GitHub View the Code

Orchestrate a Data Science Project in Python With Prefect

Motivation

As a data scientist, why should you care about optimizing your data science workflow? Let's start with an example of a basic data science project.

Imagine you were working with an Iris dataset. You started with building functions to process your data.

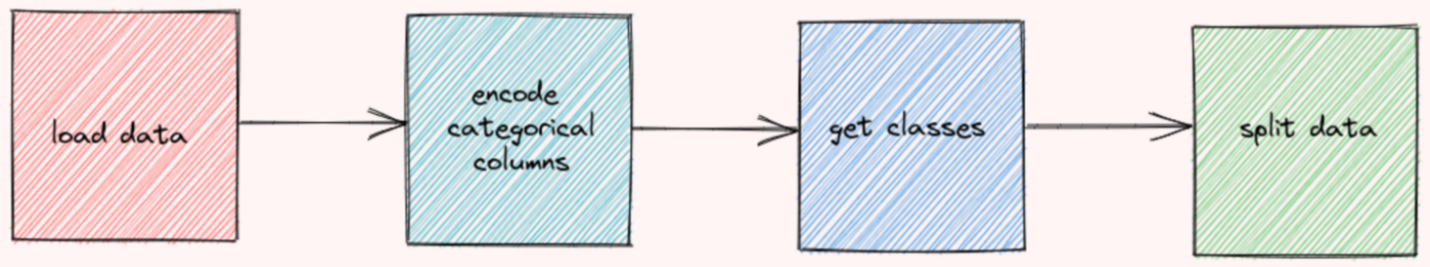
```
from typing import Any, Dict, List
import pandas as pd

def load_data(path: str) -> pd.DataFrame:
    ...

def get_classes(data: pd.DataFrame, target_col: str) ->
List[str]:
    ...
```

```
def encode_categorical_columns(data: pd.DataFrame,
target_col: str) -> pd.DataFrame:
    ...

def split_data(data: pd.DataFrame, test_data_ratio:
float, classes: list) -> Dict[str, Any]:
    ...
```

After defining the functions, you execute them.

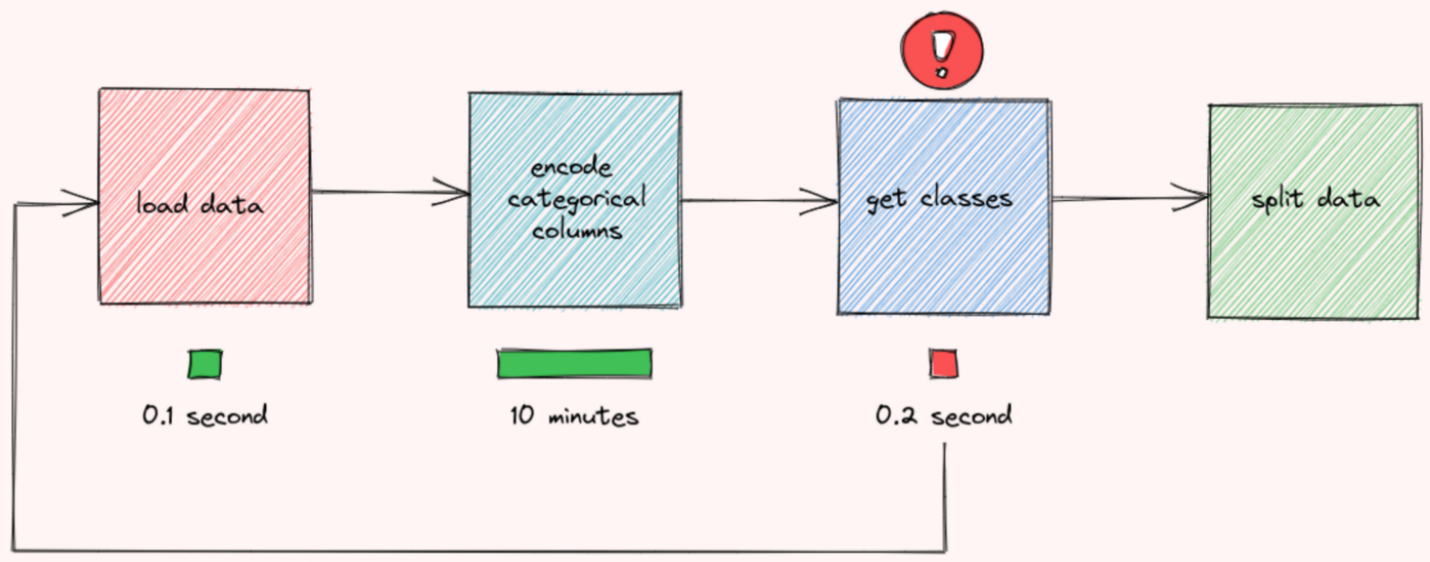
```
# Define parameters
target_col = 'species'
test_data_ratio = 0.2

# Run functions
data = load_data(path="data/raw/iris.csv")
categorical_columns =
encode_categorical_columns(data=data,
target_col=target_col)
classes = get_classes(data=data, target_col=target_col)
train_test_dict = split_data(data=categorical_columns,

 test_data_ratio=test_data_ratio,
                              classes=classes)
```

Your code ran fine, and you saw nothing wrong with the output, so you think the workflow is good enough. However, there can be many disadvantages with a linear workflow like above.

