INTRO TO METAFLOW: SIMPLIFY YOUR DATA SCIENCE WORKFLOWS

FAN YANG
PHD, LGSW

DR. AN'S AI GROUP

SEP, 2023

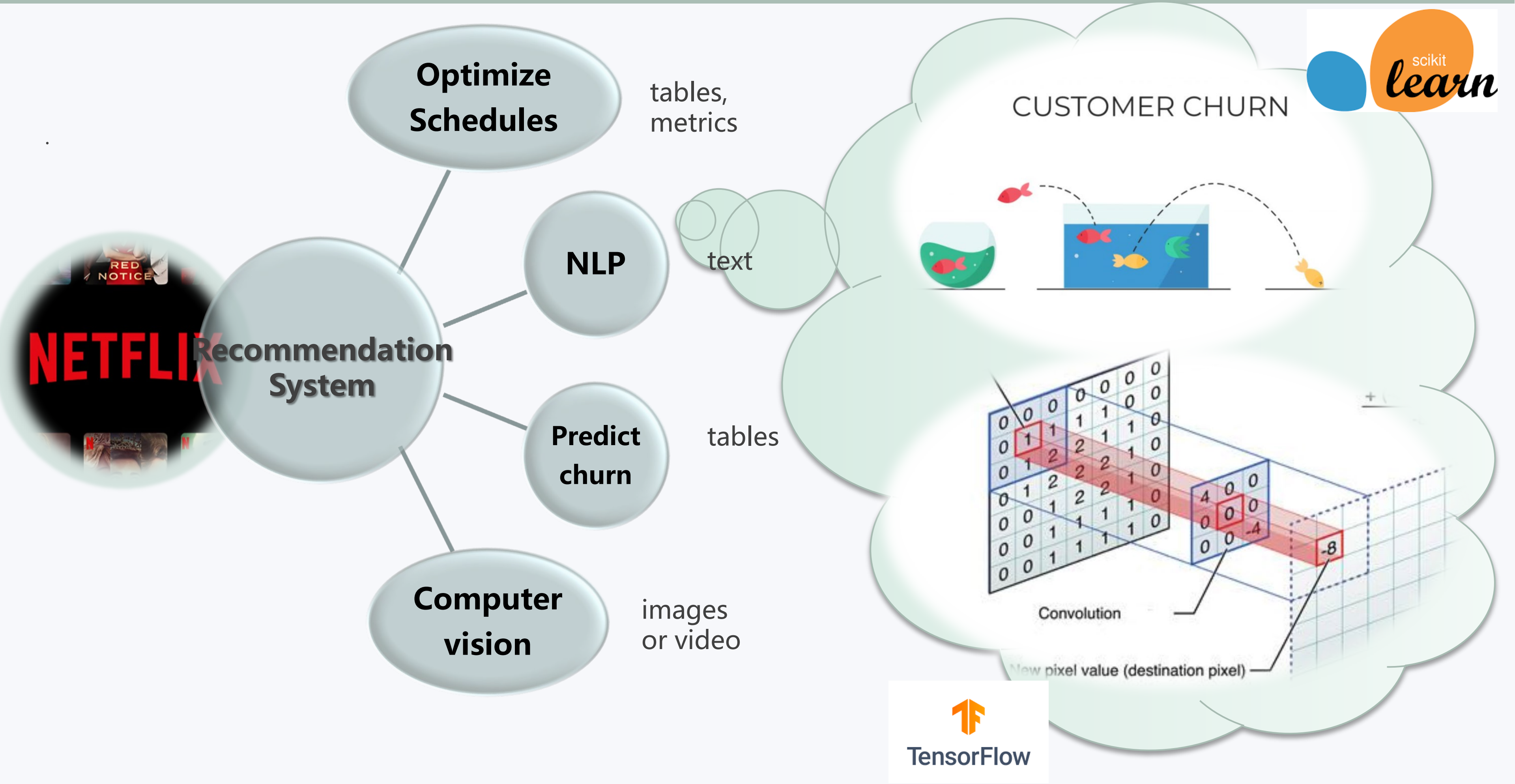
AGENDA

- WHAT IS METAFLOW
- WHY METAFLOW
- KEY FEATURES
- DEMO

01

WHAT IS METAFLOW

Story about Netflix



Recommendation System

Optimize Schedules

tables, metrics

NLP

text

Predict churn

tables

Computer vision

images or video

What is metaflow?

METAFLOW

Metaflow is a data science platform that can make data science code usable, scalable, reproducible, and production-ready.

NETFLIX

① Provide a highly usable API for structuring the code as a workflow, i.e. as a directed graph of steps (**usability**).

② Persist an immutable snapshot of data, code, and external dependencies required to execute each step (**reproducibility**).

③ Facilitate execution of the steps in various environments, from development to production (**scalability**, **production-readiness**).

④ Record metadata about previous executions and make them easily accessible (**usability**, **reproducibility**).

