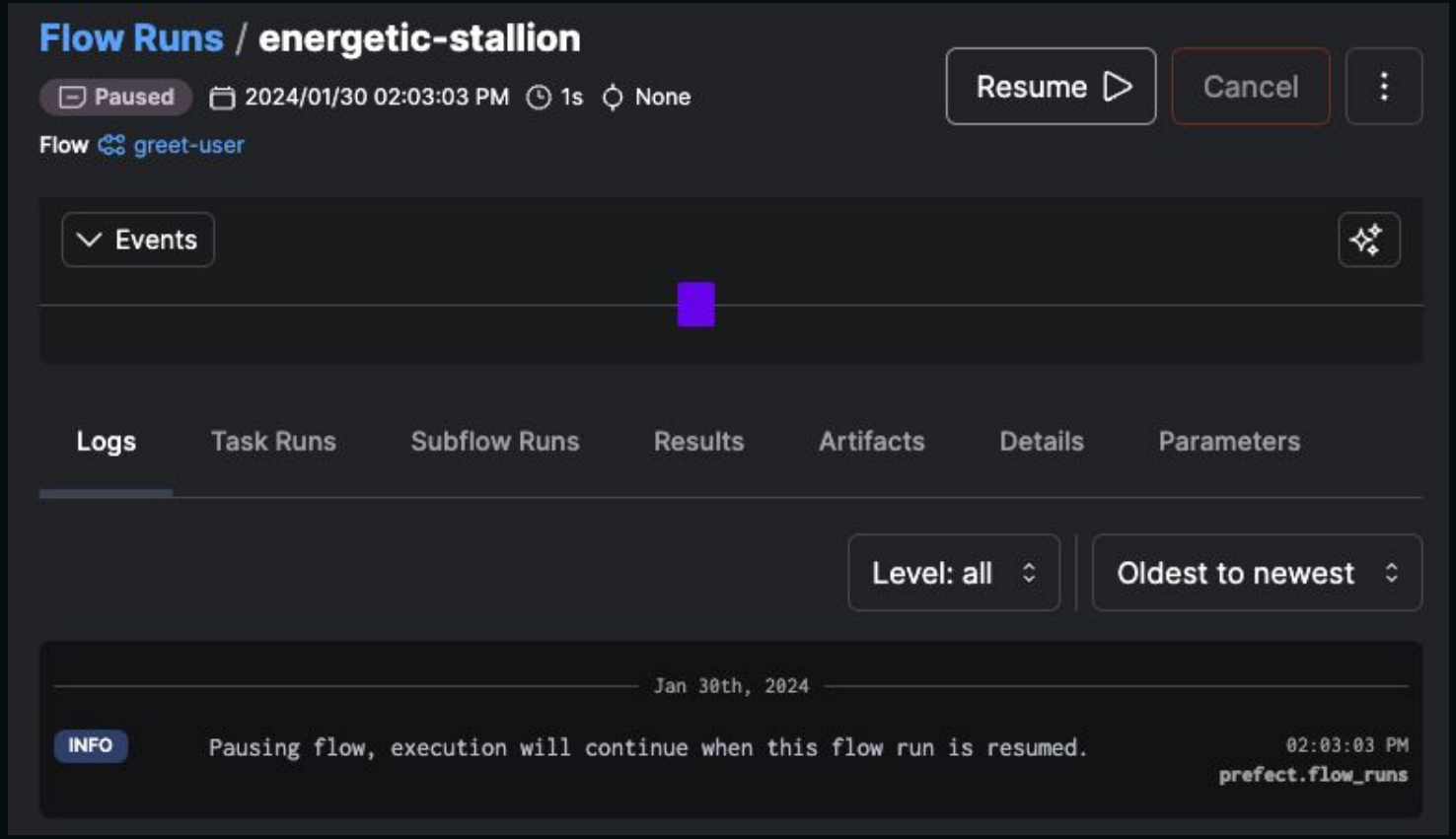
Pause a flow run to wait for input from a user via a web form (human-in-the-loop) 🙂

pause_flow_run function

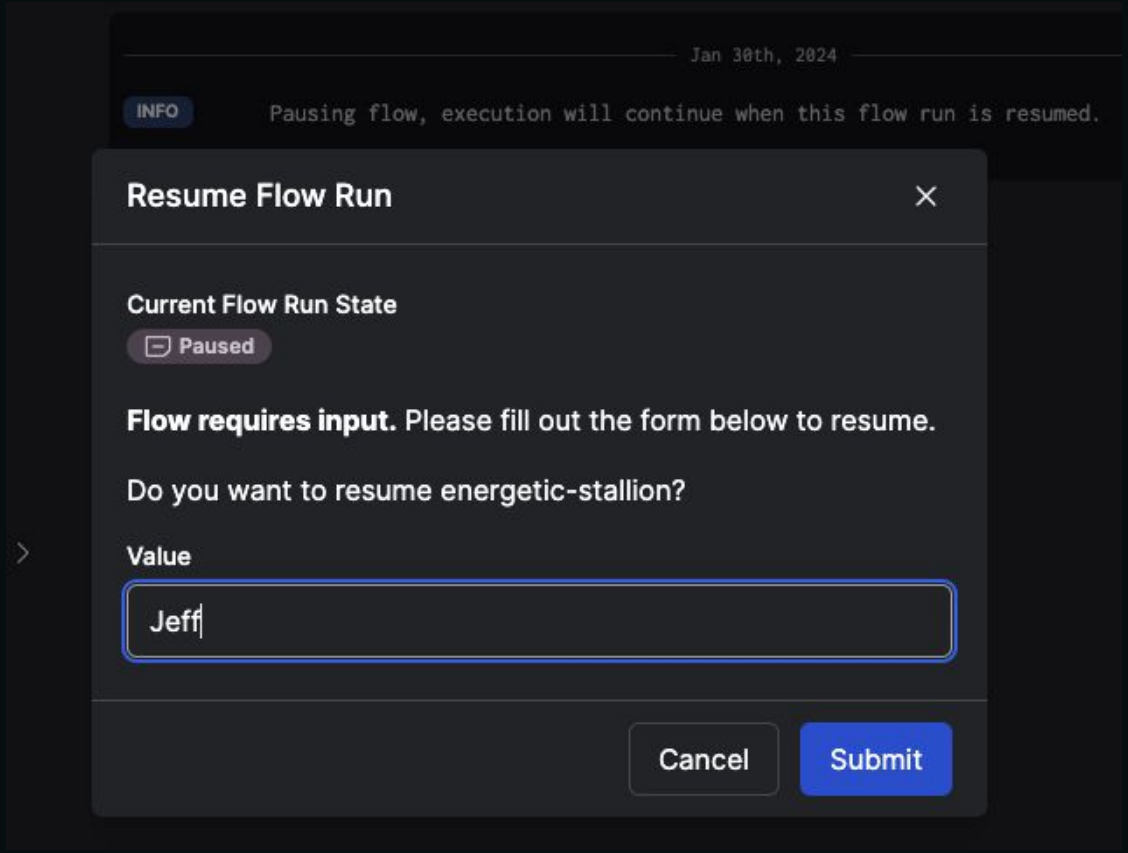
Human-in-the-loop: basic

```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()
```

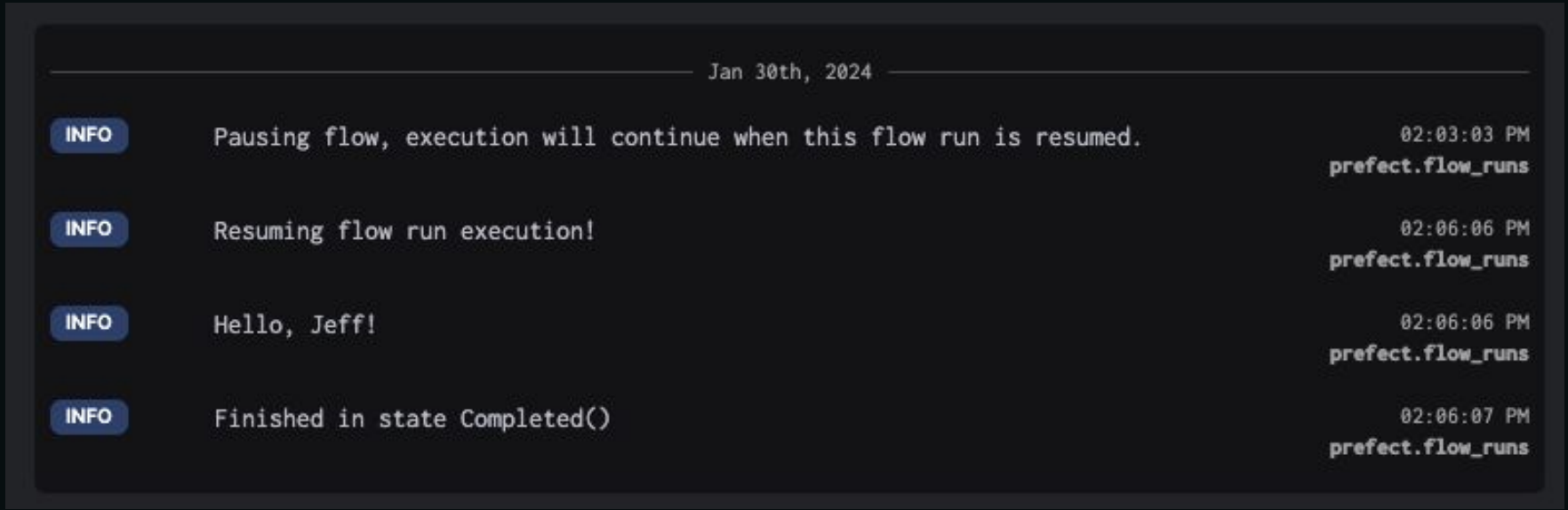

Human-in-the-loop: basic



Human-in-the-loop: basic



Human-in-the-loop: basic

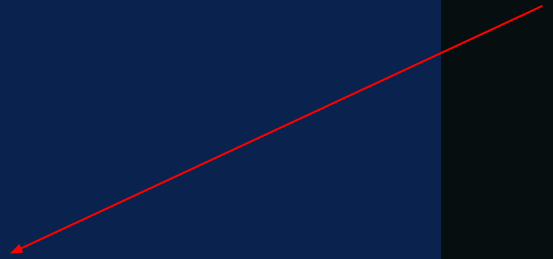


Interctive workflows

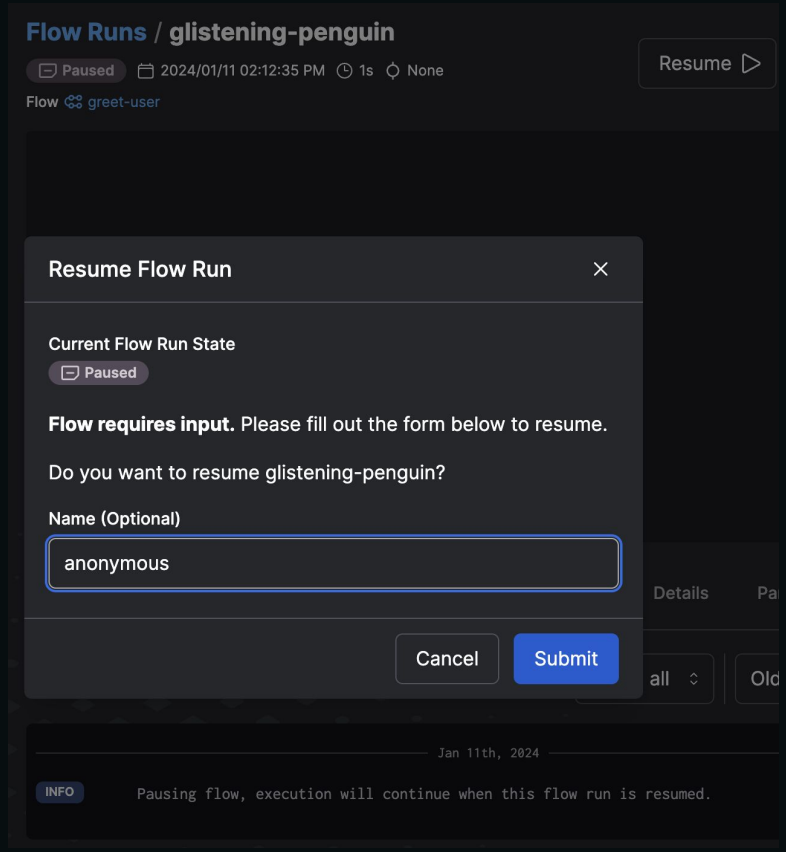
- For validation: can use *RunInput* class, which is a subclass of Pydantic's *BaseModel* class
- Able to specify a default value or create a dropdown
- Can create a default value

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



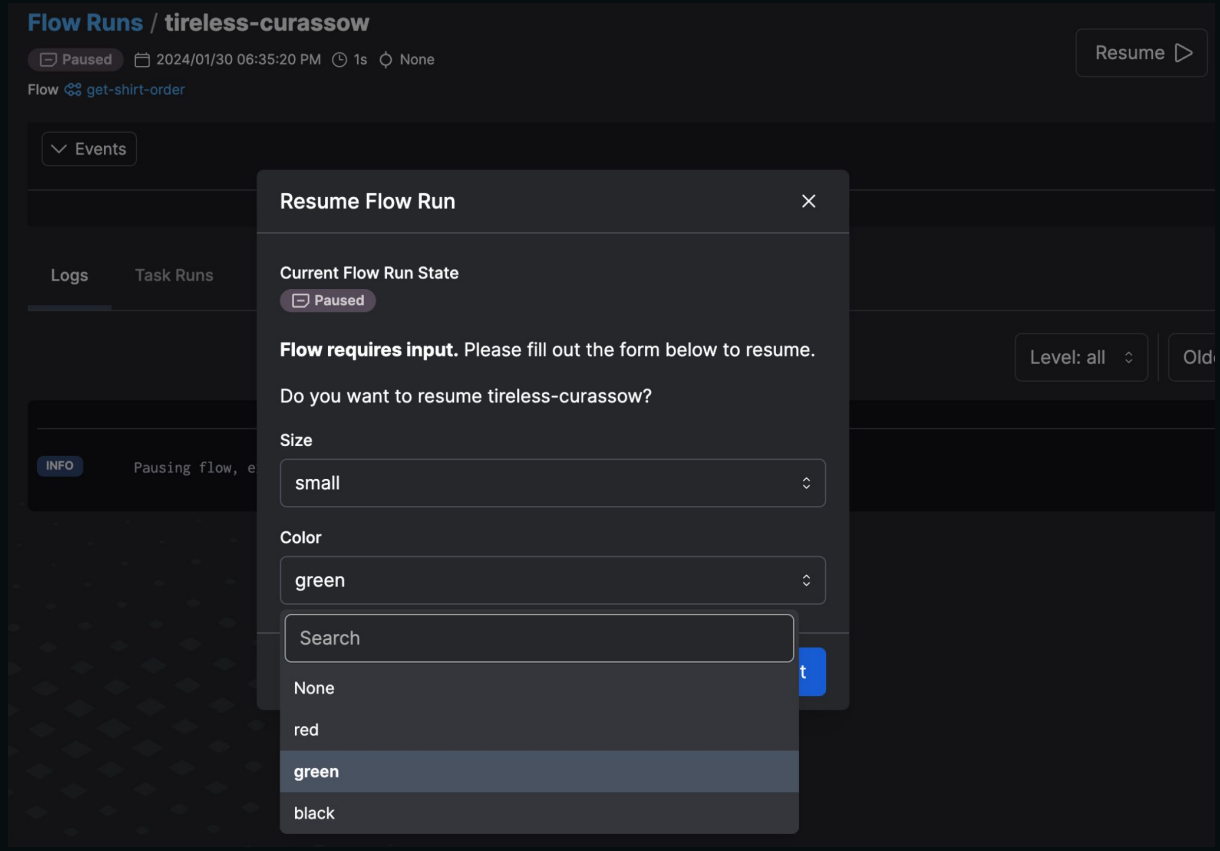
Human-in-the-loop: custom validation

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


class ShirtOrder(RunInput):
    """Shirt order options"""

    size: Literal["small", "medium", "large", "xlarge"]
    color: Literal["red", "green", "black"]

    @pydantic.validator("color")
    def validate_shirt(cls, value, values, **kwargs):
        """Validate that shirt combo exists"""

        if value == "green" and values["size"] == "small":
            raise ValueError("We don't carry that combination.")
        return value
```


Human-in-the-loop: custom validation

```python
@flow(log_prints=True)
def get_shirt_order():
    """Get shirt selection from user via UI"""
    shirt_order = None

    while shirt_order is None:
        try:
            shirt_order = pause_flow_run(wait_for_input=ShirtOrder)
            print(f"We'll send you your shirt in {shirt_order} ASAP!")
        except pydantic.ValidationError:
            print(f"Invalid size and color combination.")