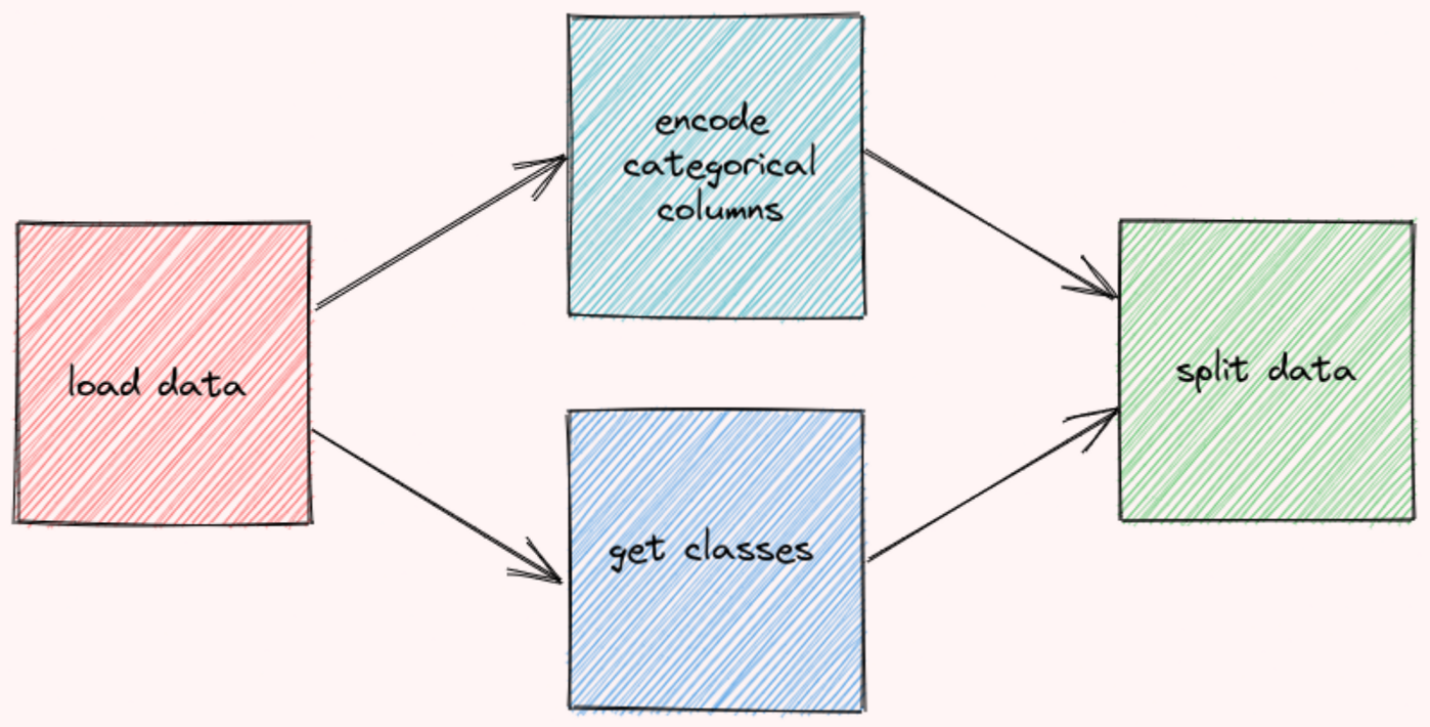


The disadvantages are:

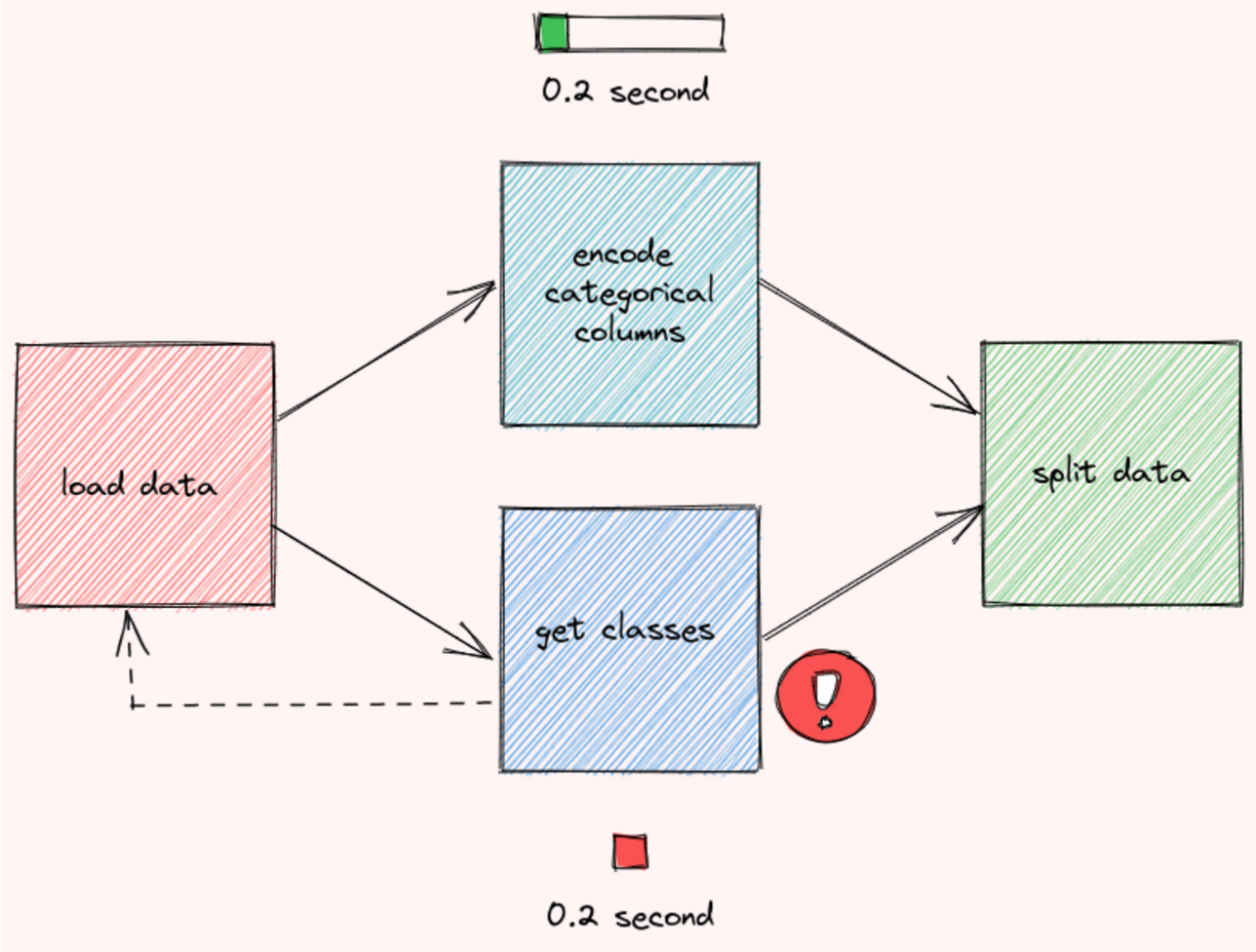
- If there is an error in the function `get_classes` , the output produced by the function `encode_categorical_columns` will be lost, and the workflow will need to start from the beginning. This can be frustrating if it takes a long time to execute the function `encode_categorical_columns` .



- Since the functions `encode_categorical_columns` and `get_classes` are not dependent on each other, they can be executed at the same time to save time:



Running the functions this way can also prevent wasting unnecessary time on functions that don't work. If there is an error in the function `get_classes` , the workflow will restart right away without waiting for the function `encode_categorical_columns` to finish.



Now, you might agree with me that it is important to optimize the workflow of different functions. However, it can be a lot of works to manually manage the workflow.

Is there a way that you can **automatically optimize the workflow** by adding only several lines of code? That is when Prefect comes in handy.

What is Prefect?

Prefect is an open-sourced framework to build workflows in Python. Prefect makes it easy to build, run, and monitor data pipelines at scale.

To install Prefect, type:

```
pip install prefect
```

Build Your Workflow with Prefect

To learn how Prefect works, let's encapsulate the workflow introduced above with Prefect.

First Step — Create Tasks

A `Task` is a discrete action in a Prefect flow. Start with turning the functions defined above into tasks using the decorator `prefect.task` :

```
from prefect import task
from typing import Any, Dict, List
import pandas as pd

@task
def load_data(path: str) -> pd.DataFrame:
    ...

@task
def get_classes(data: pd.DataFrame, target_col: str) ->
List[str]:
    ...


@task
```

```
def encode_categorical_columns(data: pd.DataFrame,
target_col: str) -> pd.DataFrame:
    ...

@task
def split_data(data: pd.DataFrame, test_data_ratio:
float, classes: list) -> Dict[str, Any]:
    ...
```

Second Step — Create a Flow

A `Flow` represents the entire workflow by managing the dependencies between tasks. To create a flow, simply insert the code to run your functions inside the `with Flow(...)` context manager.

```
from prefect import task, Flow

with Flow("data-engineer") as flow:

    # Define parameters
    target_col = 'species'
    test_data_ratio = 0.2

    # Define tasks
    data = load_data(path="data/raw/iris.csv")
    classes = get_classes(data=data,
target_col=target_col)
    ...
```

Note that none of these tasks are executed when running the code above. Prefect allows you to either run the flow right away or schedule for later.

