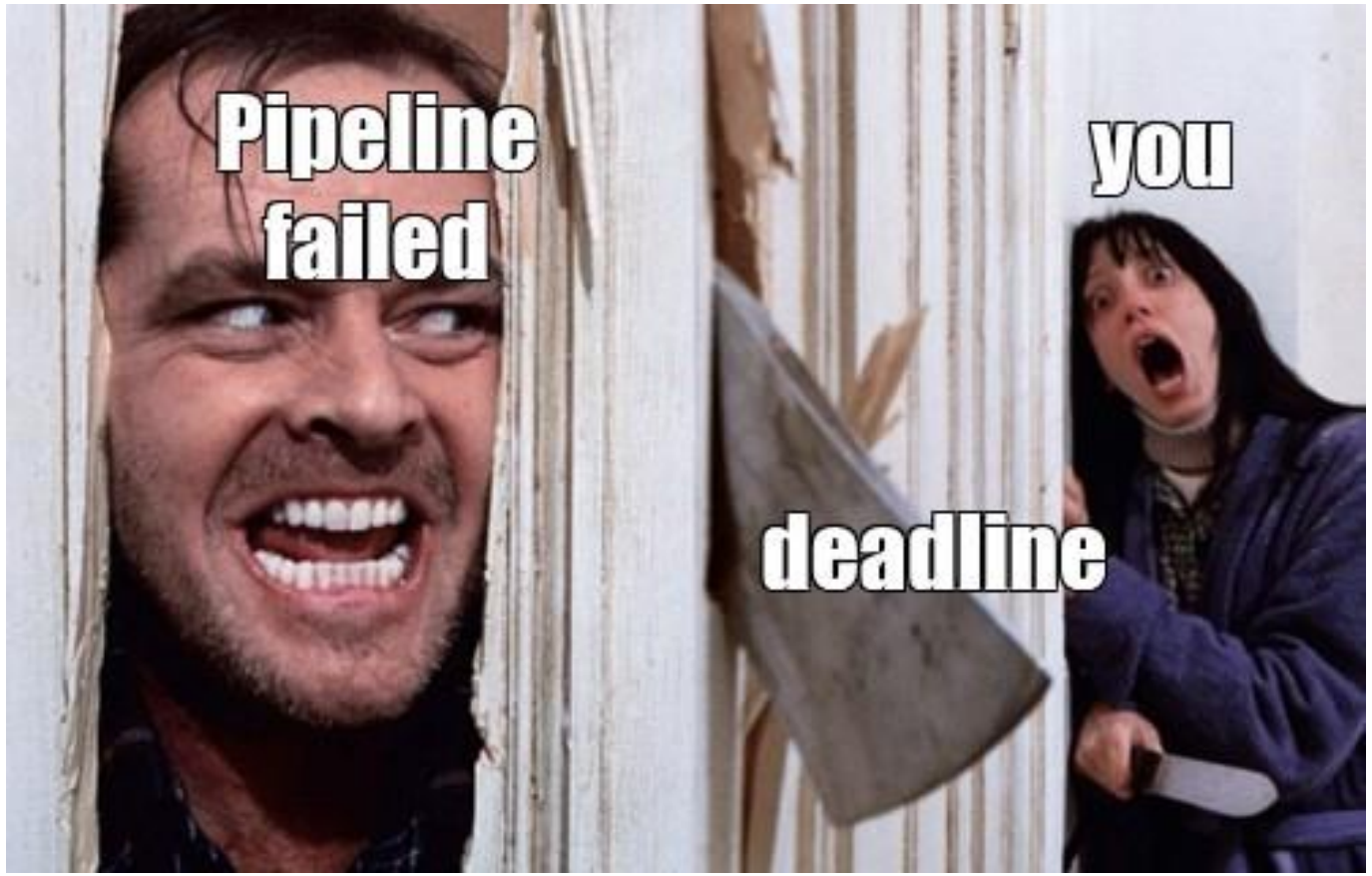
Автоматизация Part 2
Пайплайнизация



Борисенко Глеб

ФТиАД 2022

О чем болтали в прошлый раз?