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


Human-in-the-loop: custom validation



Human-in-the-loop: custom validation

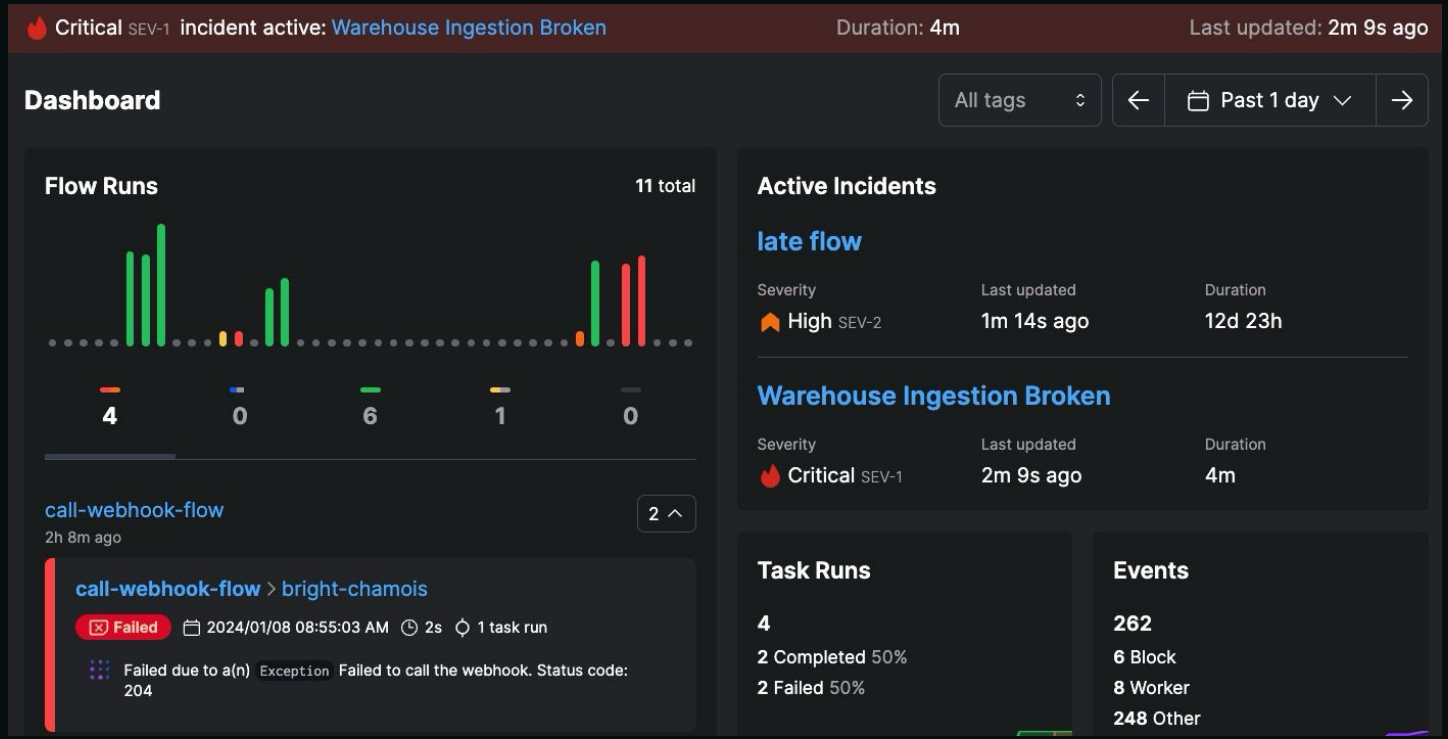
Jan 30th, 2024

```
INFO    Pausing flow, execution will continue when this flow run is resumed.

INFO    Resuming flow run execution!

INFO    Invalid size and color combination.

INFO    Pausing flow, execution will continue when this flow run is resumed.

INFO    Resuming flow run execution!

INFO    We'll send you your shirt in size='medium' color='red' ASAP!

INFO    Finished in state Completed()
```


Incidents 

Incidents

Formal declarations of disruptions to a workspace

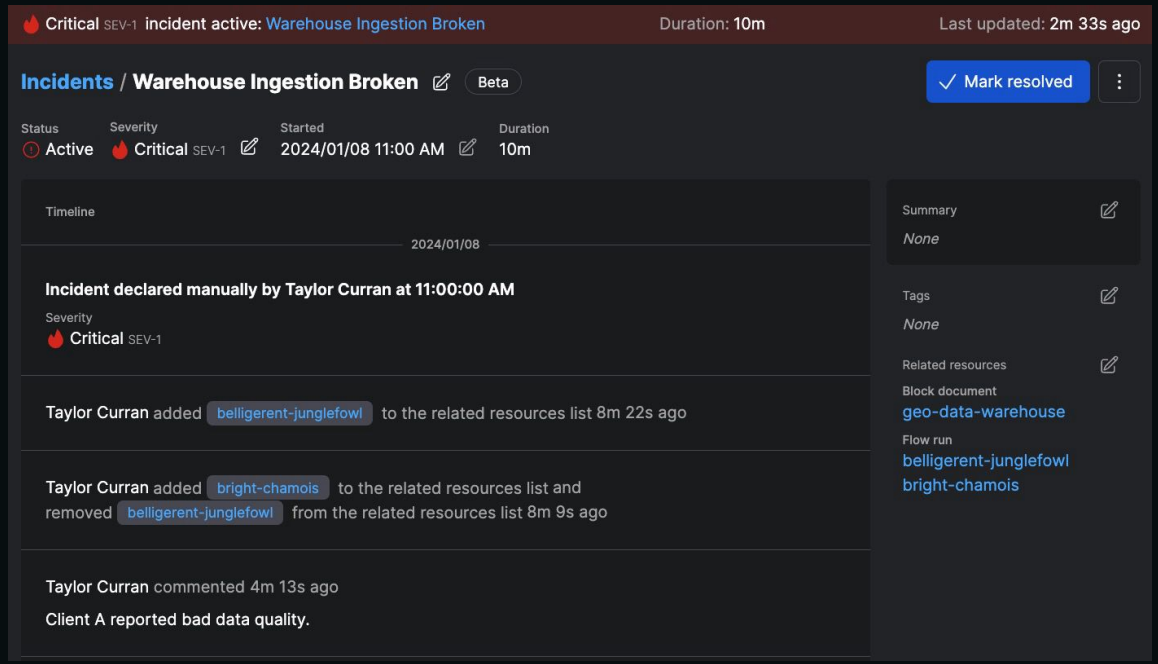


Incidents

- visible workspace-wide
- keeps team updated for faster resolution
- creates record for analysis and compliance
- Pro and Enterprise level feature

Incidents

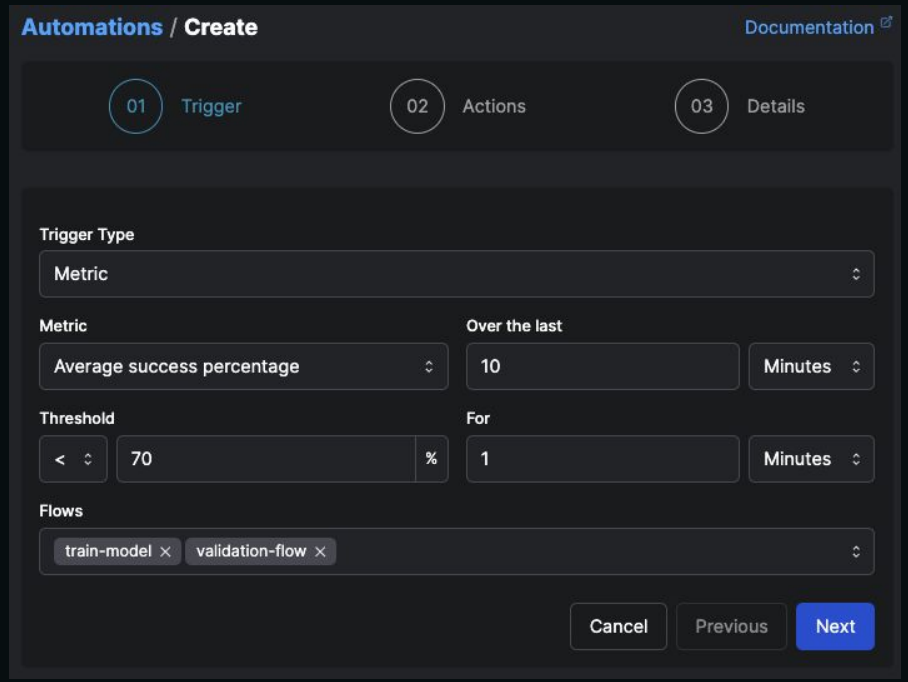
Declare an incident manually or automatically through an automation when an event occurs



Metric triggers

Metric triggers

Create an automation that uses a metric as a trigger



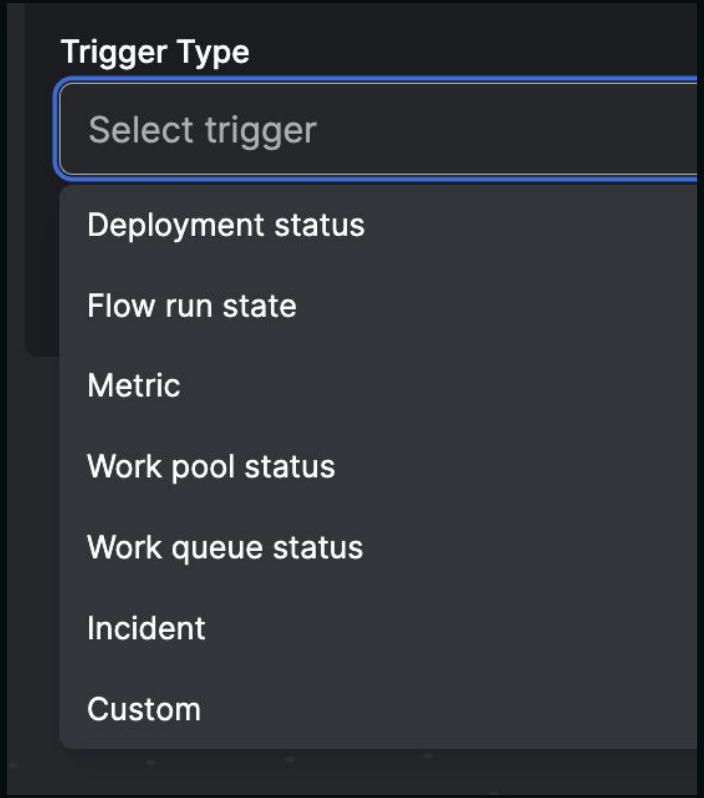
Metric triggers

When a pattern is detected, then take an action

- send a notification
- toggle on a work pool
- create an incident
- run a deployment

Other trigger types

- Can use status of many Prefect objects as triggers
- Incidents can act as a trigger



prefect.runtime

prefect.runtime module

Home for runtime context access.

Useful for labeling, logs, etc.

Includes:

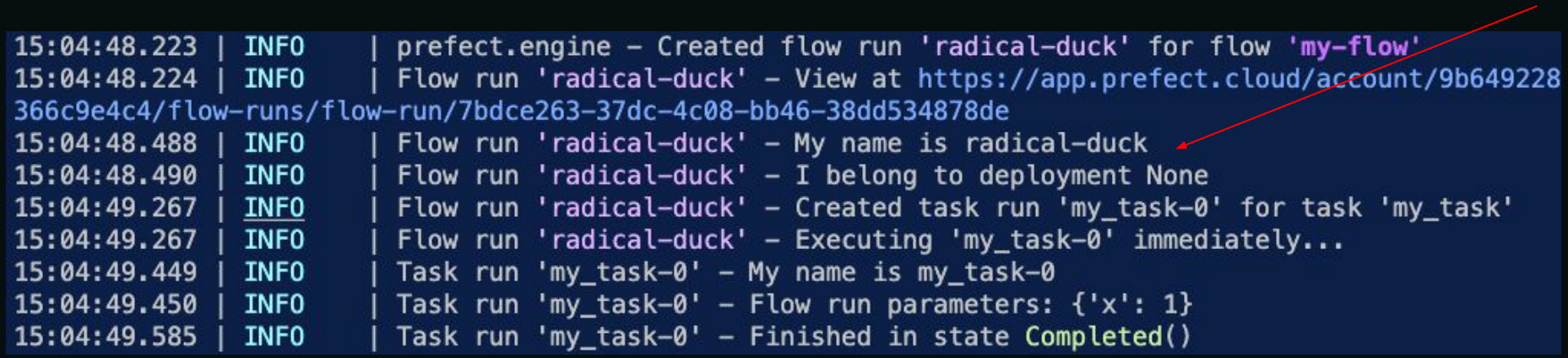
- ***deployment:*** info on the current deployment
- ***flow_run***: info on the current flow run
- ***task_run***: info on the current task run

prefect.runtime

```python
from prefect import flow, task
from prefect import runtime


@flow(log_prints=True)
def my_flow(x):
    print("My name is", runtime.flow_run.name)
    print("I belong to deployment", runtime.deployment.name)
    my_task(2)


@task
def my_task(y):
    print("My name is", runtime.task_run.name)
    print("Flow run parameters:", runtime.flow_run.parameters)


if __name__ == "__main__":
    my_flow(x=1)
```


prefect.runtime

Useful for labeling, logs, etc.