Let's try to execute the flow right away using `flow.run()` :

```
with Flow("data-engineer") as flow:
  # Define your flow here
  ...

flow.run()
```

Running the code above will give you the output similar to this:

```
└── 15:49:46 | INFO    | Beginning Flow run for 'data-
engineer'
└── 15:49:46 | INFO    | Task 'target_col': Starting
task run...
└── 15:49:46 | INFO    | Task 'target_col': Finished
task run for task with final state: 'Success'
└── 15:49:46 | INFO    | Task 'test_data_ratio':
Starting task run...
└── 15:49:47 | INFO    | Task 'test_data_ratio':
Finished task run for task with final state: 'Success'
└── 15:49:47 | INFO    | Task 'load_data': Starting
task run...
└── 15:49:47 | INFO    | Task 'load_data': Finished
task run for task with final state: 'Success'
└── 15:49:47 | INFO    | Task
'encode_categorical_columns': Starting task run...
└── 15:49:47 | INFO    | Task
'encode_categorical_columns': Finished task run for
task with final state: 'Success'
```

```
└── 15:49:47 | INFO    | Task 'get_classes': Starting
task run...
└── 15:49:47 | INFO    | Task 'get_classes': Finished
task run for task with final state: 'Success'
└── 15:49:47 | INFO    | Task 'split_data': Starting
task run...
└── 15:49:47 | INFO    | Task 'split_data': Finished
task run for task with final state: 'Success'
└── 15:49:47 | INFO    | Flow run SUCCESS: all
reference tasks succeeded
Flow run succeeded!
```

To understand the workflow created by Prefect, let's visualize the entire workflow.

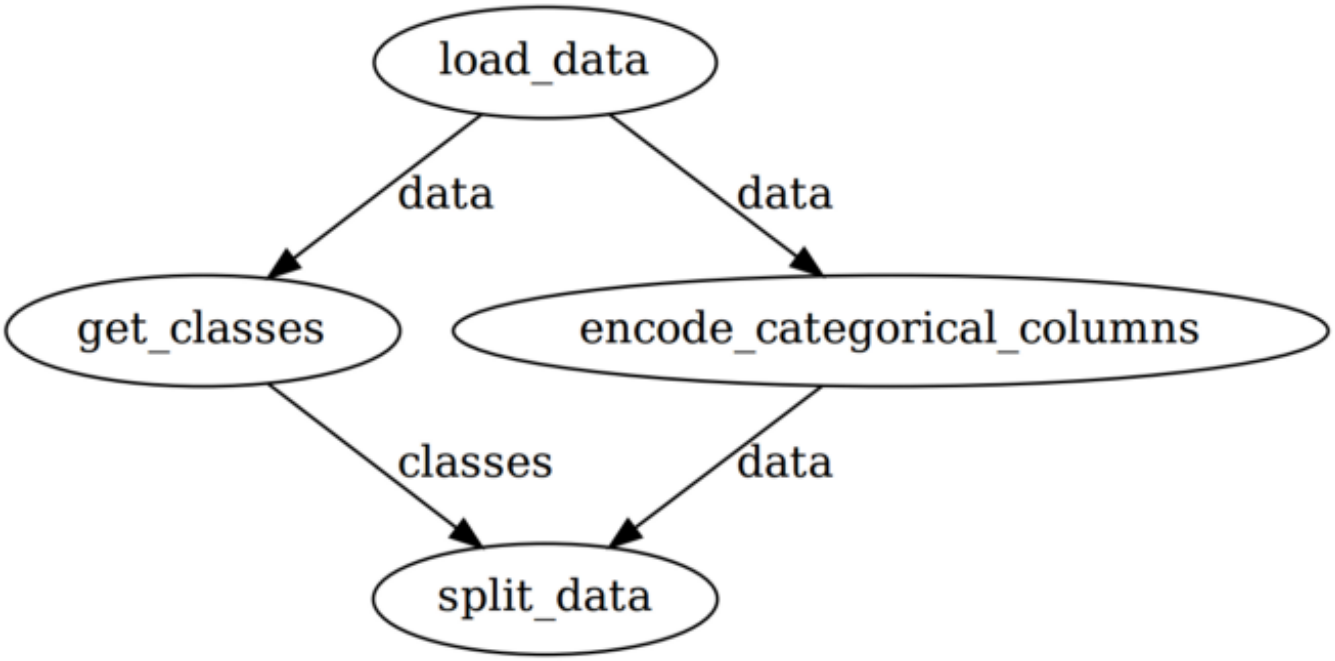
Start with installing `prefect[viz]` :

```
pip install "prefect[viz]"
```

Then add the method `visualize` to the code:

```
flow.visualize()
```

And you should see the visualization of the `data-engineer` workflow like below!



Note that Prefect automatically manages the orders of execution among tasks so that the workflow is optimized. This is pretty cool for a few additional pieces of code!

Third step — Add Parameters

If you find yourself frequently experimenting with different values of one variable, it's ideal to turn that variable into a `Parameter`.

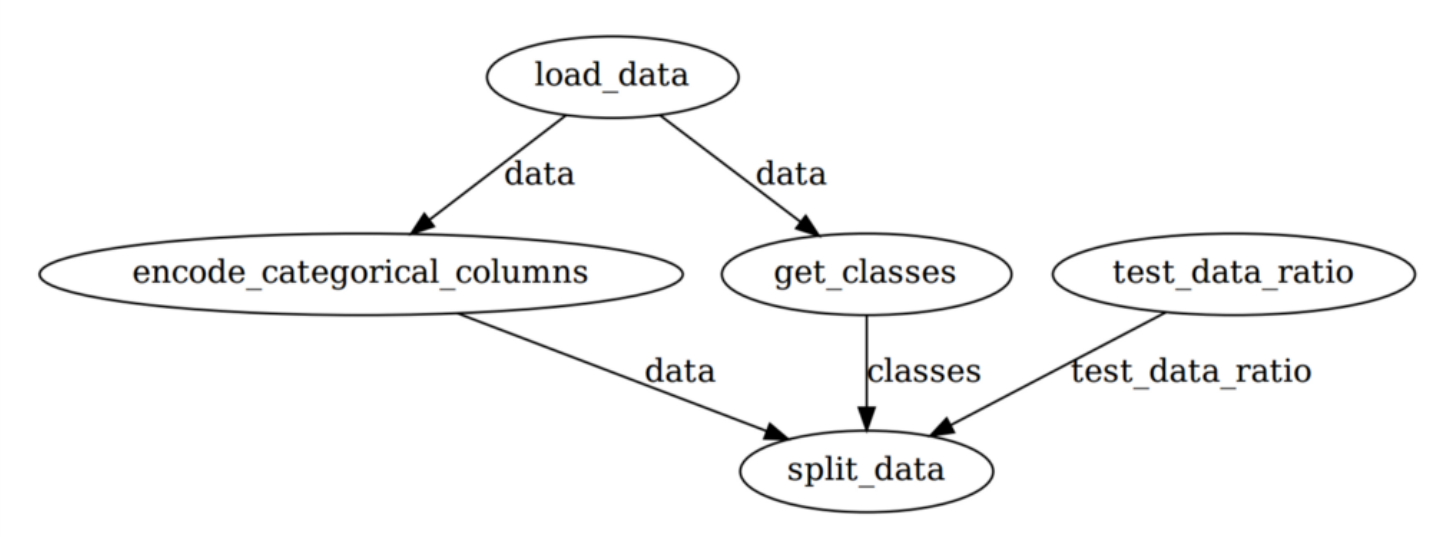
```
test_data_ratio = 0.2
train_test_dict = split_data(data=categorical_columns,

 test_data_ratio=test_data_ratio,
                              classes=classes)
```

You can consider a `Parameter` as a `Task` , except that it can receive user inputs whenever a flow is run. To turn a variable into a parameter, simply use `task.Parameter` .

```
from prefect import task, Flow, Parameter

test_data_ratio = Parameter("test_data_ratio",
default=0.2)

train_test_dict = split_data(data=categorical_columns,

 test_data_ratio=test_data_ratio,
                              classes=classes)
```

The first argument of `Parameter` specifies the name of the parameter. `default` is an optional argument that specifies the default value of the parameter.