- CI/CD
- Типовый джобы
- Gitlab CI

Что забыл рассказать по Gitlab CI

- Anchors and hidden jobs

```yaml
.job_template: &job_configuration  # Hidden yaml configuration that defines an anchor named 'job_configuration'
  image: ruby:2.6
  services:
    - postgres
    - redis

test1:
  <<: *job_configuration           # Merge the contents of the 'job_configuration' alias
  script:
    - test1 project

test2:
  <<: *job_configuration           # Merge the contents of the 'job_configuration' alias
  script:
    - test2 project
```

О чем поговорим сегодня?

- Что такое пайплайн и оркестрация, зачем все это нужно
- Примеры таких пайплайнов
- Инструмент для построения пайплайнов Dagster
- Особенности построения пайплайнов

Что такое пайплайн?

- Пайплайн – просто последовательность действий
- По сути, ваш код можно представить в виде пайплайна из команд
- Мы будем говорить о более высокоуровневых шагах
- И касаться будем пайплайнов над данными

А что такое пайплайн над данными?

- Пайплайн над данными – пайплайн, на входе которого источники данных, над которыми происходят некоторые преобразования, а на выходе (или даже в процессе) новые данные кладутся в хранилища.
- Пайплайны могут запускаться:
  - Автоматически по триггеру или расписанию
  - Руками :)
- У каждого запуска пайплайна также есть своя конфигурация

Как это вообще касается нас, DS-ов?

- Вы точно будете пользоваться пайплайнами
- И с высокой вероятностью будете разрабатывать или помогать разрабатывать пайплайны

Зачем нужны пайплайны?

- Это удобно:
  - Мы заполняем конфигурацию, а пайплайн (будто скрипт) выполняет магию, по итогу которой – у нас новые нужные нам данные
  - Их можно выполнять автоматически. Появились новые данные -> они сразу прошли какую-то обработку -> у нас нужные нам данные без каких-либо действий
  - В этих пайплайнах можно использовать модельки :)
- Повышается контроль за потоками данных, а это влечет в свою очередь кучу плюсов разных

Типовые варианты пайплайнов

- Переобучение модели каждую неделю на новых данных
- Получение новых нужных «тяжелых» данных, дернув пайплайн
- Автоматическая обработка всех новых данных и обновление витрин
- Эта обработка из пункта выше может включать как просто преобразования данных, так и применение модели над ними
- Batch инференс по запросу

Как создать пайплайн?

- Чаще всего пайплайны пишутся на Python в одном из оркестраторов:
  - Airflow
  - Dagster
  - Kubeflow
  - Etc.
- По сути, вы пишете функции, определяете их как этапы ваших пайплайнов, связываете их, дополняете технической информацией – и voilà, пайплайн готов!

Ага, а что такое оркестратор?

- Пайплайн – Последовательность шагов
- Каждый шаг надо исполнять где-то
- Оркестратор – та штука, которая берет шаги пайплайна, и запускает их в определенном месте в заданной последовательности. Также эта штука может запускать их по расписанию и по триггеру.

И как пайплайн выполняется?

- Пайплайн – по сути DAG (directed acyclic graph), в некоторых оркестраторах это синоним пайплайна. Каждый узел – этап пайплайна, ребро – связь между этапами.
- Каждый этап – отдельный вычислительный процесс, и чаще всего настраивают оркестратор так, что каждый этап выполняется в отдельном контейнере в Kubernetes.
- Передача данных между пайплайнами настривается по-разному. У нас, например, данные передаются через временный бакет в s3.
- И у каждого запуска пайплайна есть своя конфигурация

Dagster

- Как вы уже поняли, один из оркестраторов
- Есть свои фичи по сравнению с главным конкурентов Airflow
- Последнее время появляется в вакансиях все чаще

Dagster: базовые термины

- Op (operation) – этап вашего пайплайна
- Graph – собственно, пайплайн, по сути функция, в которой связываются ваши этапы
- Job – граф + конфиг (ресурсы); можно сказать, граф с технической информацией
- Asset – появилось относительно недавно, замена Op; по сути тот же Op, только результатом обязательно является артефакт какой-нибудь.