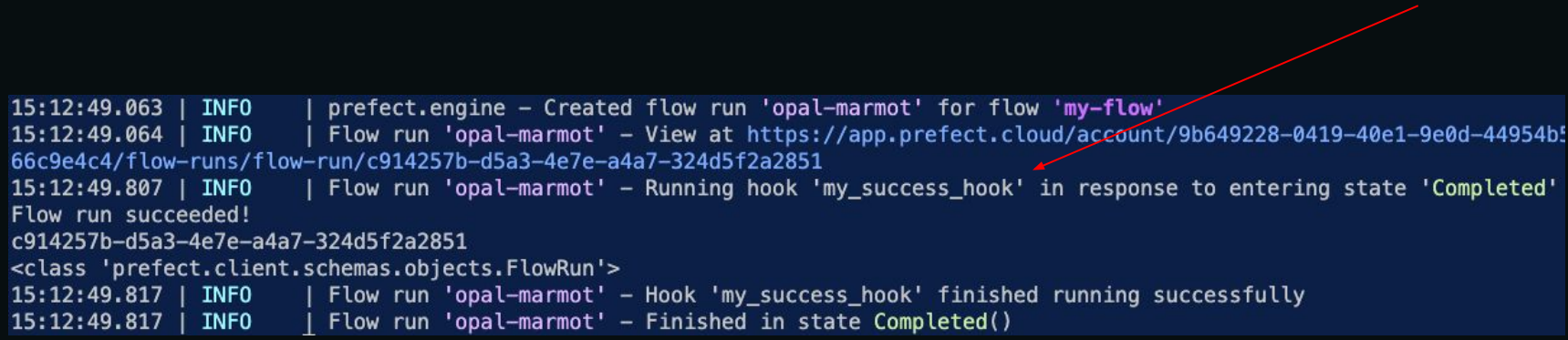
State change hooks

State change hooks

Execute code in response to flow run or task run state changes

```python
from prefect import flow


def my_success_hook(flow, flow_run, state):
    print(f"Flow run {flow_run.id} succeeded!")


@flow(on_completion=[my_success_hook])
def my_flow():
    return 42


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


State change hooks



State change hooks

Type	Flow	Task	Description
`on_completion`	✓	✓	Executes when a flow or task run enters a `Completed` state.
`on_failure`	✓	✓	Executes when a flow or task run enters a `Failed` state.
`on_cancellation`	✓	-	Executes when a flow run enters a `Cancelling` state.
`on_crashed`	✓	-	Executes when a flow run enters a `Crashed` state.

105 Recap

You've seen how to:

- Create an active workflow that pauses a flow run for input from a user
- Use a metric trigger in an automation
- Get current info into a flow with *prefect_runtime*
- Use a state change hook

Lab 105

105 Lab

- Create an active workflow that pauses a flow run for input from a user.
- Print the flow run name in your code with *prefect_runtime*
- Use a state change hook to run code when a flow run state is reached.
- Stretch (if on Pro or Enterprise plan): Use a metric trigger in an automation.

Lab 105: a solution

One person from each group, share your code in Slack 🧵

Discuss

Questions?

If you give an engineer a job…

Could you just fetch this data and save it? Oh, and …

1. set up logging?
2. do it every hour?
3. visualize the dependencies?
4. automatically retry if it fails?
5. create an artifact for human viewing?
6. add caching?
7. add collaborators to run and view - who don't code?
8. send me a message when it succeeds?
9. run it in a Docker container-based environment?
10. pause for input?
11. automatically declare an incident when a % of workflows fail?
12. automatically run a workflow in response to an event?

106 - Workflow patterns & event-based workflows

106 Agenda

Workflow patterns with

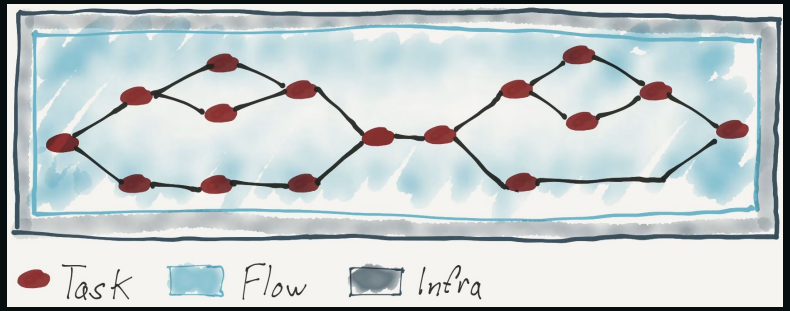
- subflows
- *run_deployment*
- automations

Automation triggers

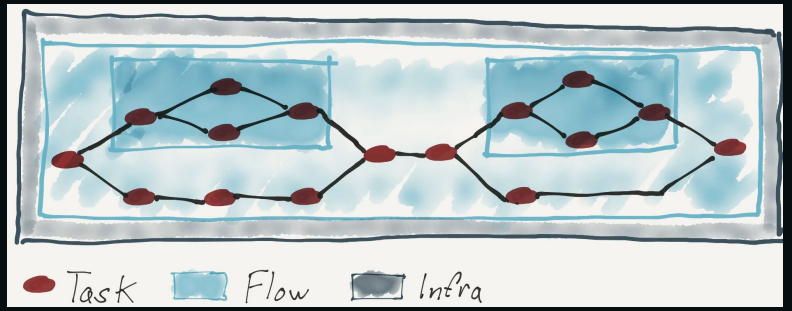
- custom events
- webhooks
- deployment triggers

Workflow patterns

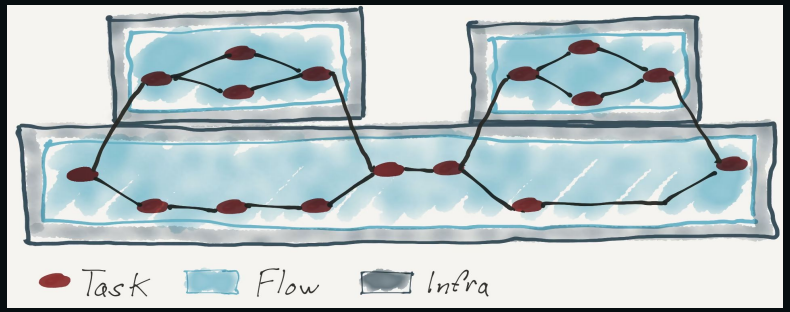
Workflow patterns - prefect.io/blog/workflow-design-patterns



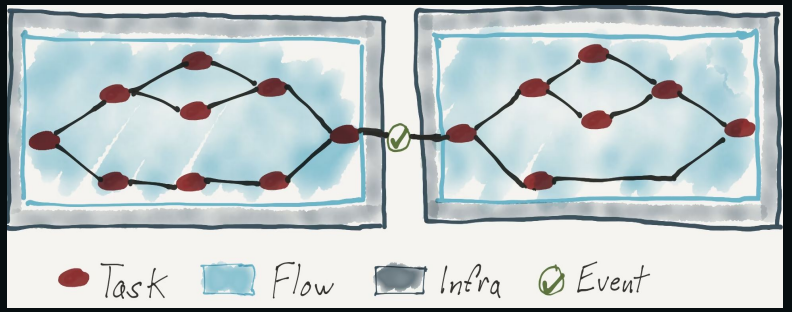
Monoflow



Flow of subflows

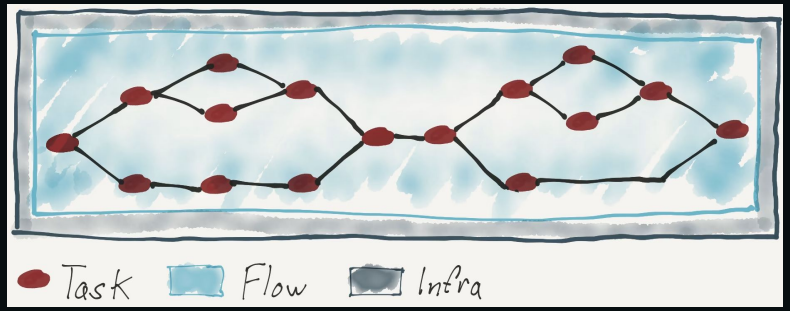


Flow of deployments

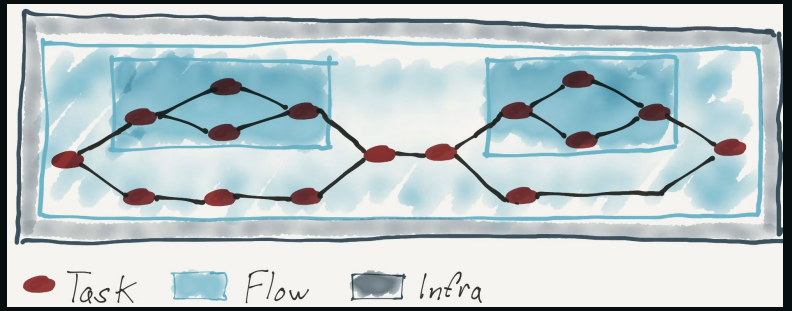


Event triggered flow

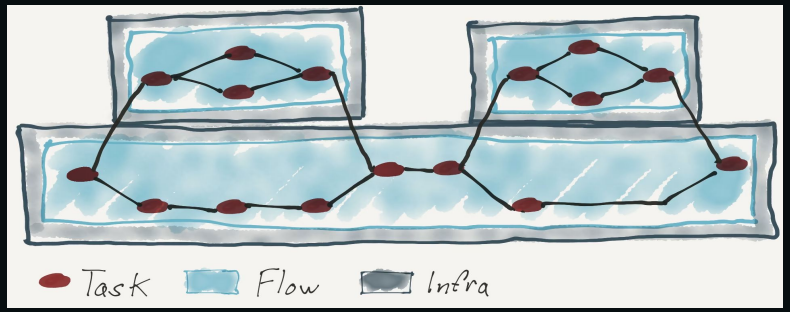
You have seen this pattern



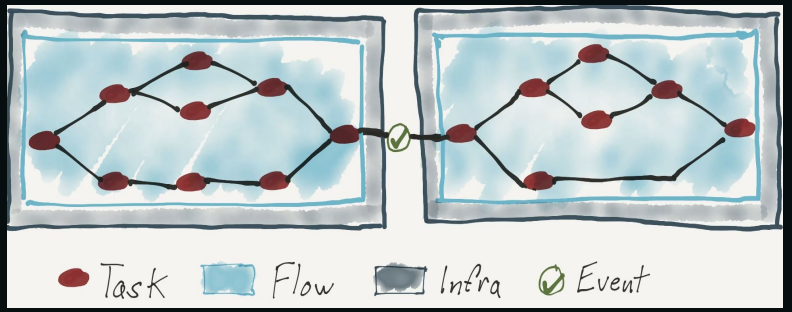
Monoflow



Flow of subflows



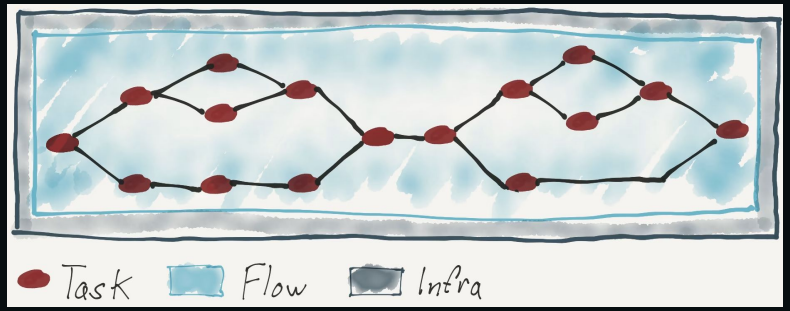
Flow of deployments



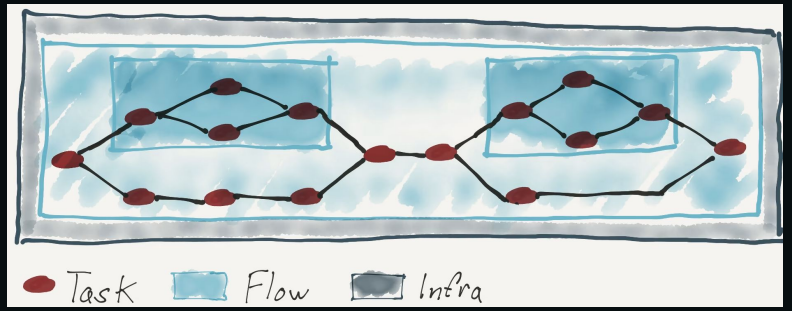
Event triggered flow

Subflows

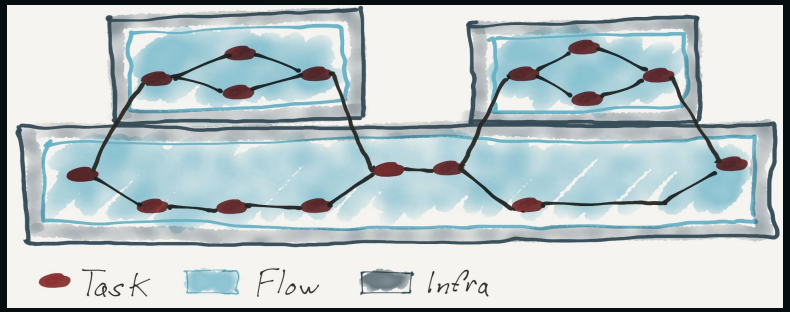
Workflow patterns - Flow of subflows



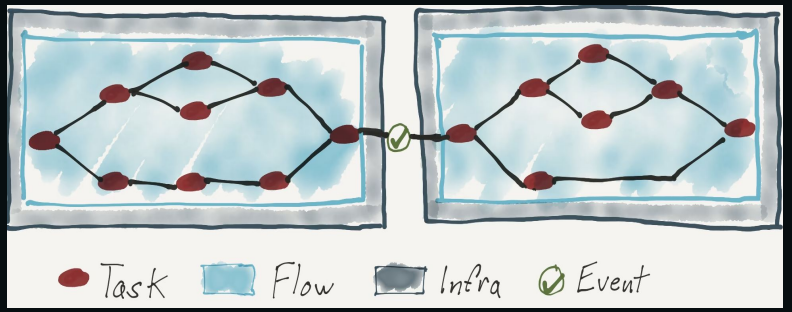
Monoflow



Flow of subflows



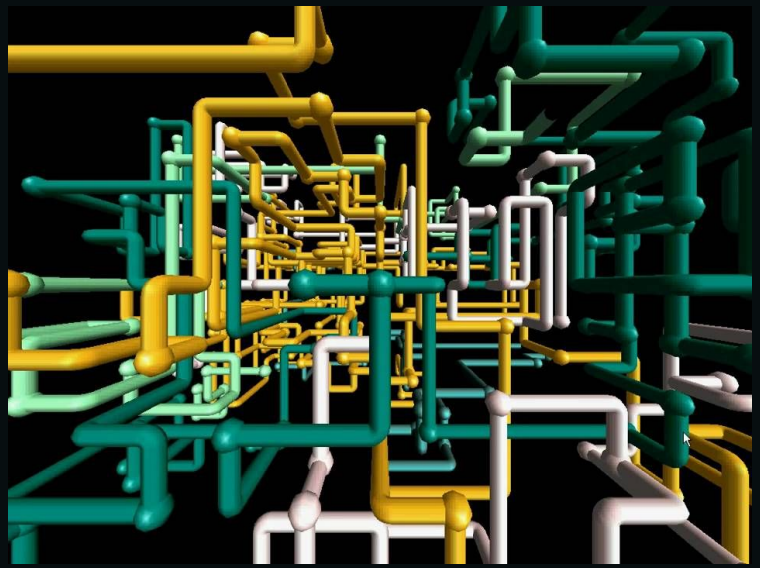
Flow of deployments



Event triggered flow

Subflow

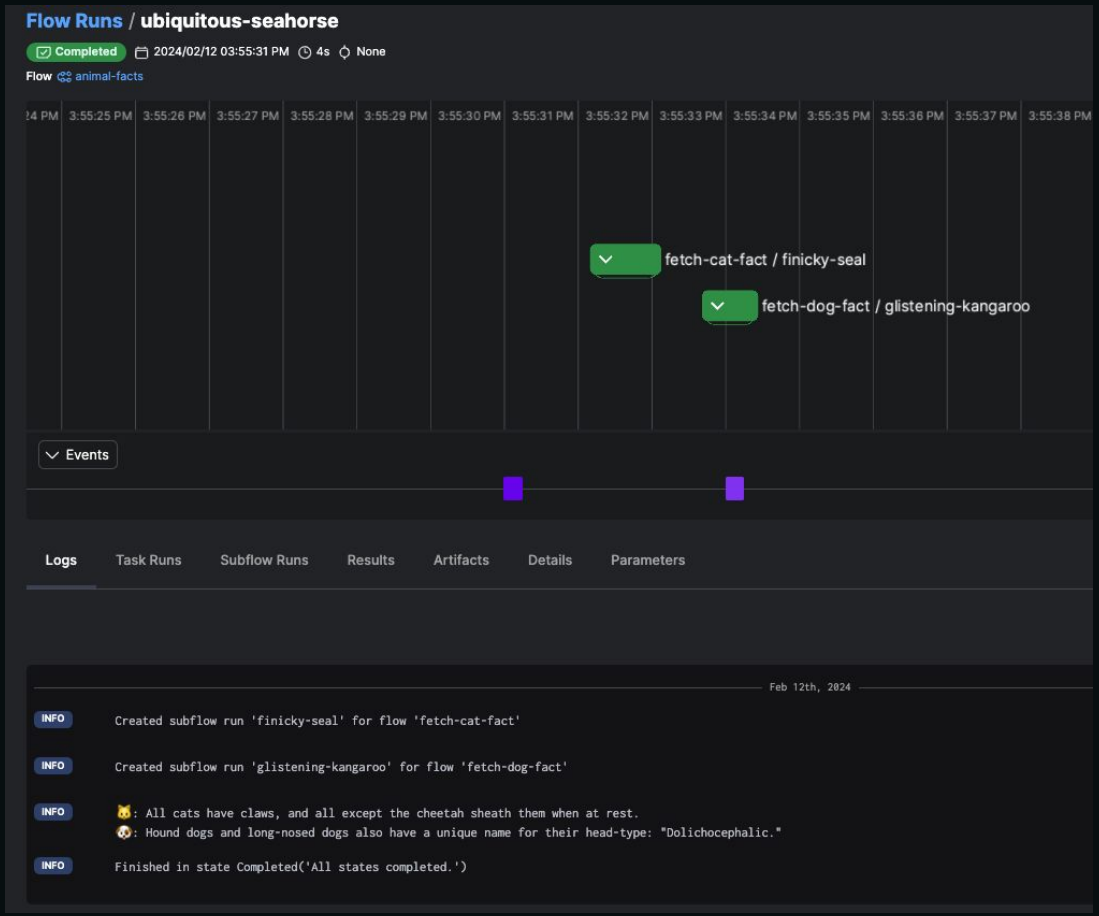
- A flow that calls another flow
- Useful for grouping related tasks



Subflows

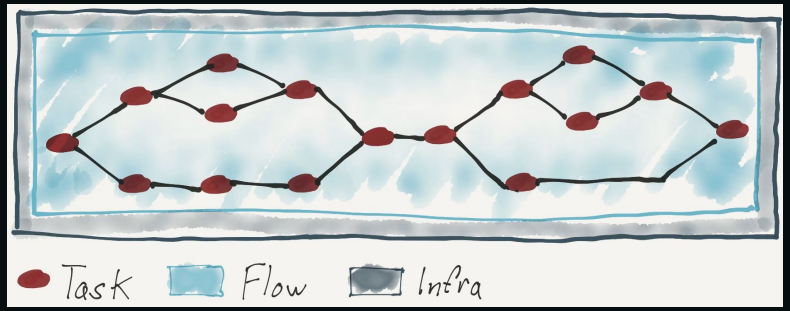
```python
import httpx
from prefect import flow