Running `flow.visualize` again will give us an output like below:



You can overwrite the default parameter for each run by:

- adding the argument `parameters` to `flow.run()` :

```
$ flow.run(parameters={'test_data_ratio': 0.3})
```

- or using Prefect CLI:

```
$ prefect run -p data_engineering.py --param
test_data_ratio=0.2
```

You can also change parameters for each run using Prefect Cloud, which will be introduced in the next section.

Monitor Your Workflow

Overview

Prefect also allows you to monitor your workflow in Prefect Cloud. Follow this instruction to install relevant dependencies for Prefect Cloud.

After all of the dependencies are installed and set up, start with creating a project on Prefect by running:

```
$ prefect create project "Iris Project"
```

Next, start a local agent to deploy our flows locally on a single machine:

```
$ prefect agent local start
```

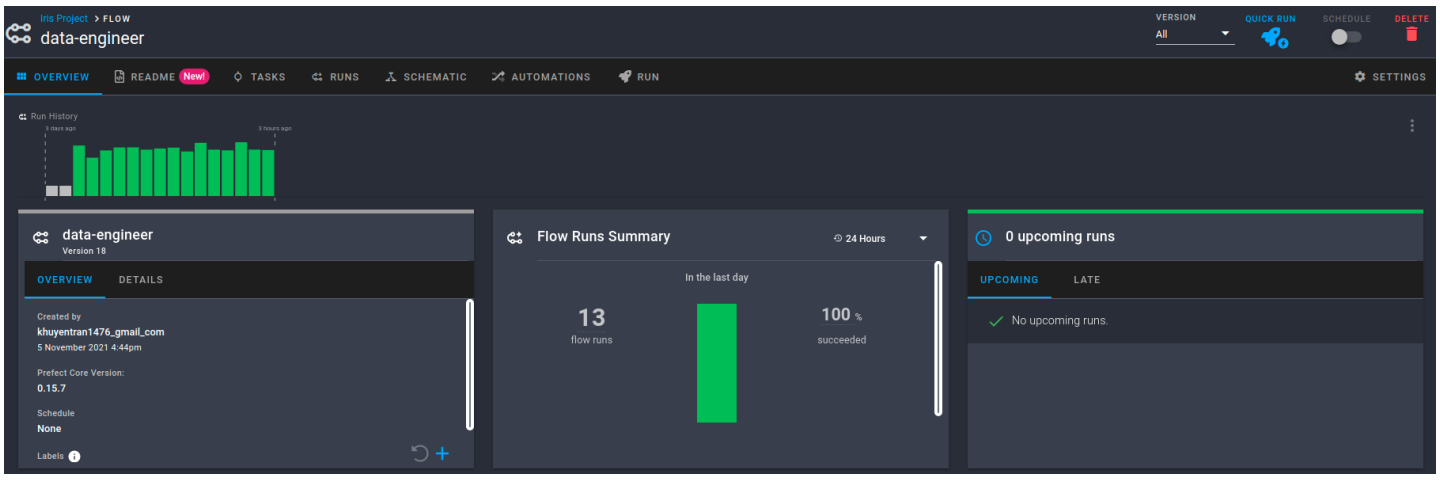
Then add:

```
flow.register(project_name="Iris Project")
```

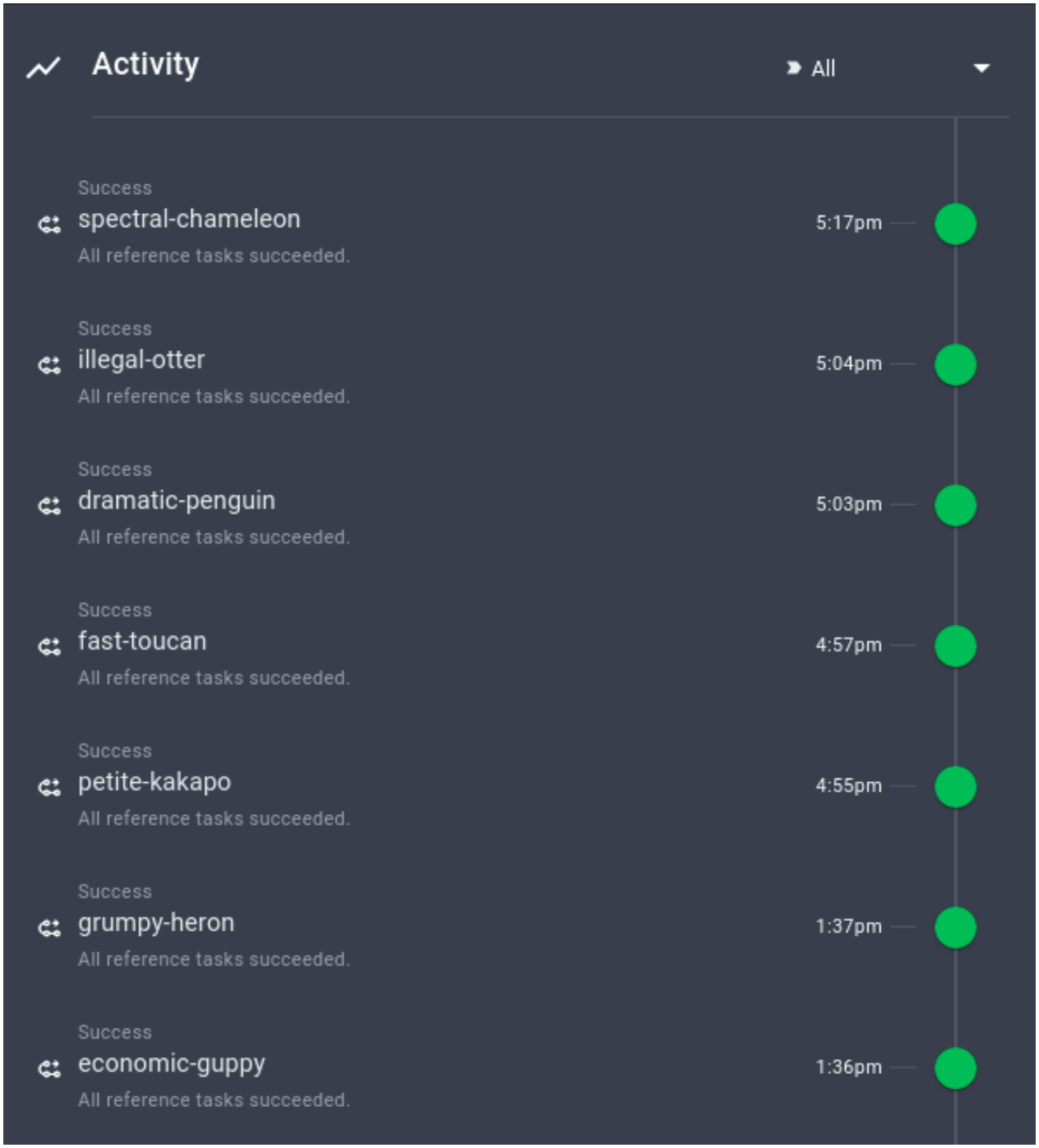
… at the end of your file. You should see something similar to the below:

```
Flow URL: https://cloud.prefect.io/khuyentran1476-
gmail-com-s-account/flow/dba26bea-8827-4db4-9988-
3289f6cb662f
 └── ID: 2944dc36-cdac-4079-8497-be4ec5594785
 └── Project: Iris Project
 └── Labels: ['khuyen-Precision-7740']
```

Click the URL in the output, and you will be redirected to an Overview page. The Overview page shows the version of your flow, when it is created, the flow's run history, and its runs summary.