Op

```python
from dagster import op


@op
def return_annotation_op() -> int:
    return 5
```

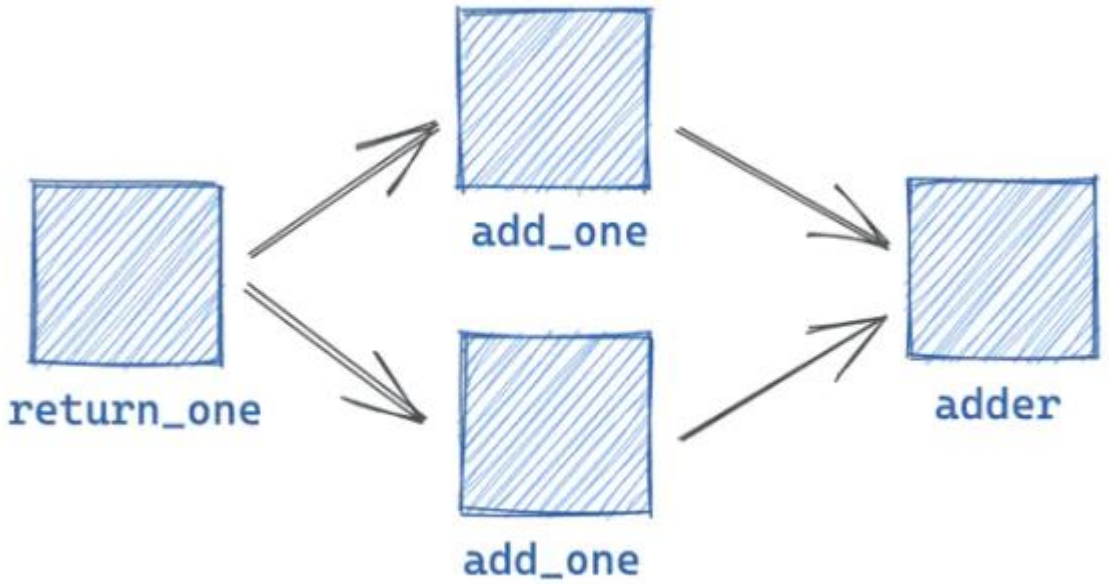
```python
MyDagsterType = DagsterType(
    type_check_fn=lambda _, value: value % 2 == 0, name="MyDagsterType"
)


@op(ins={"abc": In(dagster_type=MyDagsterType)})
def my_typed_input_op(abc):
    pass
```

```python
from dagster import op
from typing import Tuple


@op(out={"int_output": Out(), "str_output": Out()})
def my_multiple_output_annotation_op() -> Tuple[int, str]:
    return (5, "foo")
```

```python
@op(config_schema={"name": str})
def context_op(context):
    name = context.op_config["name"]
    context.log.info(f"My name is {name}")
```

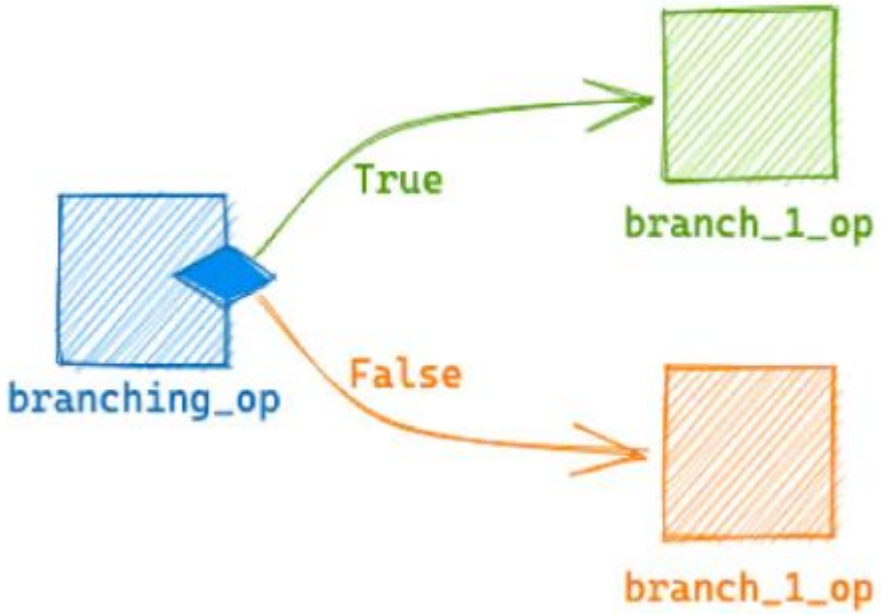
Graph: простой

```python
from dagster import graph, op


@op
def return_one(context) -> int:
    return 1


@op
def add_one(context, number: int):
    return number + 1


@op
def adder(context, a: int, b: int) -> int:
    return a + b


@graph
def inputs_and_outputs():
    value = return_one()
    a = add_one(value)
    b = add_one(value)
    adder(a, b)
```