@flow
def fetch_cat_fact():
    return httpx.get("https://catfact.ninja/fact?max_length=140").json()["fact"]


@flow
def fetch_dog_fact():
    return httpx.get(
        "https://dogapi.dog/api/v2/facts",
        headers={"accept": "application/json"},
    ).json()["data"][0]["attributes"]["body"]


@flow(log_prints=True)
def animal_facts():
    cat_fact = fetch_cat_fact()
    dog_fact = fetch_dog_fact()
    print(f"🐱: {cat_fact} \n🐶: {dog_fact}")


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

Timeline view

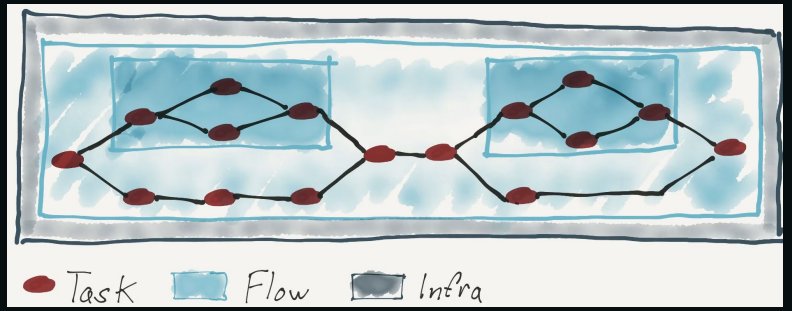


run_deployment

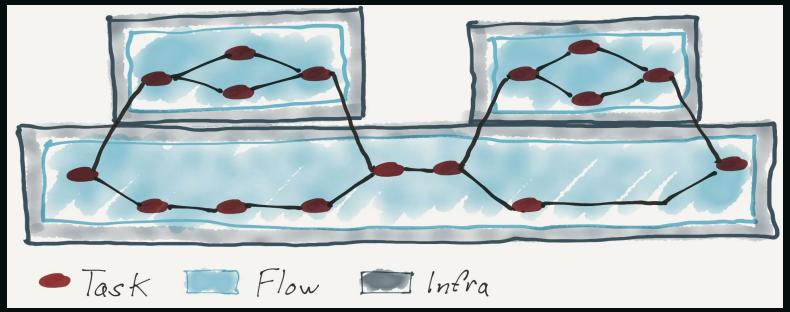
Workflow patterns - Flow of deployments (*run_deployment*)



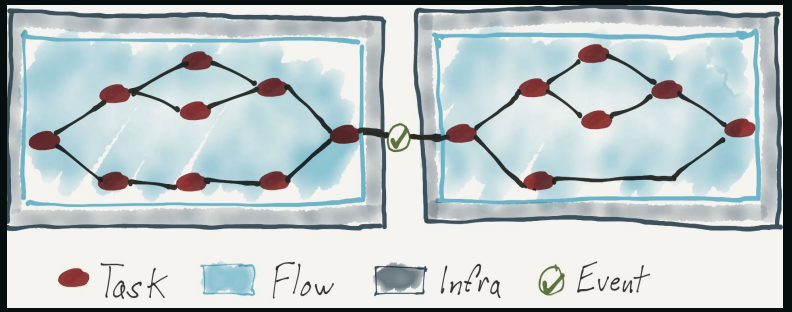
Monoflow



Flow of subflows



Flow of deployments



Event triggered flow

run_deployment

`run_deployment` `async` ¶

Create a flow run for a deployment and return it after completion or a timeout.

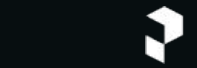
This function will return when the created flow run enters any terminal state or the timeout is reached. If the timeout is reached and the flow run has not reached a terminal state, it will still be returned. When using a timeout, we suggest checking the state of the flow run if completion is important moving forward.

Parameters:

Name	Type	Description	Default
`name`	`Union[str, UUID]`	The deployment id or deployment name in the form: `<slugified-flow-name>/<slugified-deployment-name>`	*required*
`parameters`	`Optional[dict]`	Parameter overrides for this flow run. Merged with the deployment defaults.	`None`

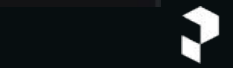
run_deployment

```python
from prefect.deployments import run_deployment

run_deployment(
    name="pipeline/my-first-managed-deployment", parameters={"lat": 1, "lon": 2}
)
```

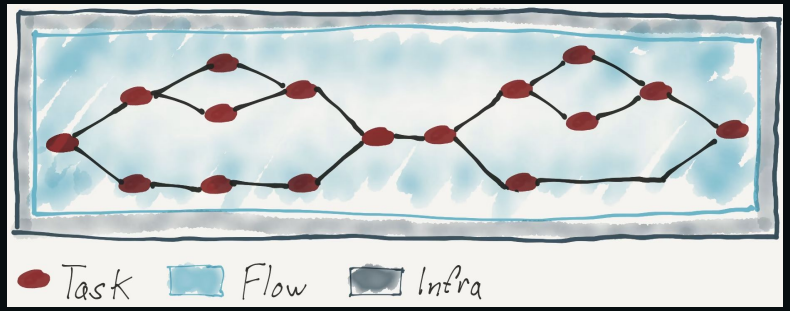
run_deployment

```
INFO    Opening process...                                                  04:52:08 PM
                                                                            prefect.flow_runs.runner

INFO    Created task run 'fetch_weather-0' for task 'fetch_weather'         04:52:15 PM
                                                                            prefect.flow_runs

INFO    Executing 'fetch_weather-0' immediately...                          04:52:15 PM
                                                                            prefect.flow_runs

INFO    Finished in state Completed()                                       04:52:16 PM
                                                                            fetch_weather-0
                                                                            prefect.task_runs

INFO    Created task run 'save_weather-0' for task 'save_weather'           04:52:16 PM
                                                                            prefect.flow_runs

INFO    Executing 'save_weather-0' immediately...                           04:52:16 PM
                                                                            prefect.flow_runs

INFO    Finished in state Completed()                                       04:52:17 PM
                                                                            save_weather-0
                                                                            prefect.task_runs

INFO    Finished in state Completed()                                       04:52:17 PM
                                                                            prefect.flow_runs

INFO    Process for flow run 'sparkling-earthworm' exited cleanly.          04:52:20 PM
                                                                            prefect.flow_runs.runner
```

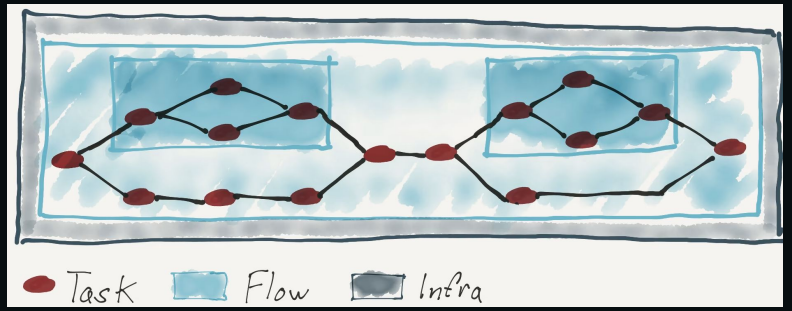



Event-triggered workflows

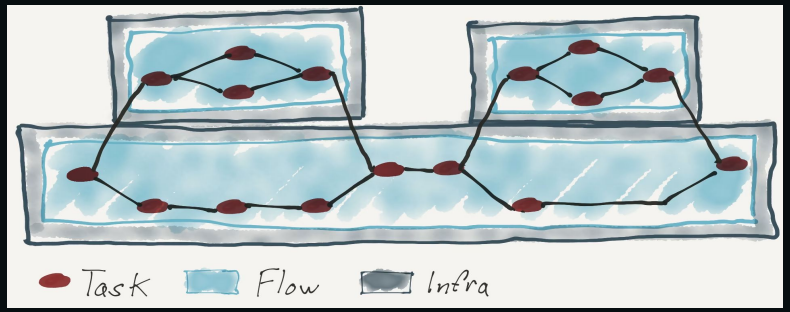
Workflow patterns - Event-triggered



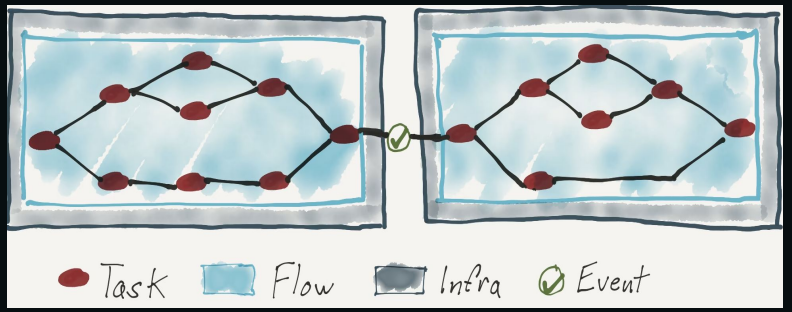
Monoflow



Flow of subflows



Flow of deployments



Event-triggered flow

Custom events in Python

Custom events

Great when working in Python land and want to get data into an automation 🐍

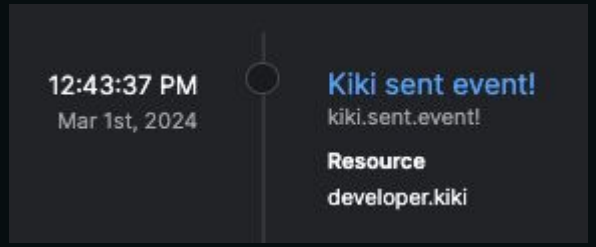
Create custom event to be emitted when code runs

emit_event must provide two args: *event and*

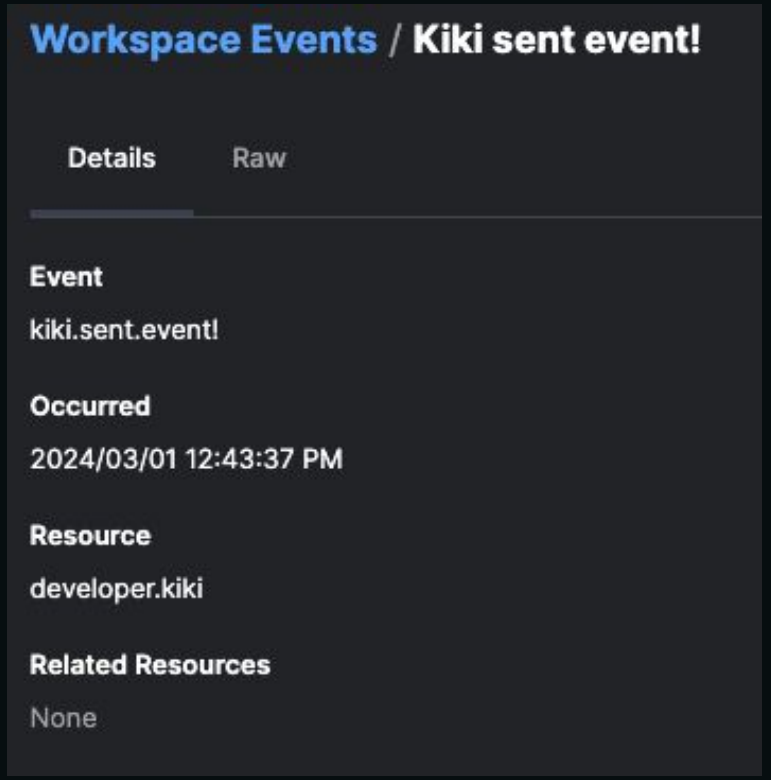
resource= {“prefect.resource.id: val”}

```python
from prefect.events import emit_event


def emit_name_event(name: str = "kiki"):
    """Emit a basic Prefect event with a dynamically populated name"""
    print(f"Hi {name}!")
    emit_event(
        event=f"{name}.sent.event!",
        resource={"prefect.resource.id": f"developer.{name}"},
        payload={"name": name},
    )


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

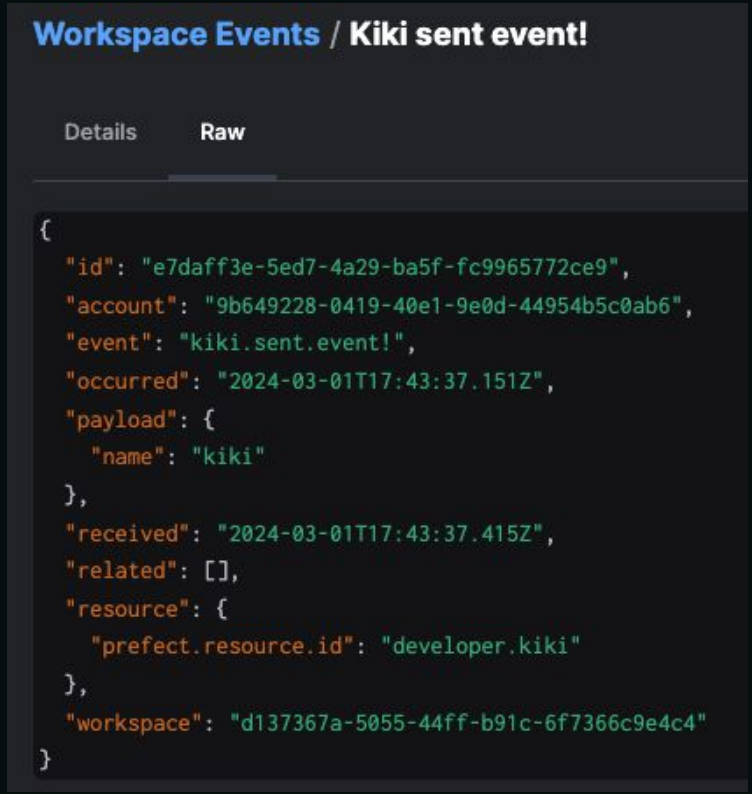

Run code and head to the **Event Feed** page



Click link to see event page



See event details on the **Raw** tab



Data from event can be used in an automation action

For example: Populate a flow param via a Run Deployment action

Use *emit_event*'s *payload* parameter

Example: custom event with detailed payload

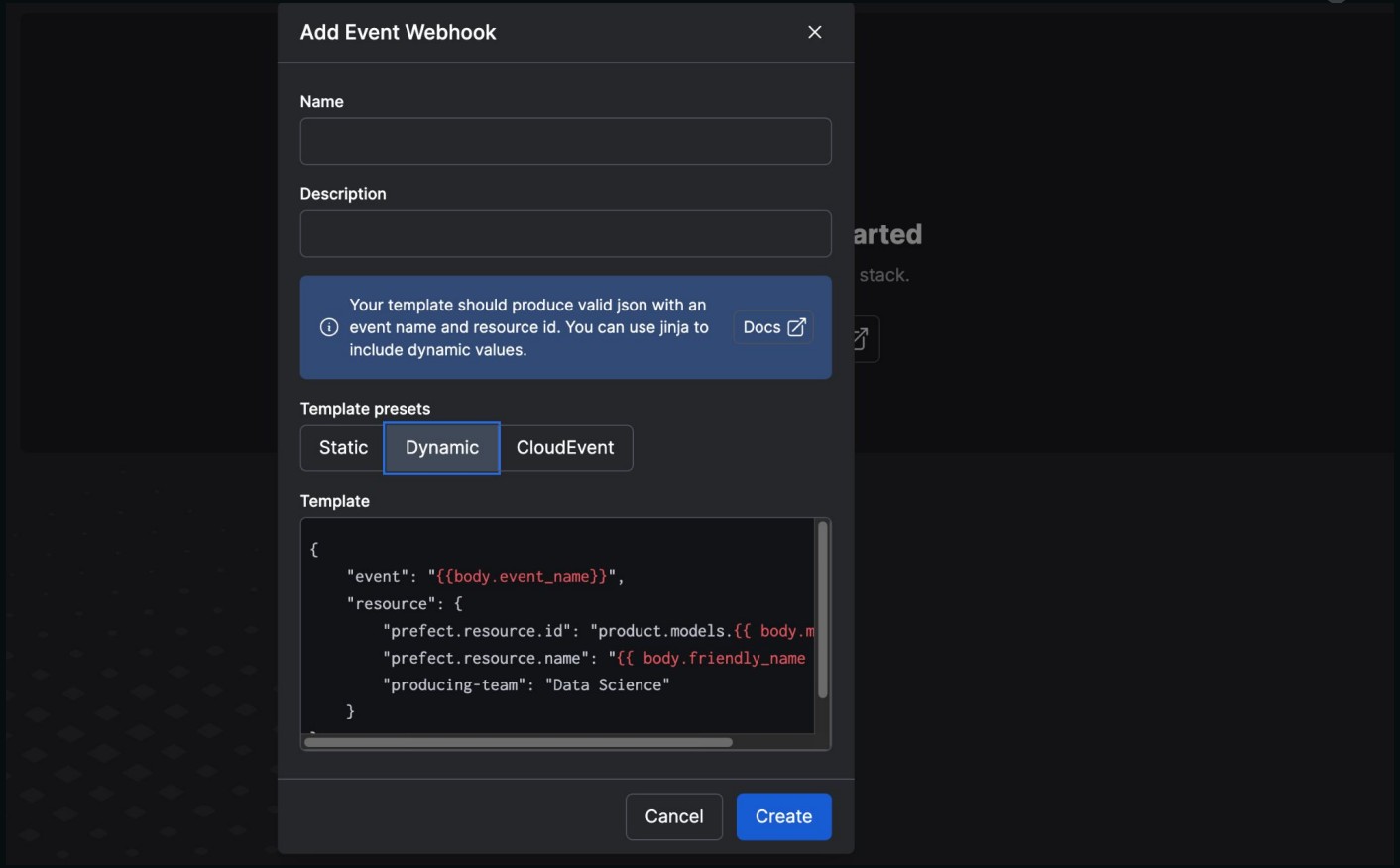
```python
from prefect.events import emit_event