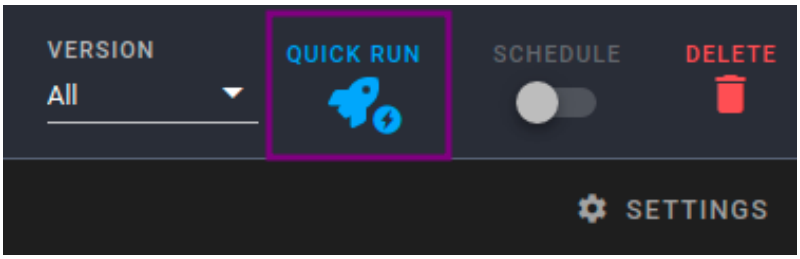
You can also view the summary of other runs, when they are executed, and their configurations.



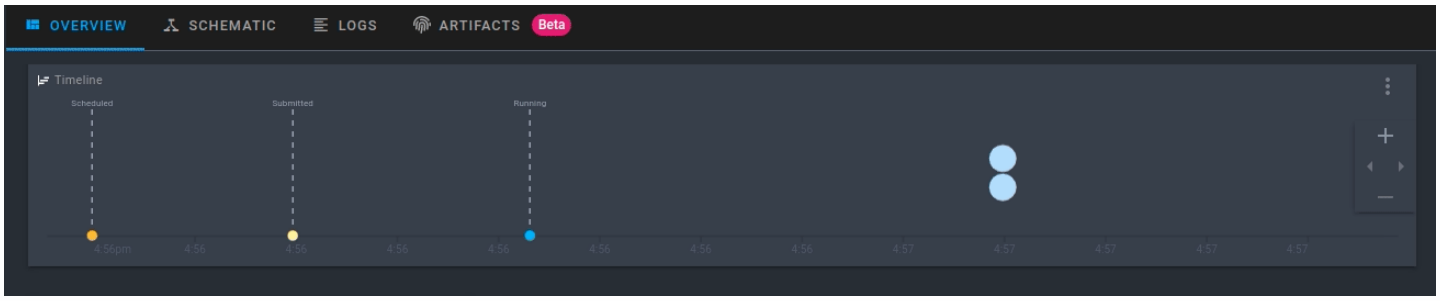
It is pretty cool how these important pieces of information are automatically tracked by Perfect!

Run the Workflow with Default Parameters

Note that the workflow is registered to Prefect Cloud, but it is not executed yet. To execute the workflow with the default parameters, click Quick Run in the top right corner.

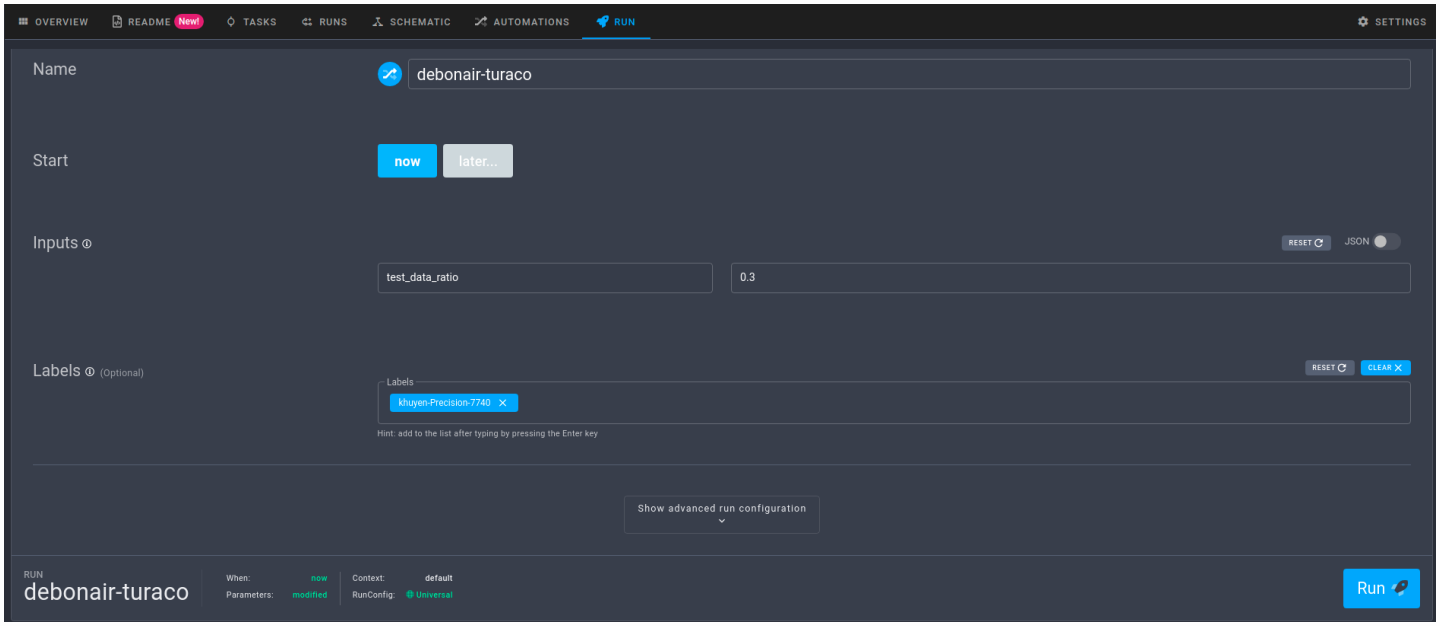


Click the run that is created. Now you will be able to see the activity for your new flow run in real-time!