Graph: с условием

```python
import random

from dagster import Out, Output, graph, op


@op(out={"branch_1": Out(is_required=False), "branch_2": Out(is_required=False)})
def branching_op():
    num = random.randint(0, 1)
    if num == 0:
        yield Output(1, "branch_1")
    else:
        yield Output(2, "branch_2")


@op
def branch_1_op(_input):
    pass


@op
def branch_2_op(_input):
    pass


@graph
def branching():
    branch_1, branch_2 = branching_op()
    branch_1_op(branch_1)
    branch_2_op(branch_2)
```



Dynamic Graph

```
@op(out=DynamicOut())
def load_pieces():
    large_data = load_big_data()
    for idx, piece in large_data.chunk():
        yield DynamicOutput(piece, mapping_key=idx)
```

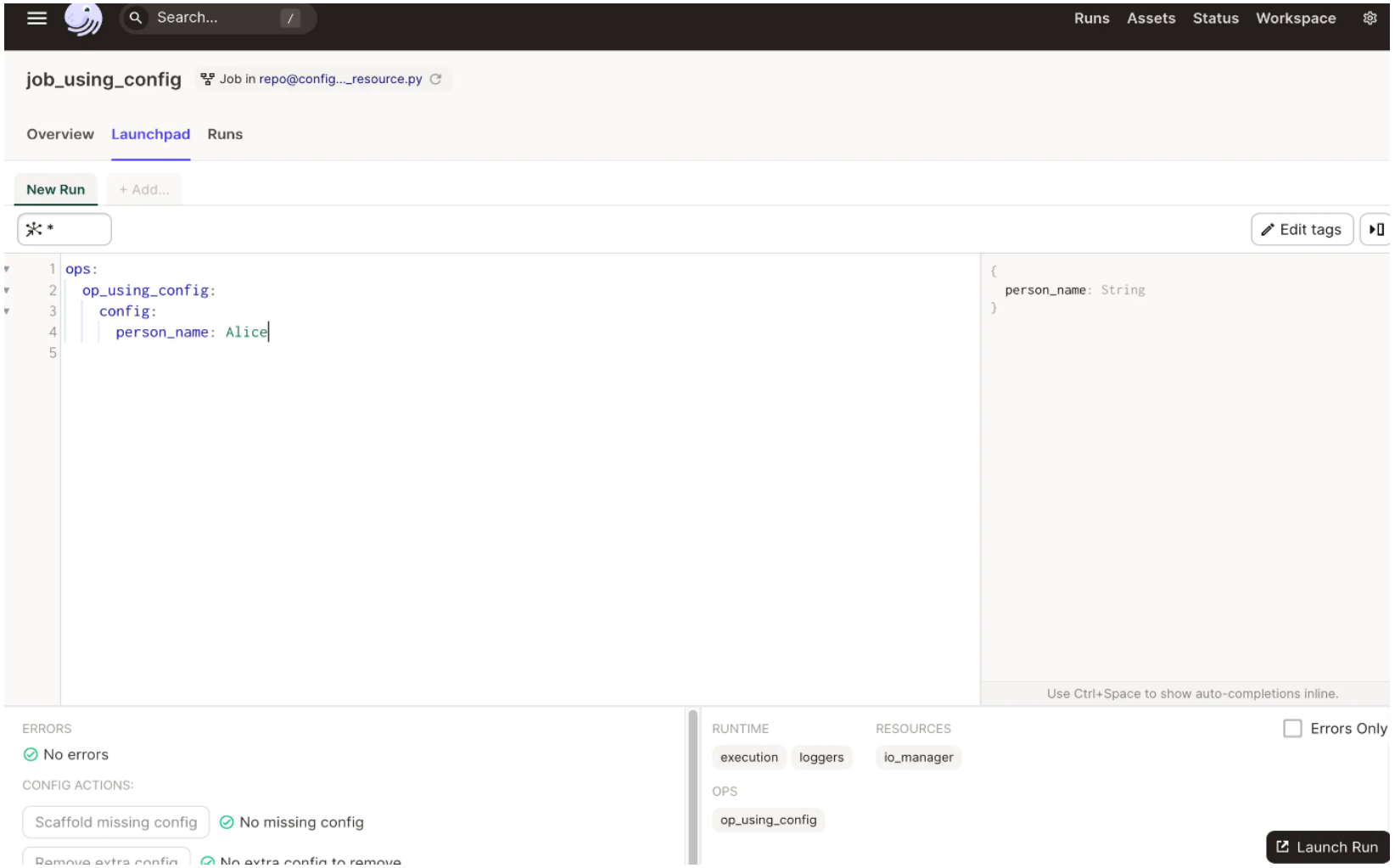
```
@job
def dynamic_graph():
    pieces = load_pieces()
    results = pieces.map(compute_piece)
    merge_and_analyze(results.collect())
```