emit_event(
    event=f"bot.{bot.name.lower()}.responded",
    resource={"prefect.resource.id": f"bot.{bot.name.lower()}"},
    payload={
        "user": event.user,
        "channel": event.channel,
        "thread_ts": thread,
        "text": text,
        "response": response.content,
        "prompt_tokens": prompt_tokens,
        "response_tokens": response_tokens,
        "total_tokens": prompt_tokens + response_tokens,
    },
)
```


Event webhooks 🕸️🪝

Event webhooks

- expose a URL endpoint
- provides consistent interface for integrating external applications with Prefect
- when webhook URL is pinged, creates a Prefect event - can be used as a trigger in an automation
- great when **not** in Python land

Event webhooks



Event webhooks

- use Jinja2 for dynamic templating
- template should be valid JSON
- create from UI or CLI

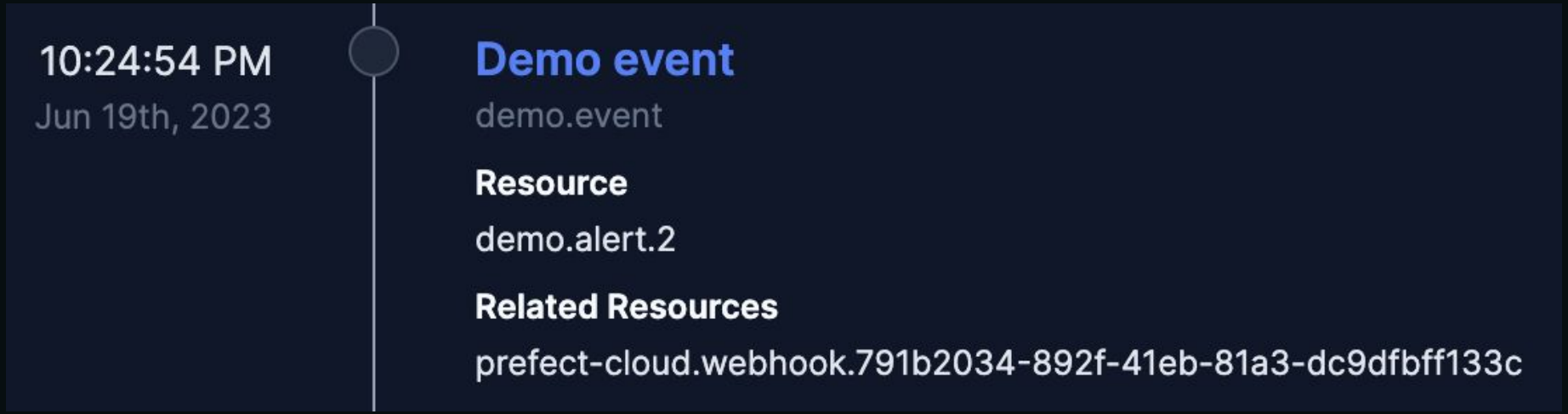
Event webhooks

Hit the endpoint provided by Prefect:

curl https://api.prefect.cloud/hooks/your_slug_here

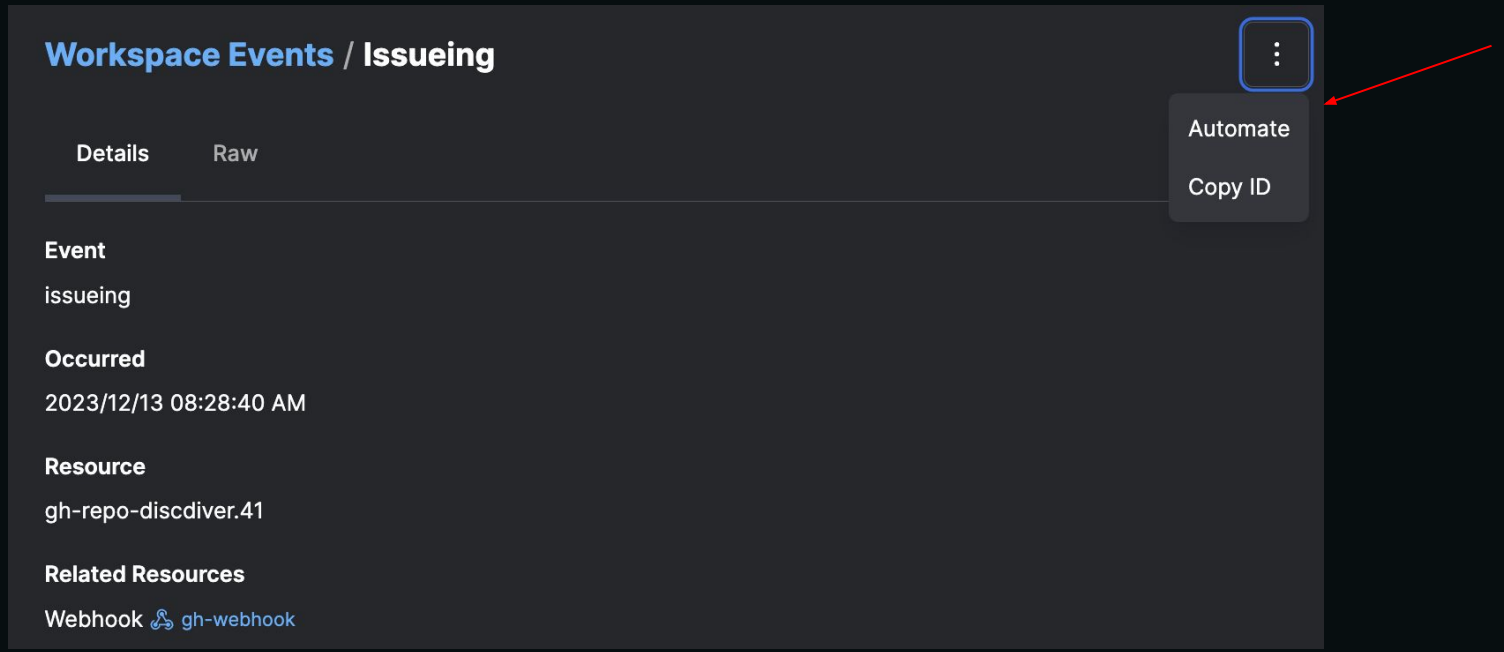
Event webhooks

See the event that is created under **Event Feed** in the UI



Event webhooks

⚡ Use this event as a custom trigger in an automation!



Deployment triggers

Deployment triggers

Alternative approach for creating an automation:

- define an automation in code
- specify the trigger condition in a *DeploymentTrigger* object and pass to *.deploy()*
- creates the automation when the deployment is created

Deployment triggers - the flow to be triggered

```python
from prefect import flow
from prefect.events.schemas import DeploymentTrigger


@flow(log_prints=True)
def downstream_flow(ticker: str = "AAPL") -> str:
    print(f"got {ticker}")
```

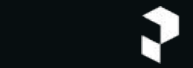

Deployment triggers - the trigger

Create a *DeploymentTrigger* object

```
downstream_deployment_trigger = DeploymentTrigger(
    name="Upstream Flow - Pipeline",
    enabled=True,
    match_related={
        "prefect.resource.id": "prefect.flow.5c933ae4-dd43-4705-90eb-cfdeb4c028fb"
    },
    expect={"prefect.flow-run.Completed"},
)
```

See the event specification docs:

docs.prefect.io/cloud/events/#event-specification



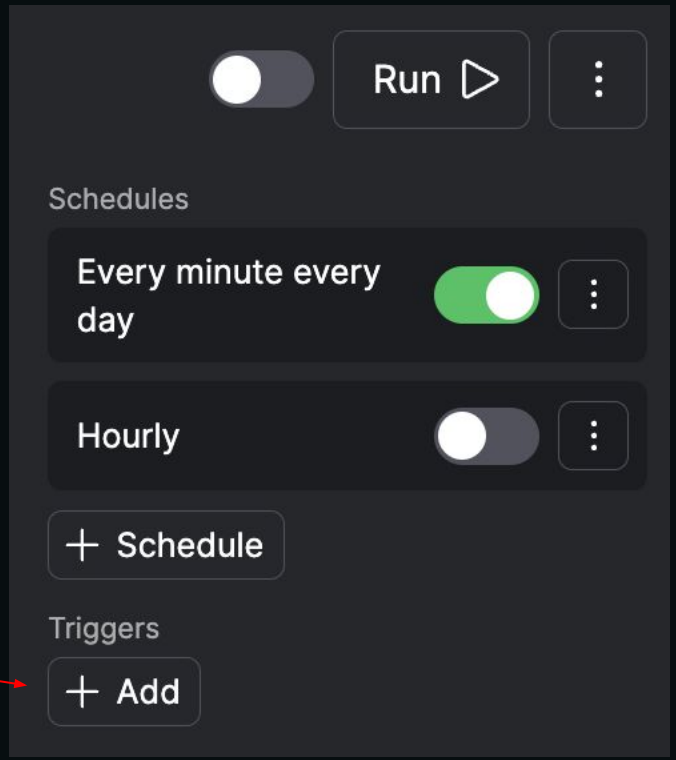
Deployment triggers - create

Pass the trigger object to *.deploy* and run the script

```python
if __name__ == "__main__":
    downstream_flow.from_source(
        source="https://github.com/discdiver/pacc-2024.git",
        entrypoint="106/deployment-trigger.py:downstream_flow",
    ).deploy(
        name="ticker-deploy",
        work_pool_name="managed1",
        triggers=[downstream_deployment_trigger],
    )
```

Another way to begin automation creation in the UI:

- Start from a deployment page
- Click the **+ Add** button under **Triggers**
- Pre-populates the automation action with the deployment run

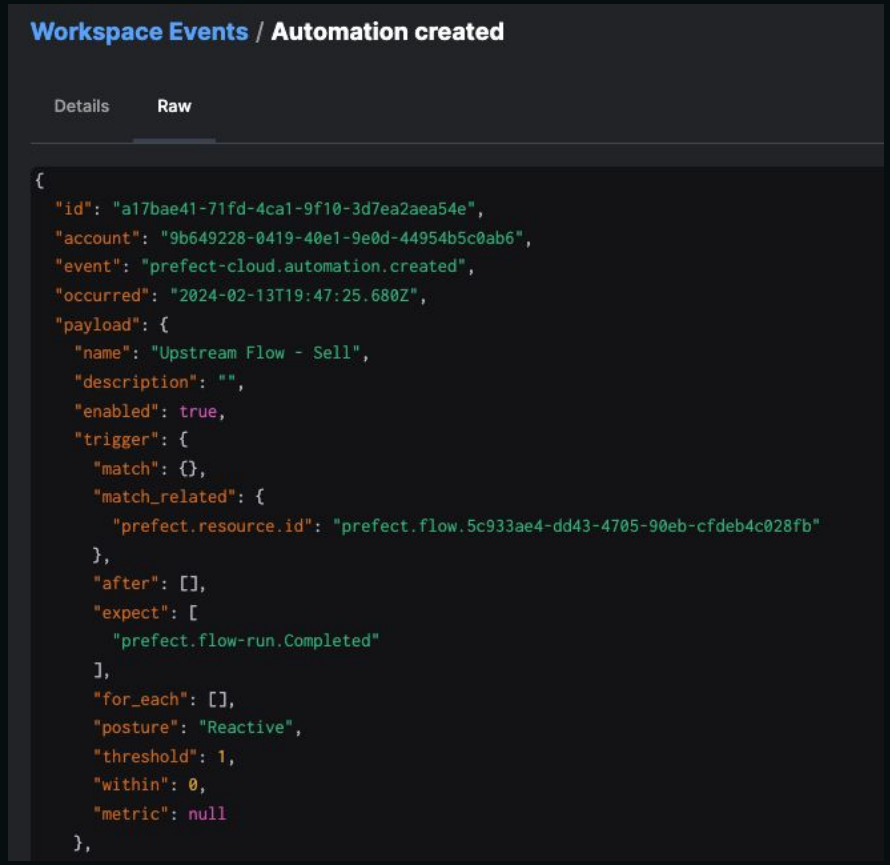


Specifying an automation trigger

To create a custom trigger check out an event in the UI (**Raw** tab)

You can copy/paste and adjust in the trigger JSON.

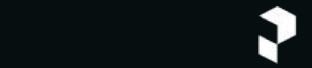
See the Events docs.



106 Recap

You've seen how to use several workflow patterns with

- subflows
- *run_deployment*
- automations
  - custom events defined in Python
  - webhooks
  - trigger defined in code at deployment creation



106 Lab

- Create a deployment that uses a subflow
- Create a second deployment that uses *run_deployment*
- Stretch: Create a webhook and an automation that runs a deployment when that webhook fires
- Stretch: Create a custom event in Python that triggers a notification action in an automation
- Super-stretch: Create a deployment that contains a trigger defined in Python code

If you give an engineer a job…

Could you just fetch this data and save it? Oh, and …

1. set up logging?
2. do it every hour?
3. visualize the dependencies?
4. automatically retry if it fails?
5. create an artifact for human viewing?
6. add caching?
7. add collaborators to run and view - who don't code?
8. send me a message when it succeeds?
9. run it in a Docker container-based environment?
10. pause for input?
11. automatically declare an incident when a % of workflows fail?
12. automatically run a workflow in response to an event?

Wrap

Brief feedback survey

Please let us know what went well and what could be improved. 🎉

Congratulations!!!

PREFECT ASSOCIATE

CERTIFICATION

Bonus content

Bonus content

- Prefect variables
- Task runners & async code
- Testing
- Prefect REST API
- Upload data to AWS S3
- Self hosted server instance
- Prefect profiles
- Deploy multiple flows
- Guided deployment creation with *prefect deploy*
- Deployments with *prefect.yaml*
- CI/CD with GitHub Actions
- Helm chart
- Terraform provider

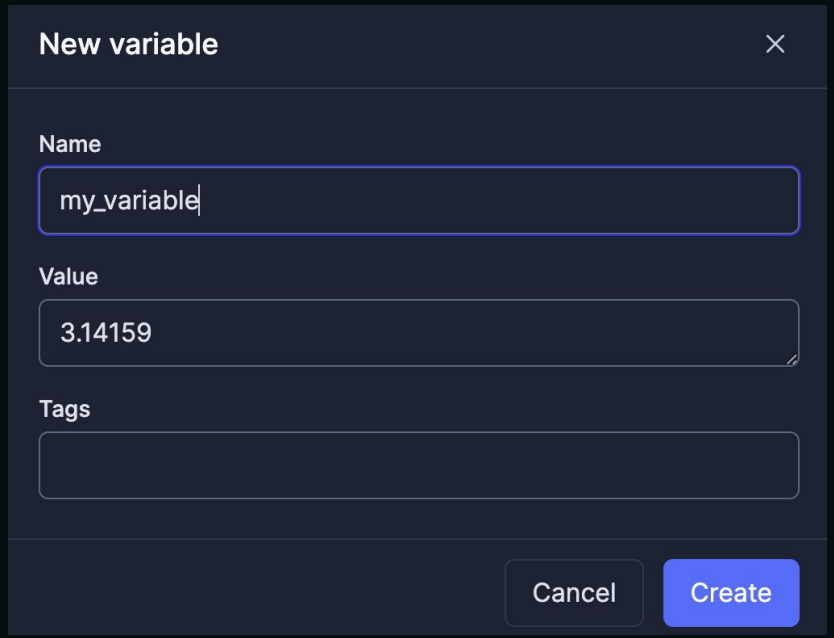
Variables

Prefect variables

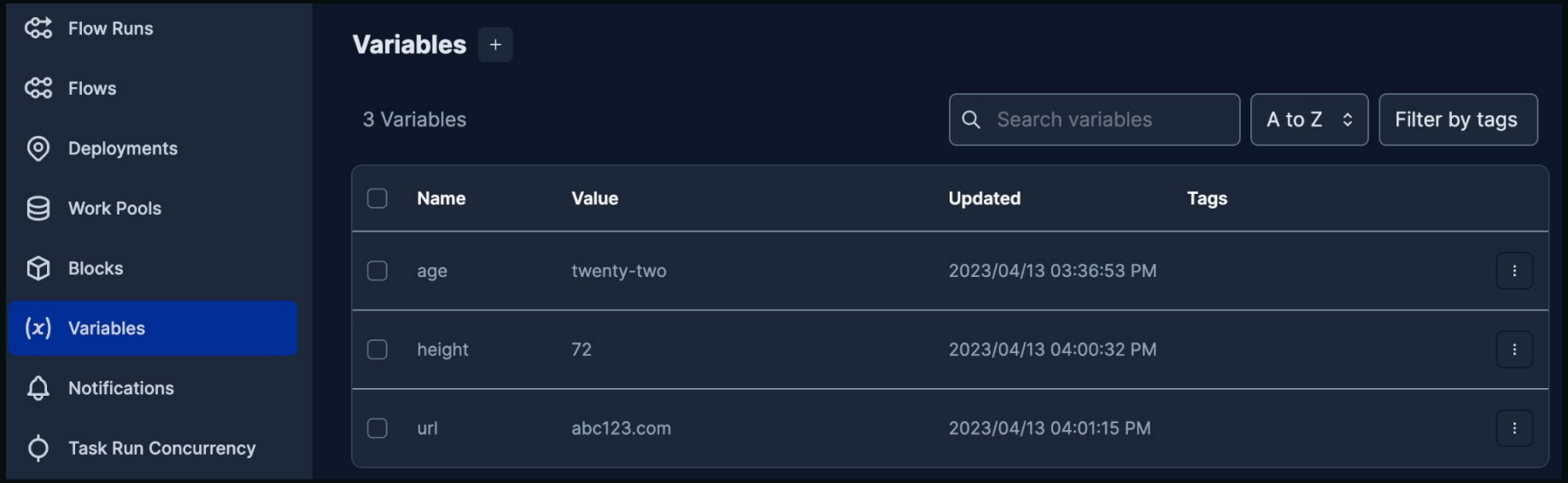
- String values evaluated at runtime
- Store and reuse non-sensitive, small data
- Create via UI or CLI

Prefect variables

Only string values



Prefect variables



Task runners for concurrency

Concurrency

- Helpful when waiting for external systems to respond
- Allows other work to be done while waiting
- Prefect's *ConcurrentTaskRunner* replaces need for using Python's *async, await,* etc.

Concurrency & Parallelism: via task runners

Concurrency & Parallelism

- **Concurrency:** single-threaded, interleaving, GIL locked
- **Parallelism:** multiple events run at the same time

Your Prefect code runs **sequentially** by default

Concurrency

Concurrency

- Helpful when waiting for external systems to respond (IO / network-bound work)
- Prefect's *ConcurrentTaskRunner* allows you to concurrently execute code without *async* syntax

Concurrency