Run the Workflow with Custom Parameters

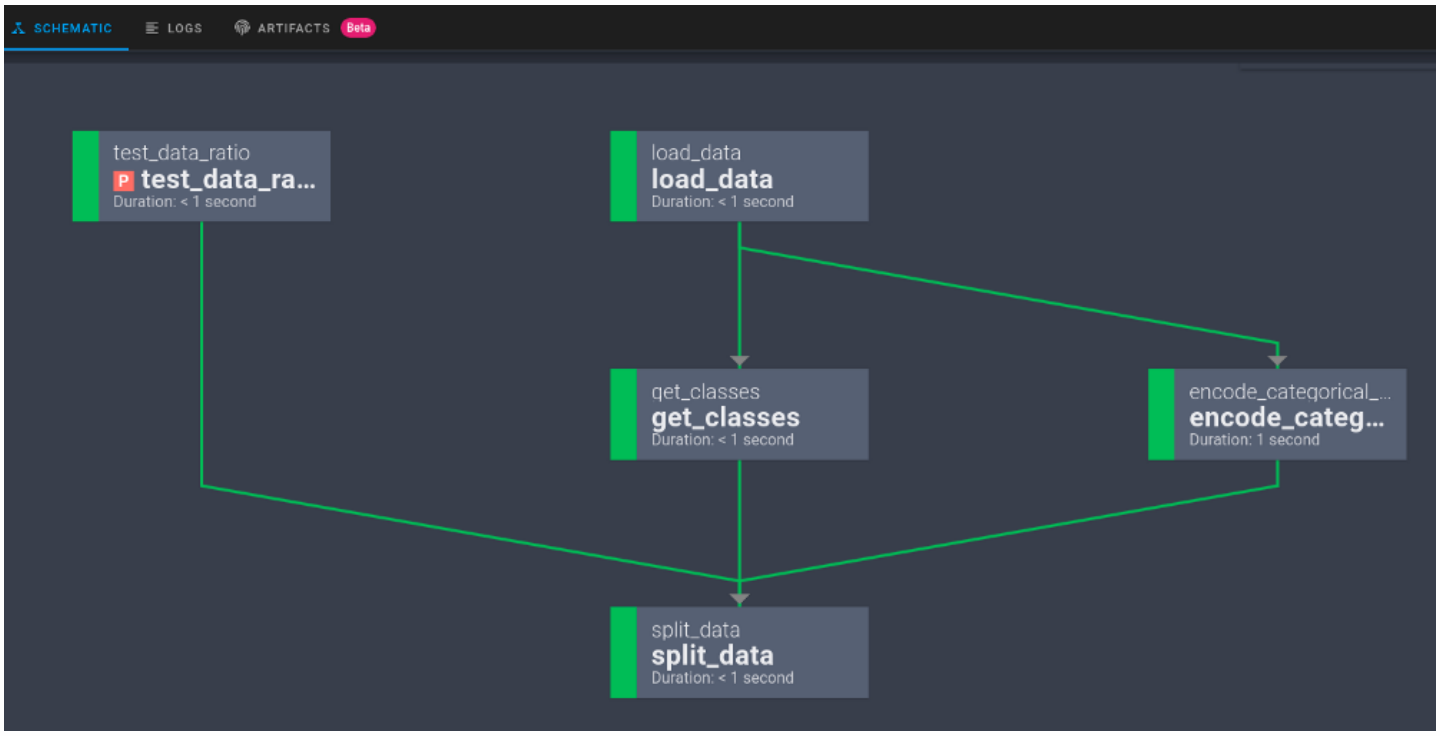
To run the workflow with custom parameters, click the Run tab, then change the parameters under Inputs.



When you are satisfied with the parameters, simply click the Run button to start the run.

View the Graph of the Workflow

Clicking Schematic will give you the graph of the entire workflow.



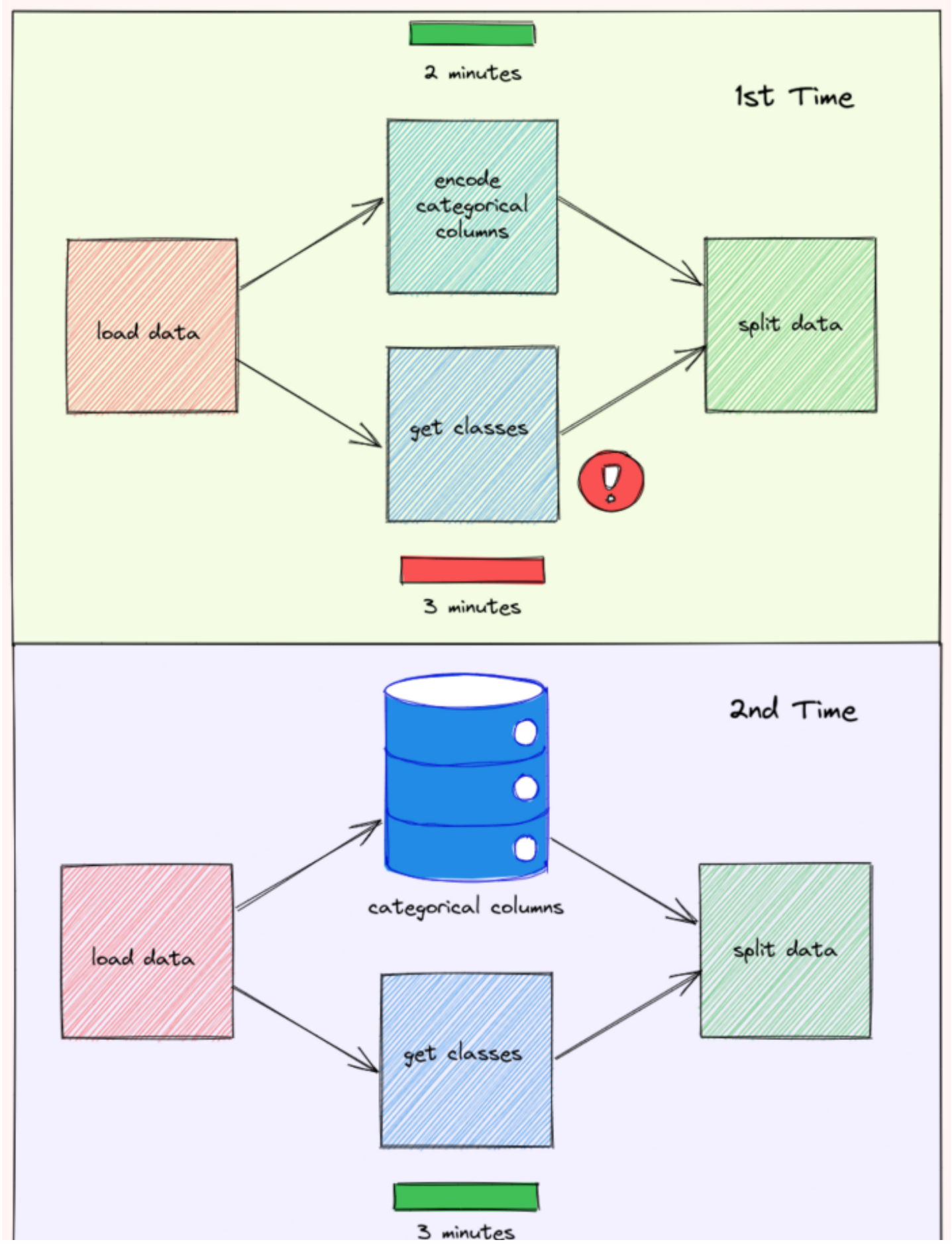
Other Features

Besides some basic features mentioned above, Prefect also provides some other cool features that will significantly increase the efficiency of your workflow.

Input Caching

Remember the problem we mentioned at the beginning of the section? Normally, if the function `get_classes` fails, the data created by the function `encode_categorical_columns` will be discarded and the entire workflow needs to start from the beginning.

However, with Prefect, the output of `encode_categorical_columns` is stored. Next time when the workflow is rerun, the output of `encode_categorical_columns` will be used by the next task **without rerunning** the task `encode_categorical_columns` .


2 minutes
1st Time
encode categorical columns
load data
split data
get classes
3 minutes
2nd Time
categorical columns
load data
split data
get classes
3 minutes

This can result in a huge decrease in the time it takes to run the workflow.

Persist Output

Sometimes, you might want to export your task's data to an external location. This can be done by inserting to the task function the code to save the data.

```
def split_data(data: pd.DataFrame, test_data_ratio:
float, classes: list) -> Dict[str, Any]:

  X_train, X_test, y_train, y_test = ...

  import pickle
  pickle.save(...)
```

However, doing that will make it difficult to test the function.