Job and Repository

```python
from dagster import job, op


@op(config_schema={"config_param": str})
def do_something(context):
    context.log.info("config_param: " + context.op_config["config_param"])


default_config = {"ops": {"do_something": {"config": {"config_param": "stuff"}}}}


@job(config=default_config)
def do_it_all_with_default_config():
    do_something()


if __name__ == "__main__":
    # Will log "config_param: stuff"
    do_it_all_with_default_config.execute_in_process()
```

```python
@repository
def my_repository():
    return [
        asset1,
        asset2,
        asset3,
        job1_schedule,
        job2_sensor,
        job3,
    ]
```

Конфиг запуска



Dagster: продвинутые термины

- Partitioned job – деление запусков джобы по определенному параметру, что позволяет настраивать на него удобно сенсоры и делать бэкфил
- Backfill – запуск ранов для необходимых партиций в истории
- Sensor – запуск джобы по изменения состояния чего-то внешнего
- IO Manager – штука, которая определяет хранение и загрузку результатов Op-ов и Asset-ов.
- Ресурс – объект, который может использоваться в нескольких Op, например, клиент s3, спарк сессия, и т.п.

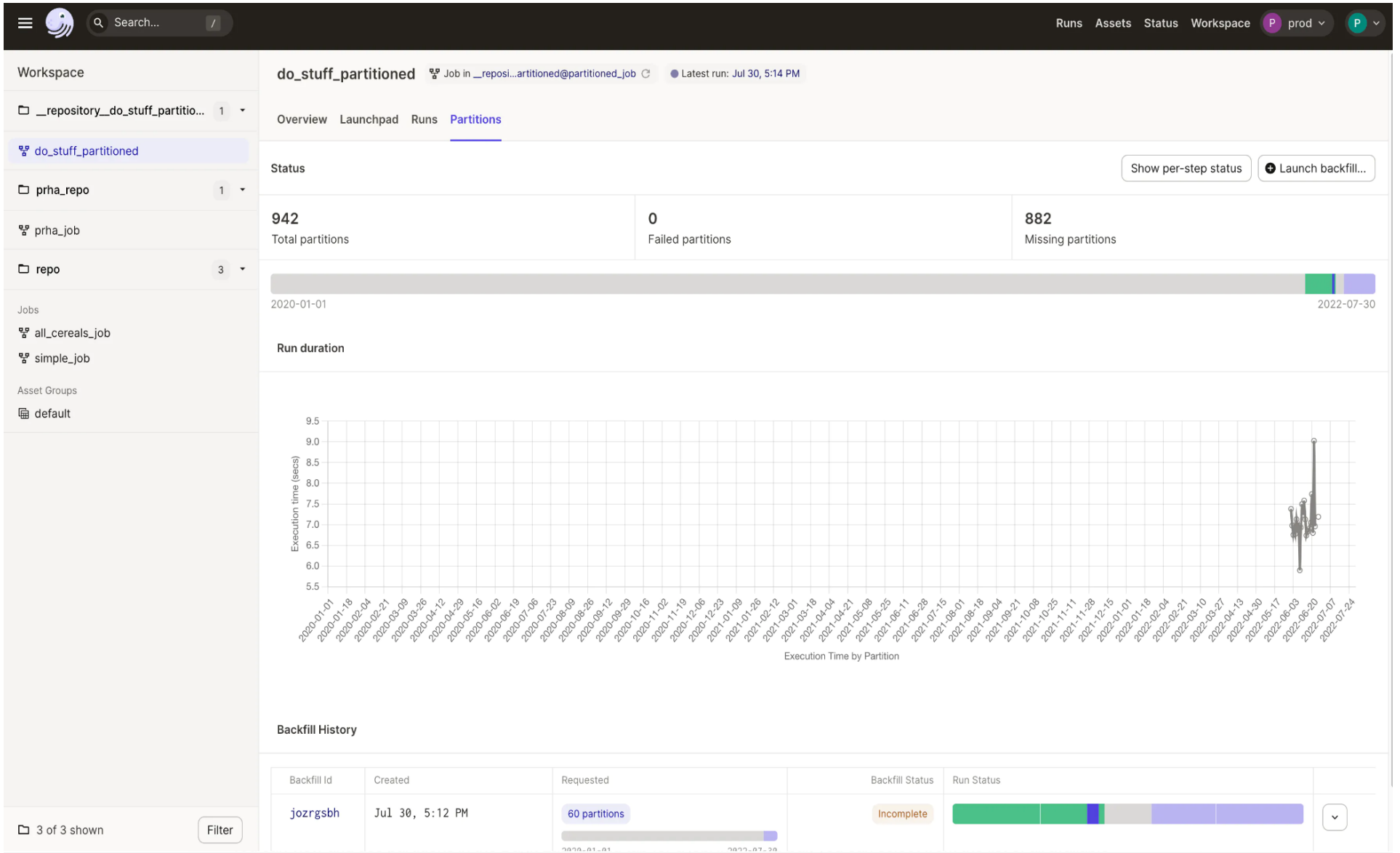
Partitioned job

```python
from dagster import daily_partitioned_config
from datetime import datetime


@daily_partitioned_config(start_date=datetime(2020, 1, 1))
def my_partitioned_config(start: datetime, _end: datetime):
    return {
        "ops": {
            "process_data_for_date": {"config": {"date": start.strftime("%Y-%m-%d")}}
        }
    }
```