```python
from prefect import flow, task
from prefect.task_runners import ConcurrentTaskRunner

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

@flow(task_runner=ConcurrentTaskRunner())
def elevator():
    for floor in range(3, 0, -1):
        stop_at_floor.submit(floor)

elevator()
```


Concurrency

```
elevator moving to floor 3
elevator stops on floor 3
elevator moving to floor 1
elevator stops on floor 1
elevator moving to floor 2
elevator stops on floor 2
```


Task Runners

- Specify in *flow* decorator
- *ConcurrentTaskRunner* is ready by default
- Use *.submit()* when call a task to return a *PrefectFuture* instead of direct result

Task runners for true parallelism

Parallelism

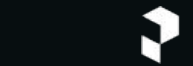
- Two or more operations happening at the same time on one or more machines
- Helpful when operations limited by CPU
- Many machine learning algorithms parallelizable

Task Runners for parallelism

- DaskTaskRunner
- RayTaskRunner

Both require an integration package:

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



DaskTaskRunner for parallelism

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


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


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


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


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


`DaskTaskRunner` for parallelism

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

Prefect REST API

If you want to talk to the API without Python

Cloud and server REST API interactive docs:

docs.prefect.io/latest/api-ref/rest-api

curl or use an HTTP client (*httpx*, *requests*)

PrefectClient to interact with the REST API

Or use the built-in *PrefectClient* for convenience

```python
from prefect import get_client

async with get_client() as client:
    response = await client.hello()
    print(response.json()) # 👋
```

docs.prefect.io/guides/using-the-client

Common methods

- *create_flow_run_from_deployment*
- *read_flow_run / read_flow_runs*
- *update_deployment*
- *delete_flow_run*

github.com/PrefectHQ/prefect/blob/main/src/prefect/client/orchestration.py

Testing

Testing

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

```python
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

```python
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
```

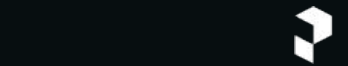

Upload data to AWS S3

Steps

1. Install prefect-aws
2. Register new blocks
3. Create S3 bucket
4. Create S3Bucket block from UI or CLI
5. Use in a flow

Install prefect-aws

pip install -U prefect-aws



Register new blocks

prefect blocks register -m prefect_aws

See block types & blocks from CLI

prefect block type ls

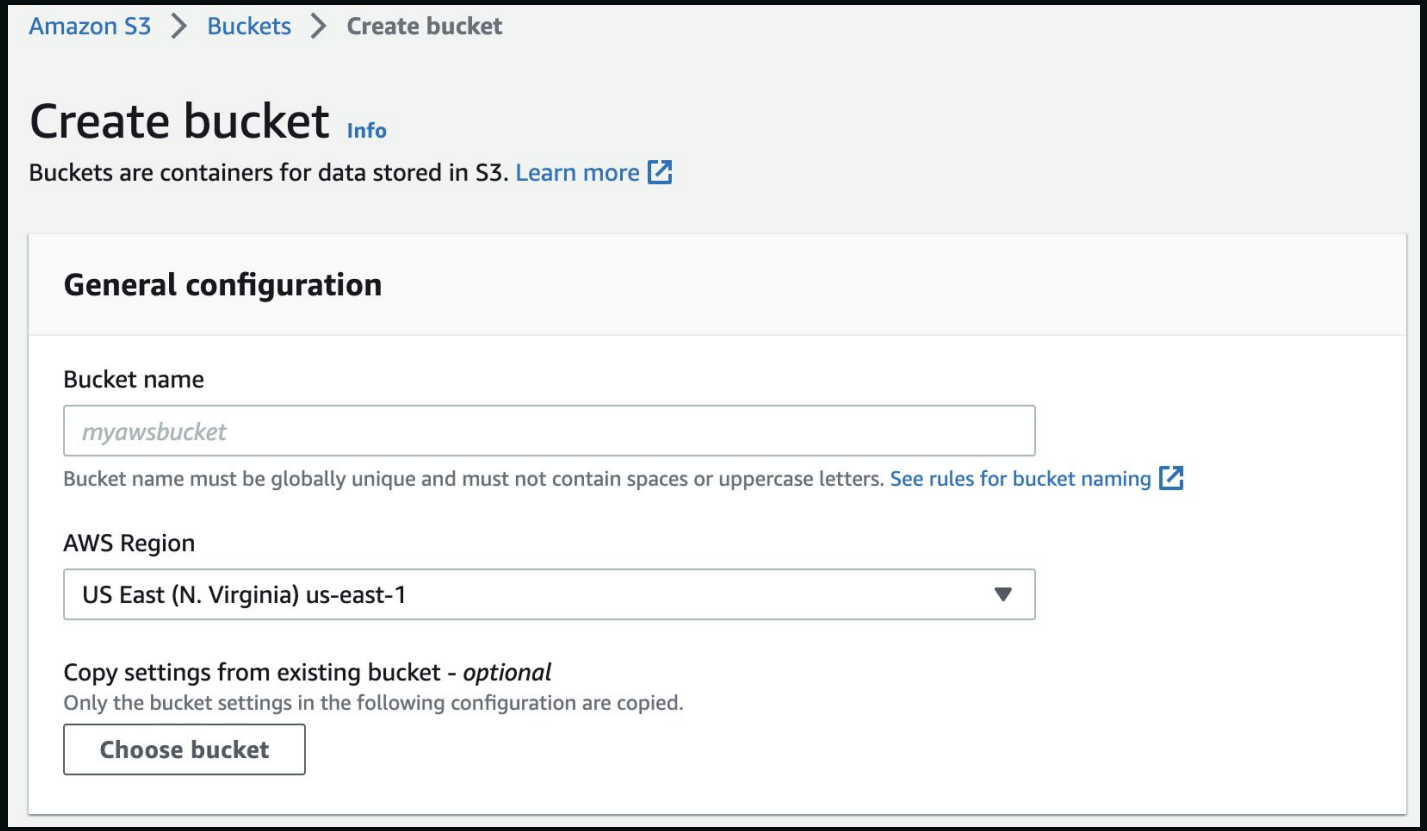
prefect block ls

Make an *S3Bucket* block

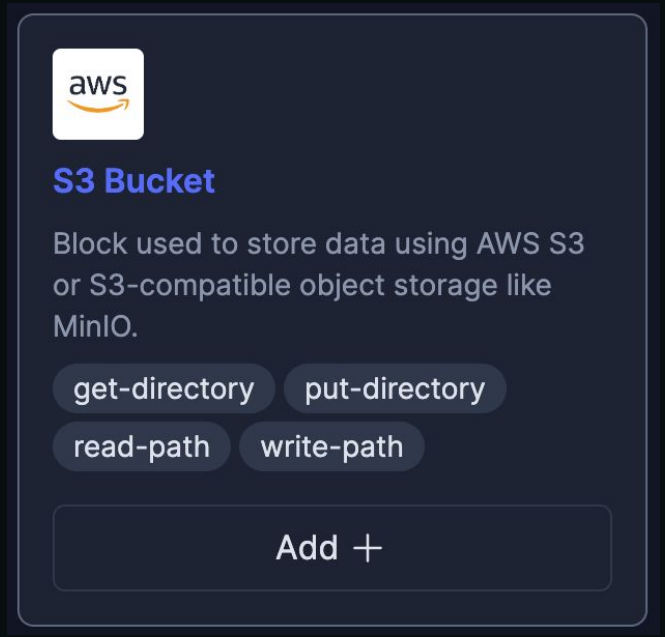
⚠️ ***S3Bucket* block from prefect-aws != *S3* block that ships with Prefect**

- Both block types upload and download data
- *S3Bucket* block has many methods
- We are showing how to use *S3*Bucket block