Prefect makes it easy to save the output of a task for each run by:

- setting the checkpoint to `True`

```
$ export PREFECT__FLOWS__CHECKPOINTING=true
```

- and adding `result = LocalResult(dir=...))` to the decorator `@task` .

```
from prefect.engine.results import LocalResult

@task(result = LocalResult(dir='data/processed'))
def split_data(data: pd.DataFrame, test_data_ratio:
float, classes: list) -> Dict[str, Any]:
    """Task for splitting the classical Iris data set
into training and test
    sets, each split into features and labels.
    """
    X_train, X_test, y_train, y_test = ...

    return dict(
        train_x=X_train,
        train_y=y_train,
        test_x=X_test,
        test_y=y_test,
```

Now the output of the task `split_data` will be saved to the directory `data/processed` ! The name will look something similar to this:

```
prefect-result-2021-11-06t15-37-29-605869-00-00
```

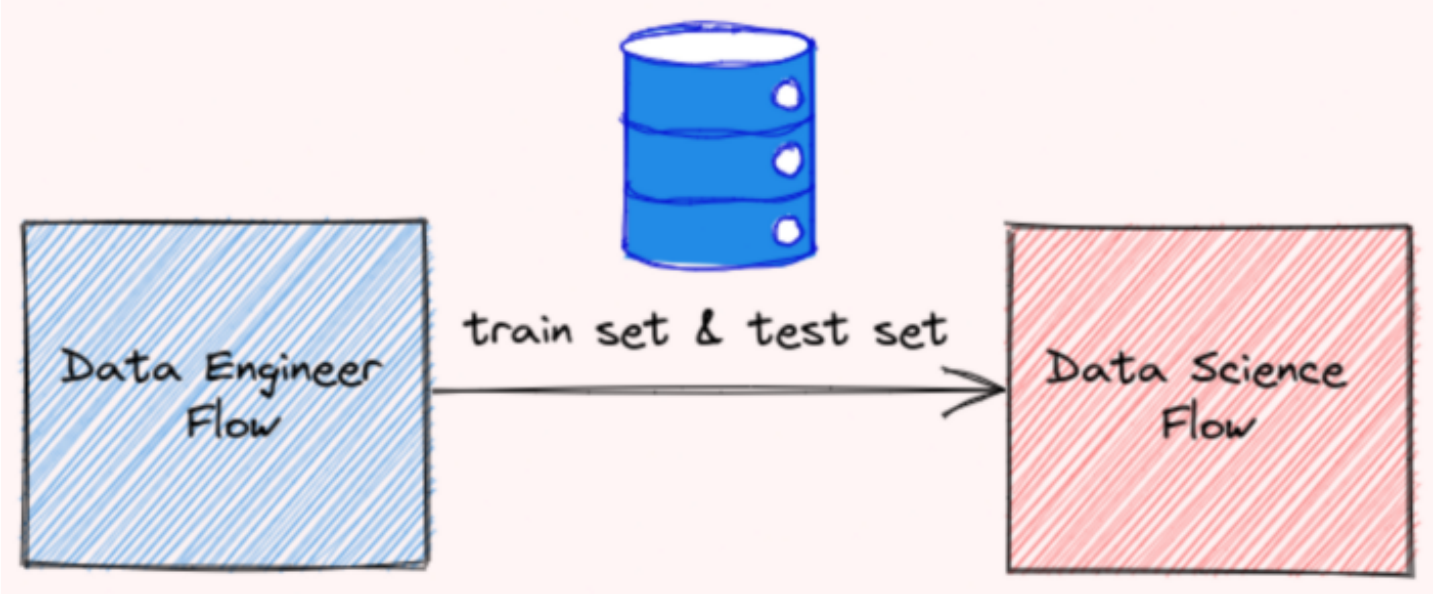
If you want to customize the name of your file, you can add the argument `target` to `@task` :

```
from prefect.engine.results import LocalResult

@task(target="
{date:%a_%b_%d_%Y_%H:%M:%S}/{task_name}_output",
      result = LocalResult(dir='data/processed'))
def split_data(data: pd.DataFrame, test_data_ratio:
float, classes: list) -> Dict[str, Any]:
    """Task for splitting the classical Iris data set
into training and test
    sets, each split into features and labels.
    """
    ...
```

Prefect also provides other `Result` classes such as `GCSResult` and `S3Result` . You can check out API docs for results here.

Use Output of Another Flow for the Current Flow

If you are working with multiple flows, for example, `data-engineer` flow and `data-science` flow, you might want to use the output of the `data-engineer` flow for the `data-science` flow.



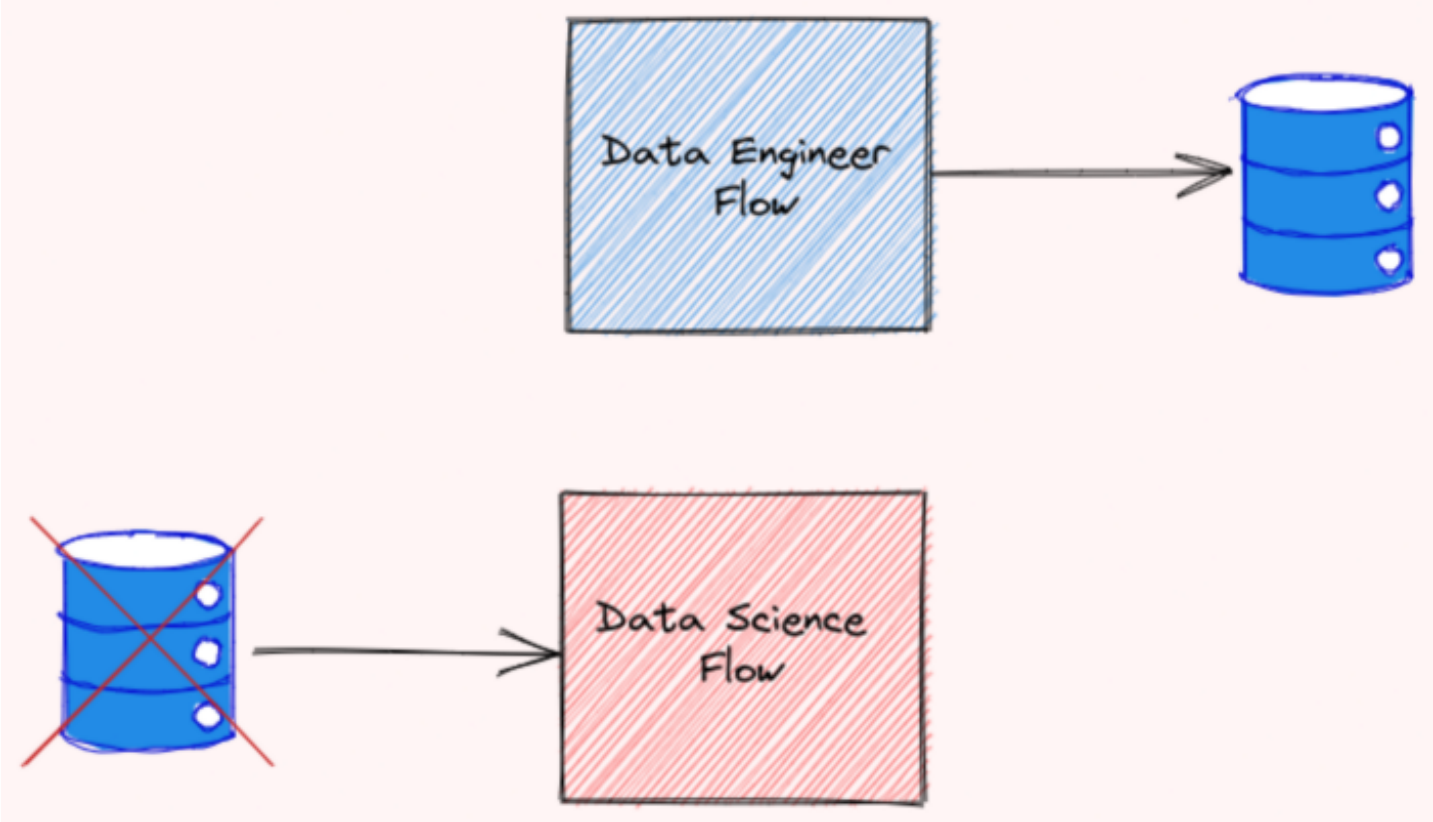
After saving the output of your `data-engineer` flow as a file, you can read that file using the `read` method:

```
from prefect.engine.results import LocalResult

train_test_dict =
LocalResult(dir=...).read(location=...).value
```