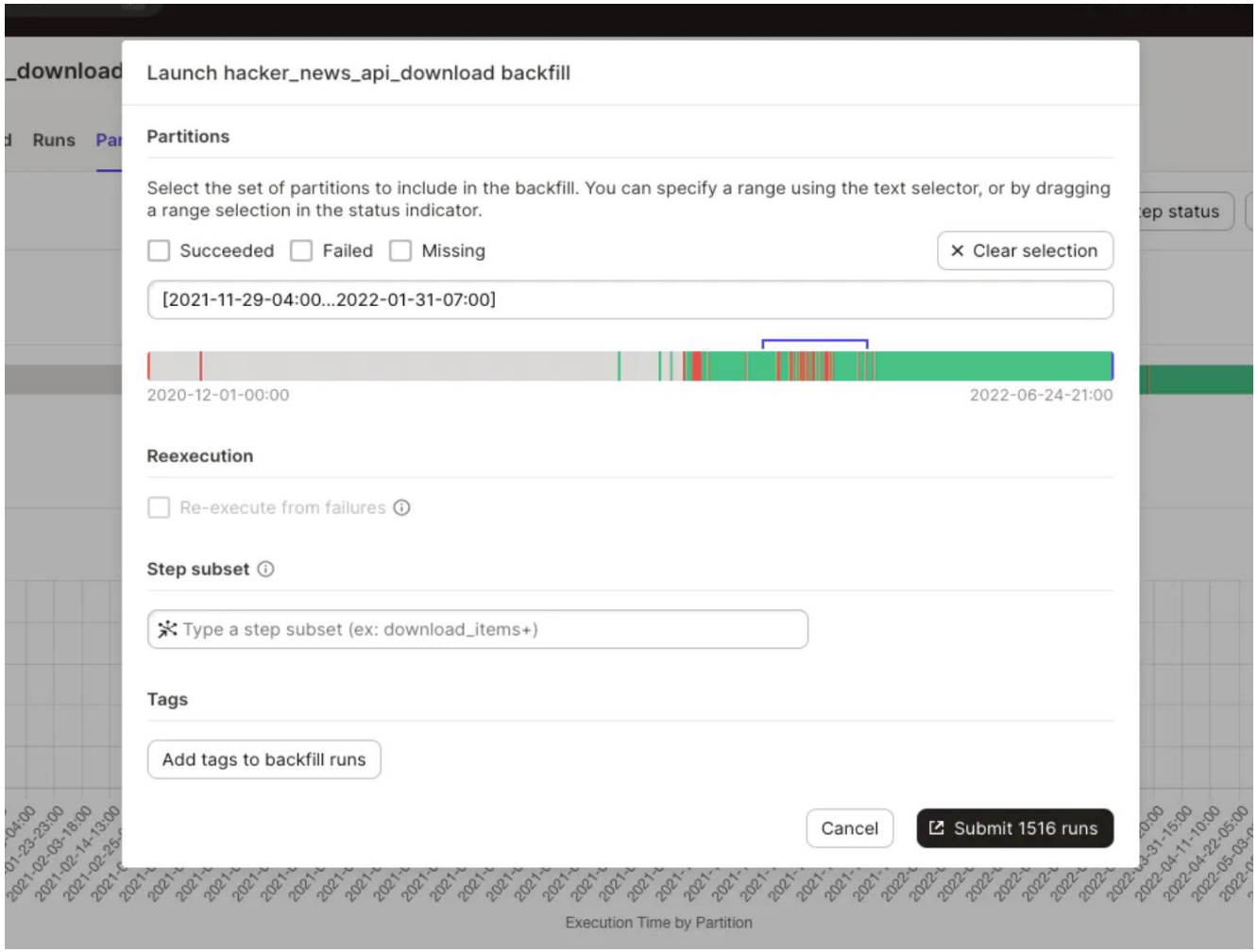
```python
@job(config=my_partitioned_config)
def do_stuff_partitioned():
    process_data_for_date()
```

```python
from dagster import job, op


@op(config_schema={"date": str})
def process_data_for_date(context):
    date = context.op_config["date"]
    context.log.info(f"processing data for {date}")


@job
def do_stuff():
    process_data_for_date()
```

Partitioned job



Backfill


Launch hacker_news_api_download backfill
Partitions
Select the set of partitions to include in the backfill. You can specify a range using the text selector, or by dragging a range selection in the status indicator.
Succeeded
Failed
Missing
Clear selection
[2021-11-29-04:00...2022-01-31-07:00]
2020-12-01-00:00
2022-06-24-21:00
Reexecution
Re-execute from failures
Step subset
Type a step subset (ex: download_items+)
Tags
Add tags to backfill runs
Cancel
Submit 1516 runs
Execution Time by Partition

Schedule

```
@job
def my_job():
    ...


basic_schedule = ScheduleDefinition(job=my_job, cron_schedule="0 0 * * *")
```