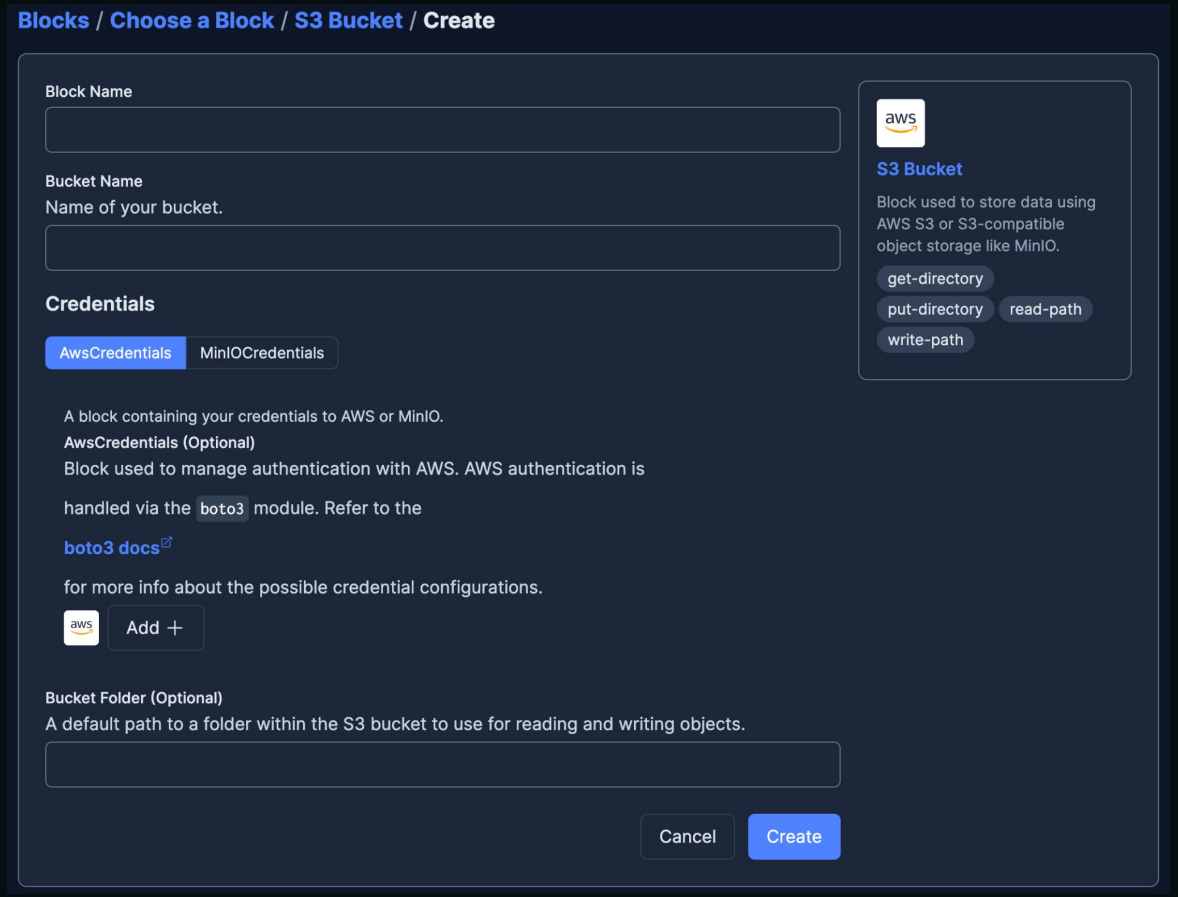
Create S3 Bucket



Create S3Bucket block from UI

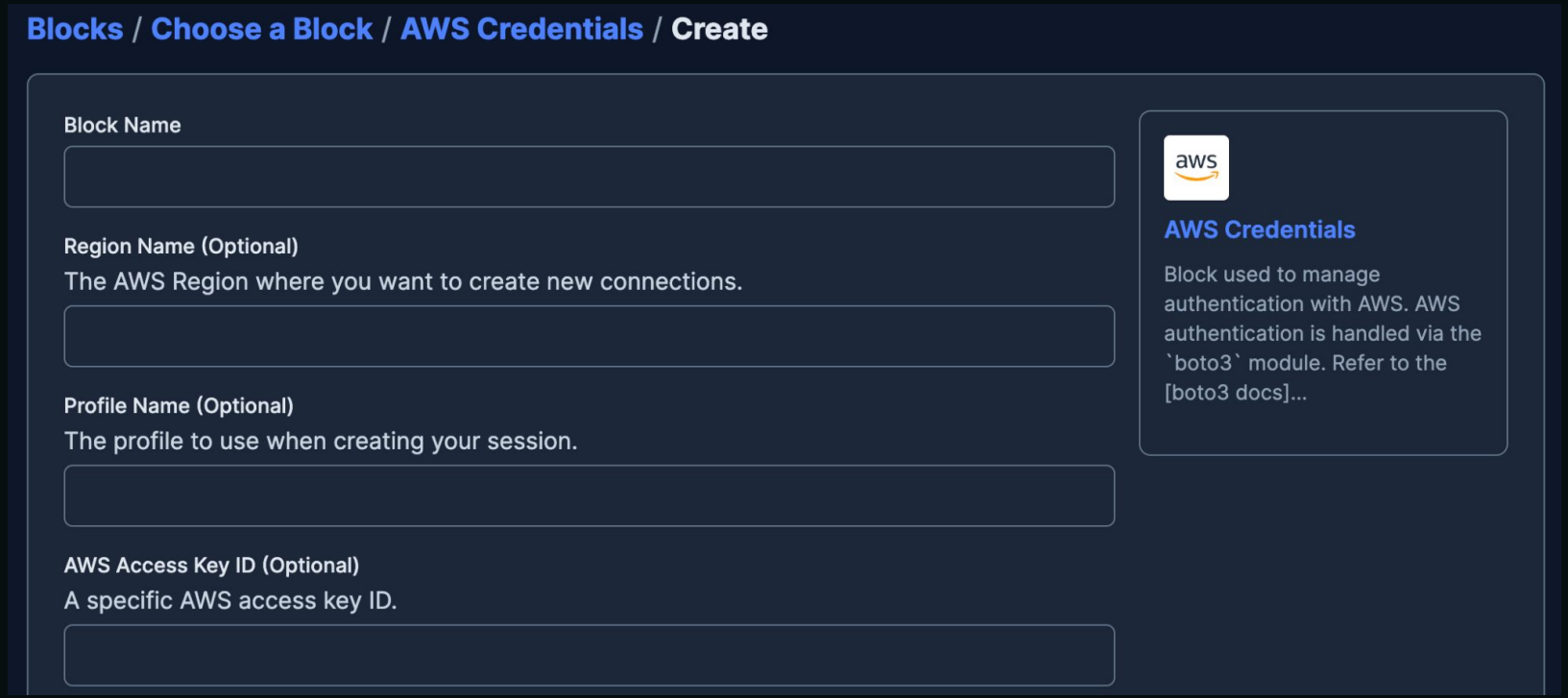


Create S3Bucket block from UI



AWS Credentials block from UI

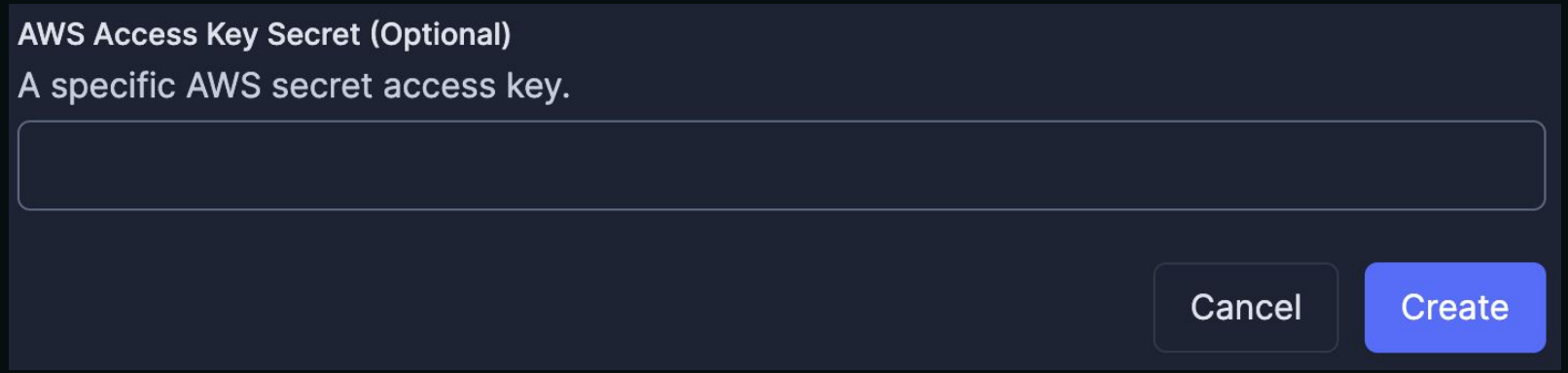
Use the nested AWS Credentials block as needed



AWS Credentials block from UI

Leave most fields blank.

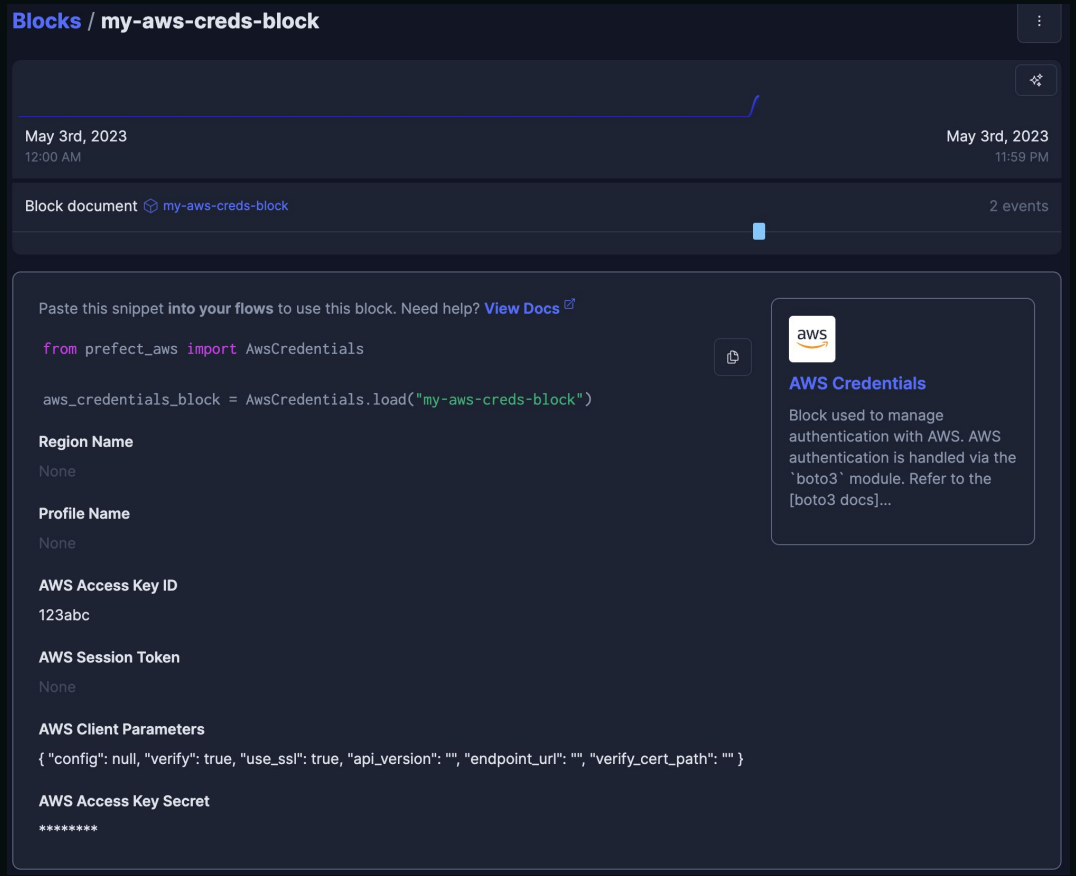
Probably use *AWS Access Key ID* & *AWS Access Key Secret*.



Or create blocks with Python code

```python
from time import sleep
from prefect_aws import S3Bucket, AwsCredentials

def create_aws_creds_block():
    # environment variables can be helpful for creating credentials blocks
    # do not store credential values in public locations (e.g. GitHub public repo)
    my_aws_creds_obj = AwsCredentials(
        aws_access_key_id="123abc",
        aws_secret_access_key="ab123",
    )
    my_aws_creds_obj.save(name="my-aws-creds-block", overwrite=True)

def create_s3_bucket_block():
    aws_creds = AwsCredentials.load("my-aws-creds-block")
    my_s3_bucket_obj = S3Bucket(
        bucket_name="my-first-bucket-abc", credentials=aws_creds
    )
    my_s3_bucket_obj.save(name="s3-bucket-block", overwrite=True)

if __name__ == "__main__":
    create_aws_creds_block()
    sleep(5)  # ensure server has time to create credentials block before loading
    create_s3_bucket_block()
```


View block in the UI



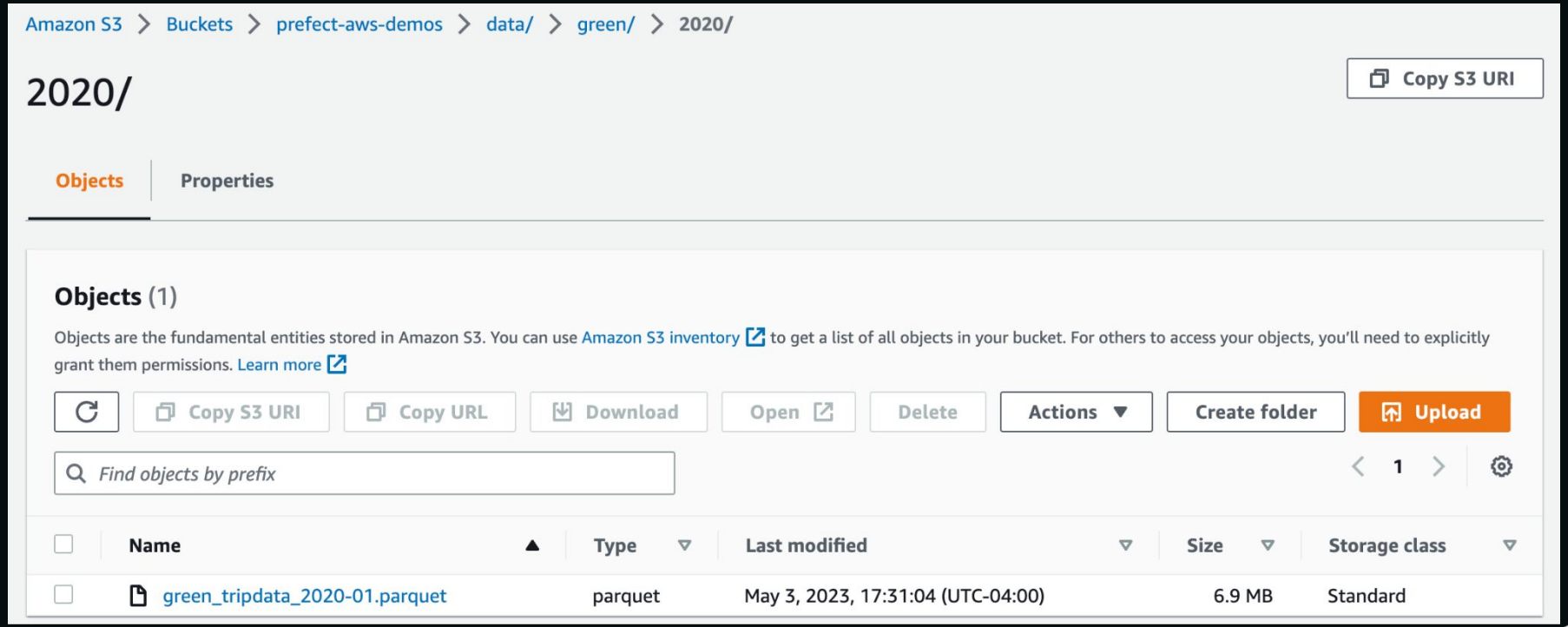
Flow code loads S3 block and uploads data file

```python
from pathlib import Path
from prefect import flow
from prefect_aws.s3 import S3Bucket

@flow()
def upload_to_s3(color: str, year: int, month: int) -> None:
    """The main flow function to upload taxi data"""
    path = Path(f"data/{color}/{year}/{color}_tripdata_{year}-{month:02}.parquet")
    s3_block = S3Bucket.load("s3-bucket-block")
    s3_block.upload_from_path(from_path=path, to_path=path)

if __name__ == "__main__":
    upload_to_s3(color="green", year=2020, month=1)
```

Use your flow code!

- Can test with *python my_script.py*
- Then create a deployment and run it! 🎉

See file in S3 bucket



Self-hosted server instance

Self-hosted server instance

Alternative to Prefect Cloud: host your own Prefect server instance

- Backed by SQLite db by default
- Or use PostgreSQL in production
- Similar UI
- No events, push work pools, email server, authentication, user management, error summaries, etc.

Self-hosted server instance

- Switch to a new profile
- Use an ephemeral API (default) or set the API endpoint (required if in a Docker container)

Self-hosted server instance

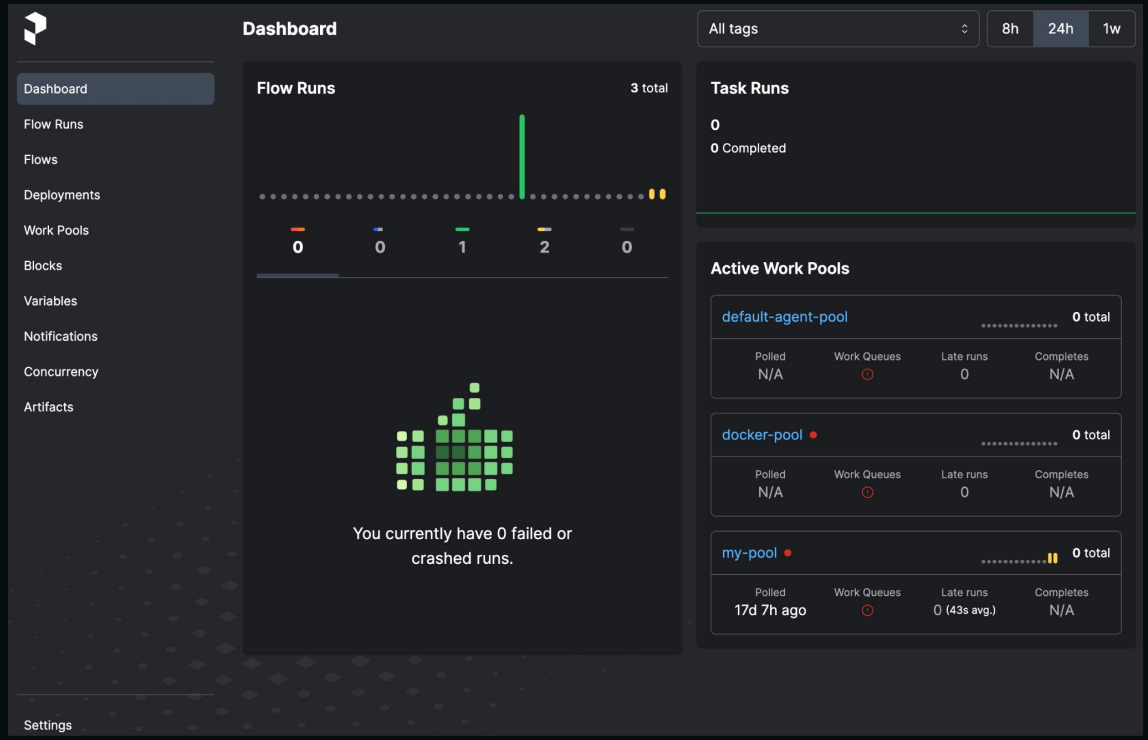
Start a server in another terminal with:

prefect server start

```
 ___ ___ ___ ___ ___ ___ _____
| _ \ _ \ __| __| __/ __|_   _|
|  _/   / _|| _|| _| (__  | |
|_| |_|_\___|_| |___\___| |_|

Configure Prefect to communicate with the server with:

    prefect config set PREFECT_API_URL=http://127.0.0.1:4200/api

View the API reference documentation at http://127.0.0.1:4200/docs

Check out the dashboard at http://127.0.0.1:4200
```


Self-hosted server instance

Head to the UI at http://127.0.0.1:4200



Self-hosted server instance

Required when running Prefect inside a container:

PREFECT_API_URL="http://127.0.0.1:4200/api"

See Prefect Helm Chart if running on Kubernetes
github.com/PrefectHQ/prefect-helm

Prefect profiles

Prefect profiles

If you don’t already have a profile with Prefect Cloud you want to use for this course, create a new profile

Create: *prefect profile create my_cloud_profile*

Prefect profiles

Inspect: *prefect profile inspect my_cloud_profile*

Select: *prefect profile use my_cloud_profile*

Deploy multiple flows with *serve*

Deploy multiple flows

```python
import time
from prefect import flow, serve

@flow
def slow_flow(sleep: int = 60):
    "Sleepy flow - sleeps the provided amount of time (in seconds)."
    time.sleep(sleep)

@flow
def fast_flow():
    "Fastest flow this side of the Atlantic."
    return

if __name__ == "__main__":
    slow_deploy = slow_flow.to_deployment(name="sleeper-scheduling")
    fast_deploy = fast_flow.to_deployment(name="fast-scheduling")
    serve(slow_deploy, fast_deploy)
```


Deploy multiple flows

- import *serve*
- use *to_deployment()* method
- use *serve* function and pass it the deployment objects