Connect Dependent Flows

Imagine this scenario: You created two flows that depend on each other. The flow `data-engineer` needs to be executed before the flow `data-science`

Somebody who looked at your workflow didn't understand the relationship between these two flows. As a result, they executed the flow `data-science` and the flow `data-engineer` at the same time and encountered an error!



To prevent this from happening, we should specify the relationship between flows. Luckily, Prefect makes it easier for us to do so.

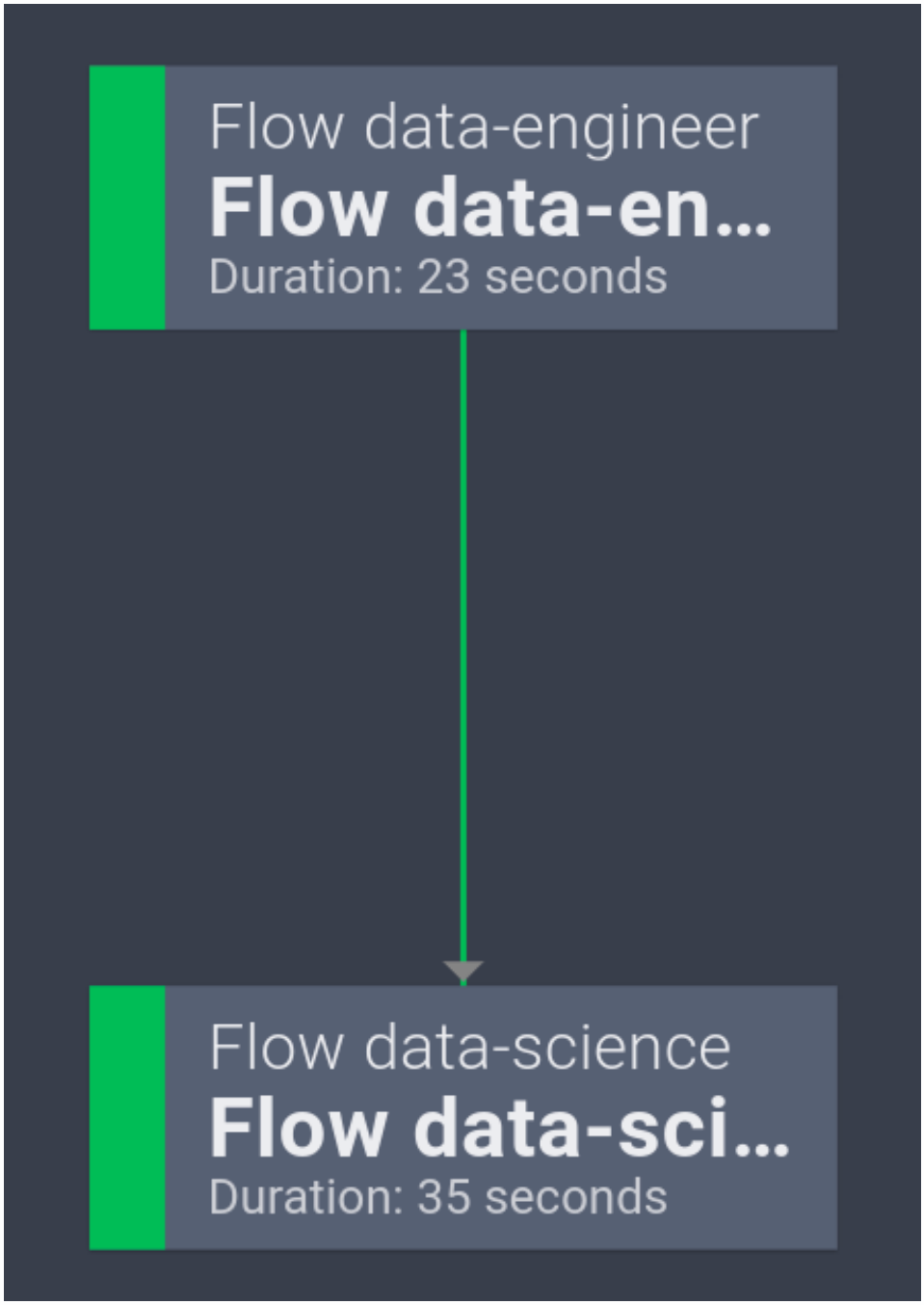
Start with grabbing two different flows using `StartFlowRun` . Add `wait=True` to the argument so that the downstream flow is executed only after the upstream flow finishes executing.

```
from prefect import Flow
from prefect.tasks.prefect import StartFlowRun

data_engineering_flow = StartFlowRun(
  flow_name="data-engineer",
  project_name='Iris Project',
  wait=True)

data_science_flow = StartFlowRun(
  flow_name="data-science",
  project_name='Iris Project',
  wait=True)
```

Next, calling `data_science_flow` under the `with Flow(...)` context manager. Use `upstream_tasks` to specify the tasks/flows that will be executed before the `data-science` flow is executed.

```
with Flow("main-flow") as flow:
    result = data_science_flow(upstream_tasks=
[data_engineering_flow])

flow.run()
```

Now the two flows are connected like below:



Pretty cool!

Schedule Your Flow

Prefect also makes it seamless to execute a flow at a certain time or at a certain interval.

For example, to run a flow every 1 minute, you can initiate the class `IntervalSchedule` and add `schedule` to the `with Flow(...)` context manager:

```
from prefect.schedules import IntervalSchedule

schedule = IntervalSchedule(
    start_date=datetime.utcnow() +
timedelta(seconds=1),
    interval=timedelta(minutes=1),
)

data_engineering_flow = ...
data_science_flow = ...


with Flow("main-flow", schedule=schedule) as flow:
    data_science = data_science_flow(upstream_tasks=
[data_engineering_flow])
```

Now your flow will be rerun every 1 minute!

Learn more about different ways to schedule your flow here.

Logging

You can log the print statements within a task by simply adding `log_stdout=True` to `@task` :

```
@task(log_stdout=True)
def report_accuracy(predictions: np.ndarray, test_y:
pd.DataFrame) -> None:

  target = ...
  accuracy = ...

  print(f"Model accuracy on test set: {round(accuracy *
100, 2)}")
```

And you should see an output like below when executing the task:

```
[2021-11-06 11:41:16-0500] INFO - prefect.TaskRunner |
Model accuracy on test set: 93.33
```