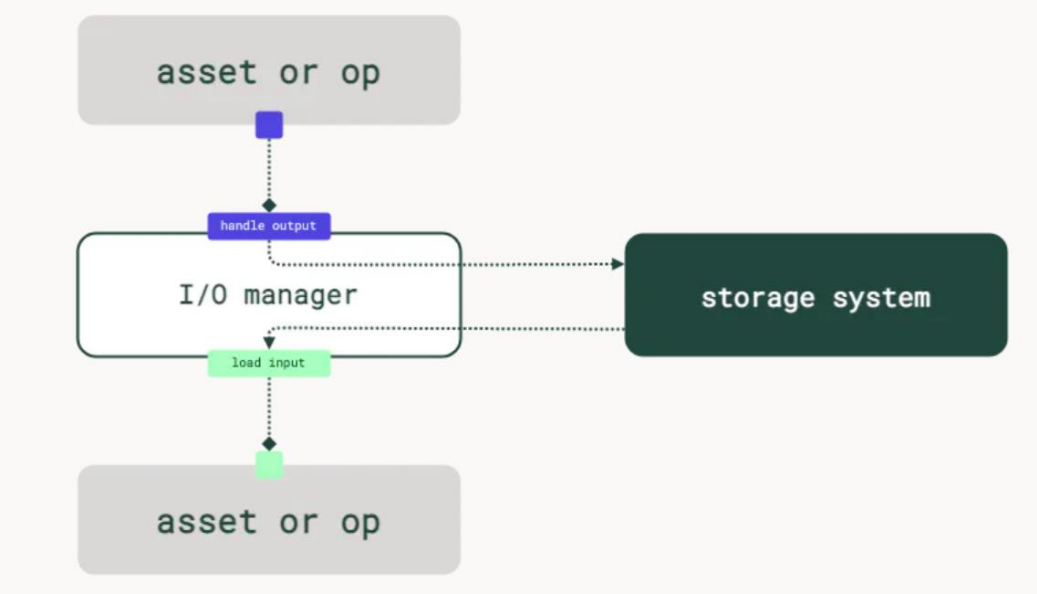
```
from dagster import AssetSelection, define_asset_job

asset_job = define_asset_job("asset_job", AssetSelection.groups("some_asset_group"))

basic_schedule = ScheduleDefinition(job=asset_job, cron_schedule="0 0 * * *")
```

Sensor

```python
import os
from dagster import sensor, RunRequest


@sensor(job=log_file_job)
def my_directory_sensor():
    for filename in os.listdir(MY_DIRECTORY):
        filepath = os.path.join(MY_DIRECTORY, filename)
        if os.path.isfile(filepath):
            yield RunRequest(
                run_key=filename,
                run_config={
                    "ops": {"process_file": {"config": {"filename": filename}}}
                },
            )
```

```python
from dagster import op, job


@op(config_schema={"filename": str})
def process_file(context):
    filename = context.op_config["filename"]
    context.log.info(filename)


@job
def log_file_job():
    process_file()
```

IO Manager




```python
from dagster import fs_io_manager, job, op


@op
def op_1():
    return 1


@op
def op_2(a):
    return a + 1


@job(resource_defs={"io_manager": fs_io_manager})
def my_job():
    op_2(op_1())
```

Resource

```python
from dagster import op

CREATE_TABLE_1_QUERY = "create table_1 as select * from table_0"


@op(required_resource_keys={"database"})
def op_requires_resources(context):
    context.resources.database.execute_query(CREATE_TABLE_1_QUERY)
```

```python
from dagster import resource


class ExternalCerealFetcher:
    def fetch_new_cereals(self, start_ts, end_ts):
        pass


@resource
def cereal_fetcher(init_context):
    return ExternalCerealFetcher()
```

Особенности разработки

- Данные полностью контролируете сами, а если передаете что-то между этапами – лучше это делать через однозначно временный способ, и точно знать, что это за способ.
- Выносите в конфигурацию запуска все, что планируете менять от запуска к запуску
- Деление на этапы повышает надежность, простоту дебага, контроль
- Но слишком большое деление – наоборот, понижает ☺
- Каждый этап должен иметь какое-то логическое значение
- Совет: смотрите уже существующие пайплайны, как происходит деление в них