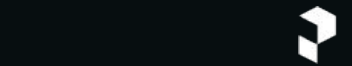
Guided deployment creation

Deployments: ETL code

```python
@task
def fetch_cat_fact():
    return httpx.get("https://catfact.ninja/fact?max_length=140").json()["fact"]

@task
def formatting(fact: str):
    return fact.title()

@task
def write_fact(fact: str):
    with open("fact.txt", "w+") as f:
        f.write(fact)
    return "Success!"
```


Deployments: ETL code

```python
@flow
def pipe():
    fact = fetch_cat_fact()
    formatted_fact = formatting(fact)
    msg = write_fact(formatted_fact)
    print(msg)
```

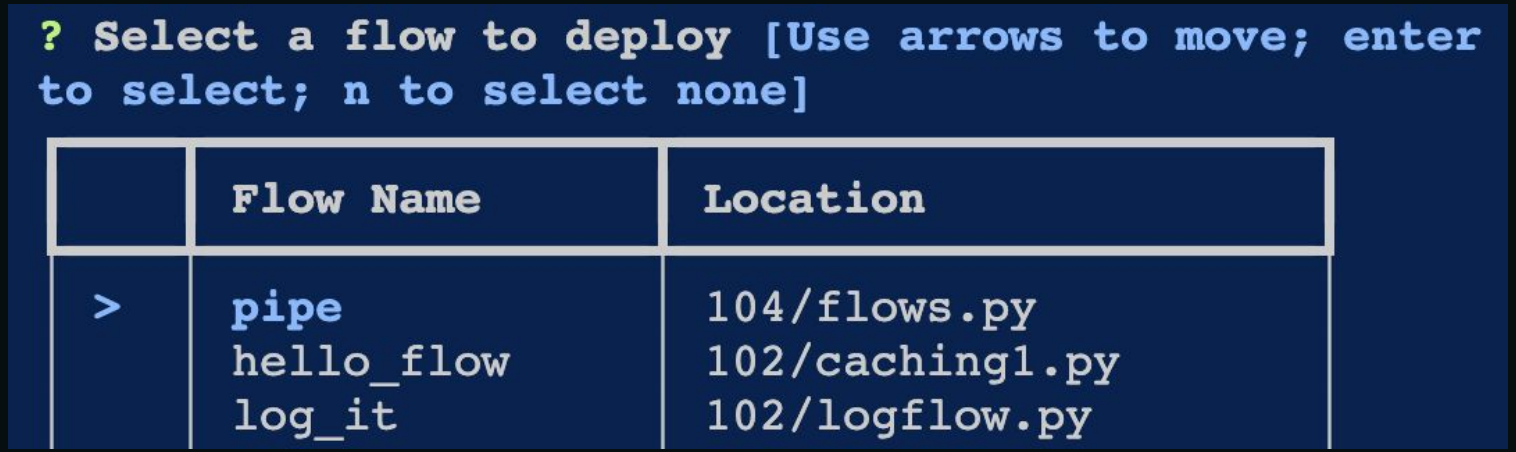



Send deployment to server

From the **root of your repo** run:

prefect deploy

Choose the flow you want to put into a deployment



Send deployment to server

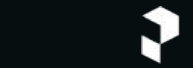
Enter a deployment name and then *n* for no schedule.

```
? Deployment name (default): first_deploy
? Would you like to schedule when this flow runs? [y/n] (y): n
```

Create a work pool

```
? Looks like you don't have any work pools this flow can be deployed to. Would you like to create one? [y/n] (y): y
? What infrastructure type would you like to use for your new work pool? [Use arrows to move; enter to select]
```

	Type	Description
>	process	Execute flow runs as subprocesses on a worker. Works well for local execution when first getting started.
	ecs	Execute flow runs within containers on AWS ECS. Works with existing ECS clusters and serverless execution via AWS Fargate. Requires an AWS account.



Work pools

Give your work pool a name.

Or, if you have existing work pools, choose one

```
? Which work pool would you like to deploy this flow to? [Use arrows to move; enter to select]
```

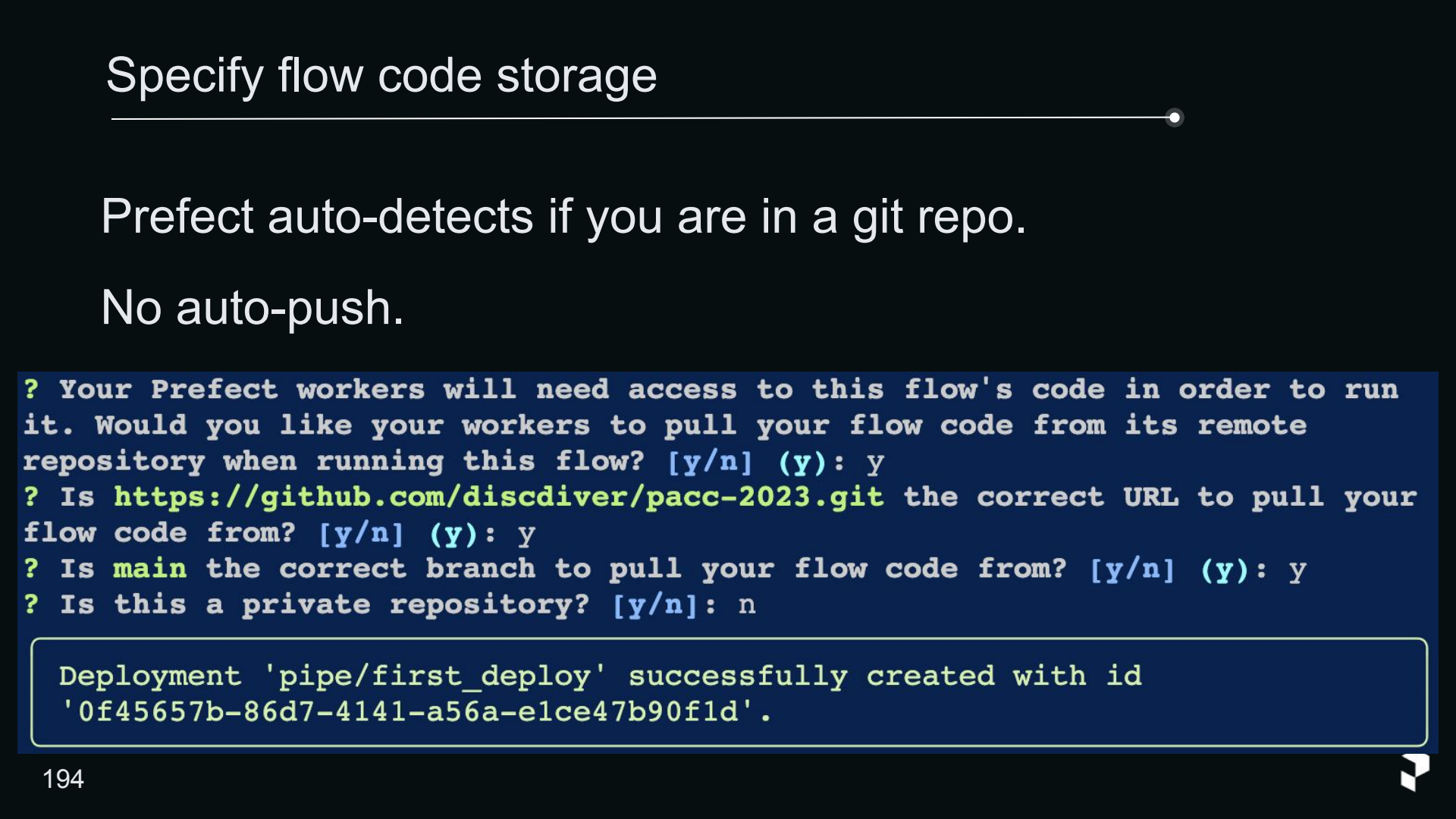
	Work Pool Name	Infrastructure Type	Description
	docker-work	docker	
	local-work	process	
>	**my-pool**	process	
	prod-pool	kubernetes	
	staging-pool	kubernetes	
	zoompool	process	

Specify flow code storage

Prefect auto-detects if you are in a git repo.

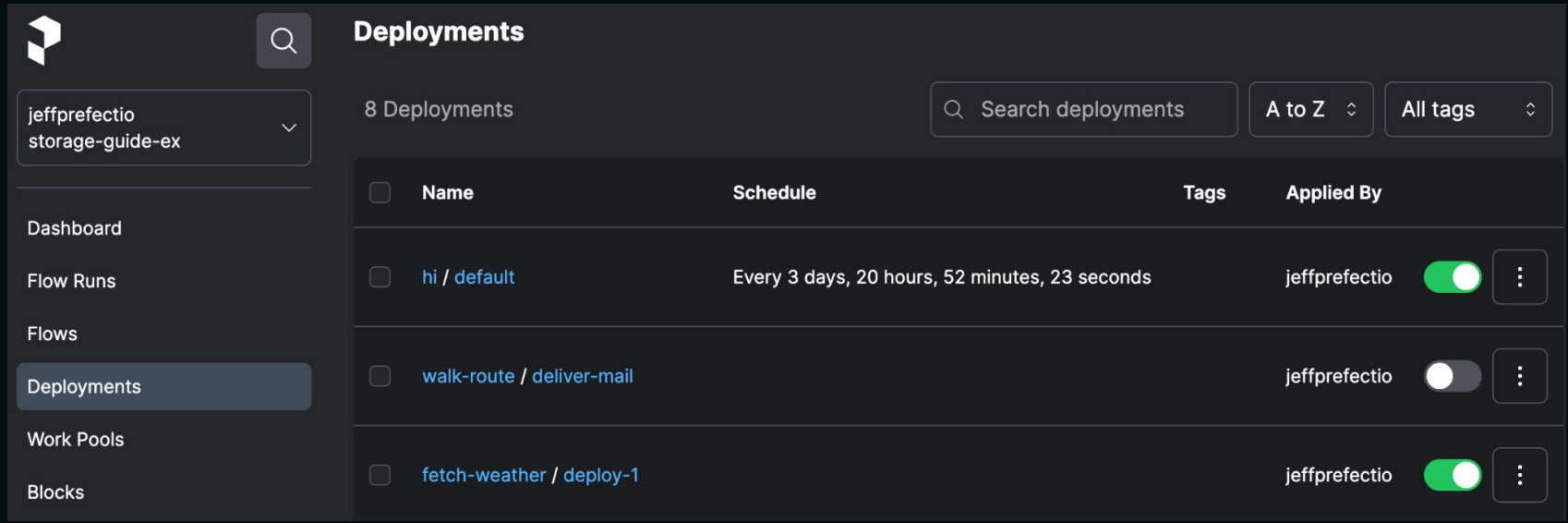
No auto-push.

```
? Your Prefect workers will need access to this flow's code in order to run
it. Would you like your workers to pull your flow code from its remote
repository when running this flow? [y/n] (y): y
? Is https://github.com/discdiver/pacc-2023.git the correct URL to pull your
flow code from? [y/n] (y): y
? Is main the correct branch to pull your flow code from? [y/n] (y): y
? Is this a private repository? [y/n]: n

 Deployment 'pipe/first_deploy' successfully created with id
 '0f45657b-86d7-4141-a56a-e1ce47b90f1d'.
```

Deployments in the UI

The deployment lives on the server. See it in the UI.



Save deployment configuration to *prefect.yaml*

```
? Would you like to save configuration for this deployment for
faster deployments in the future? [y/n]: y

Deployment configuration saved to prefect.yaml! You can now deploy
using this deployment configuration with:

        $ prefect deploy -n first_deploy

You can also make changes to this deployment configuration by
making changes to the prefect.yaml file.
```

Recap of our setup

- Deployment & work pool created on Prefect Cloud
- Worker runs on local machine
- Worker polls Prefect Cloud, looking for scheduled work in the *my_pool* work pool
- Deployment configuration saved to *prefect.yaml*

Schedule a run - what happened?

- Running worker finds scheduled work in *my_pool* work pool.
- Worker and work pool are typed. *Local subprocess* in this case.
- Worker creates a local subprocess to kick off flow run.
- Flow code cloned from GitHub into temporary directory.
- Flow code runs.
- Metadata and logs sent to Prefect Cloud.
- Temporary directory deleted.

Deployment creation with prefect.yaml

prefect.yaml

```yaml
# Generic metadata about this project
name: pacc-2023
prefect-version: 2.10.18

# build section allows you to manage and build docker images
build: null

# push section allows you to manage if and how this project is
push: null

# pull section allows you to provide instructions for cloning
pull:
- prefect.deployments.steps.git_clone:
    repository: https://github.com/discdiver/pacc-2023.git
    branch: main
```


prefect.yaml

Configuration for creating deployments

- ***pull*** step (repository & branch): from git repo

prefect.yaml

- ***deployments***:

  Config for one or more deployments

Required keys:

- *name*
- *entrypoint*
- *work_pool -> name*

```yaml
deployments:

- name: deployment1
  entrypoint: 202/flows.py:pipe
  work_pool:
    name: local-work

- name: deployment2
  entrypoint: 202/flows2.py:pipe2
  work_pool:
    name: local-work
```


Can override steps above on per-deployment basis

```yaml
deployments:
  - name: prod-deployment
    entrypoint: 202/flows.py:pipe
    work_pool:
      name: prod-pool
    schedule:
      interval: 600
    pull:
      - prefect.deployments.steps.git_clone:
          repository: https://github.com/discdiver/pacc-london-2023.git
          branch: prod
          access_token: "{{prefect.blocks.secret.gh-secret}}"

  - name: staging-deployment
    entrypoint: 202/flows.py:pipe
    work_pool:
      name: staging-pool
    pull:
      - prefect.deployments.steps.git_clone:
          repository: https://github.com/discdiver/pacc-london-2023.git
          branch: staging
```


Re-deploy a deployment

Requires a *prefect.yaml* file

prefect deploy

```
? Would you like to use an existing deployment configuration? [Use arrows to move; enter to select; n to select none]
```

	Name	Entrypoint	Description
>	first_deploy	104/flows.py:pipe	

Deploy multiple deployments at once

Deploy all deployments in a *prefect.yaml* file:

prefect deploy --all

prefect deploy

If choose *docker* typed work pool you will be asked docker-related questions

```
┌────┬──────────────┬──────────────────────┬─────────────┐
│    │ Work Pool Name │ Infrastructure Type │ Description │
├────┼──────────────┼──────────────────────┼─────────────┤
│ >  │ docker-pool  │ docker               │             │
│    │ my-pool      │ process              │             │
└────┴──────────────┴──────────────────────┴─────────────┘
? Would you like to build a custom Docker image for this deployment? [y/n]
(n): 
```


Method 1: *prefect deploy*

Use the defaults for the work pool

OR

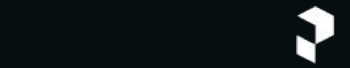
Build a custom Docker image with flow code

- Push image to a Docker registry
  - Use existing Dockerfile
  - Auto-includes packages in *requirements.txt*

Follow the prompts. 🙂

Resulting *prefect.yaml*

```yaml
- name: dock-interact
  version:
  tags: []
  description:
  entrypoint: 104/flows.py:pipe
  parameters: {}
  work_pool:
    name: docker-pool
    work_queue_name:
    job_variables:
      image: '{{ build-image.image }}'
  schedule:
  build:
  - prefect_docker.deployments.steps.build_docker_image:
      requires: prefect-docker>=0.3.1
      id: build-image
      dockerfile: auto
      image_name: discdiver/dock-interact
      tag: 0.0.1
```

CI/CD with GitHub Actions

GitHub Actions with deployments

- CI/CD - when you push code or make a PR automatically take an action
- Pre-built Github Action to deploy a Prefect deployment
- github.com/marketplace/actions/deploy-a-prefect-flow

GitHub Action

```yaml
name: Deploy a Prefect flow
on:
  push:
    branches:
      - main
jobs:
  deploy_flow:
    runs-on: ubuntu-latest
    steps:
      - uses: checkout@v3

      - uses: actions/setup-python@v4
        with:
          python-version: '3.10'

      - name: Run Prefect Deploy
        uses: PrefectHQ/actions-prefect-deploy@v1
        with:
          prefect-api-key: ${{ secrets.PREFECT_API_KEY }}
          prefect-workspace: ${{ secrets.PREFECT_WORKSPACE }}
          requirements-file-path: ./examples/simple/requirements.txt
          entrypoint: ./examples/simple/flow.py:call_api
          additional-args: --cron '30 19 * * 0'
```


Helm Chart

Prefect Helm Chart for K8s

Provides a variety of functionality

Creating workers is a popular use case.

See more in the docs:
github.com/PrefectHQ/prefect-helm/tree/main/charts/prefect-worker

Terraform provider

Prefect Cloud Terraform Provider

registry.terraform.io/providers/PrefectHQ/prefect/latest/docs

PACC

Prefect Associate Certification Course

PREFECT