METAFLOW

Model Development

Feature Engineering

Model Operations

Architecture

Versioning

Job Scheduler

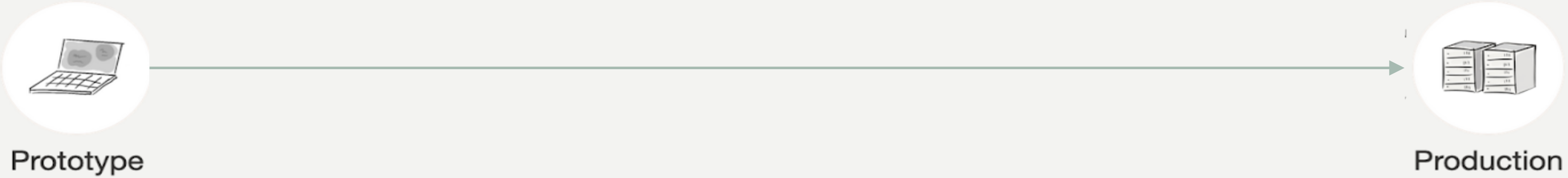
Compute Resources

Data Warehouse

02

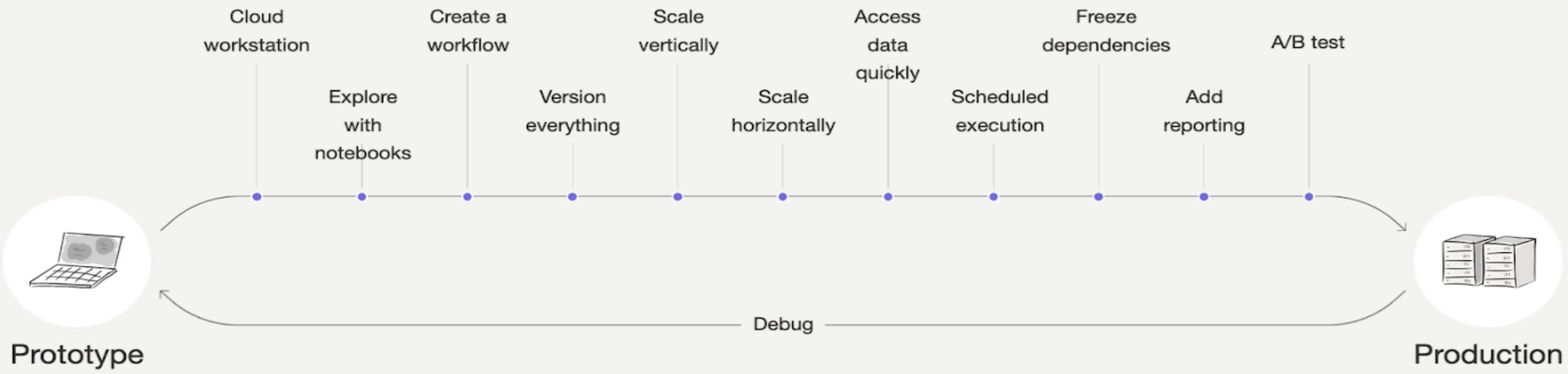
WHY METAFLOW

Project lifecycle: Baby-steps towards production



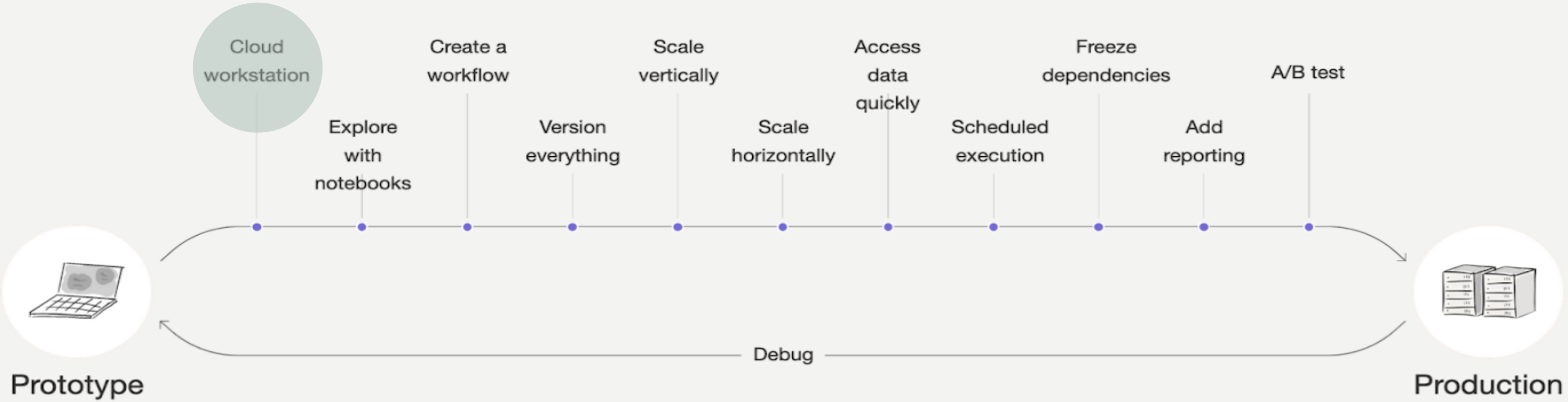
Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

Project lifecycle: Baby-steps towards production



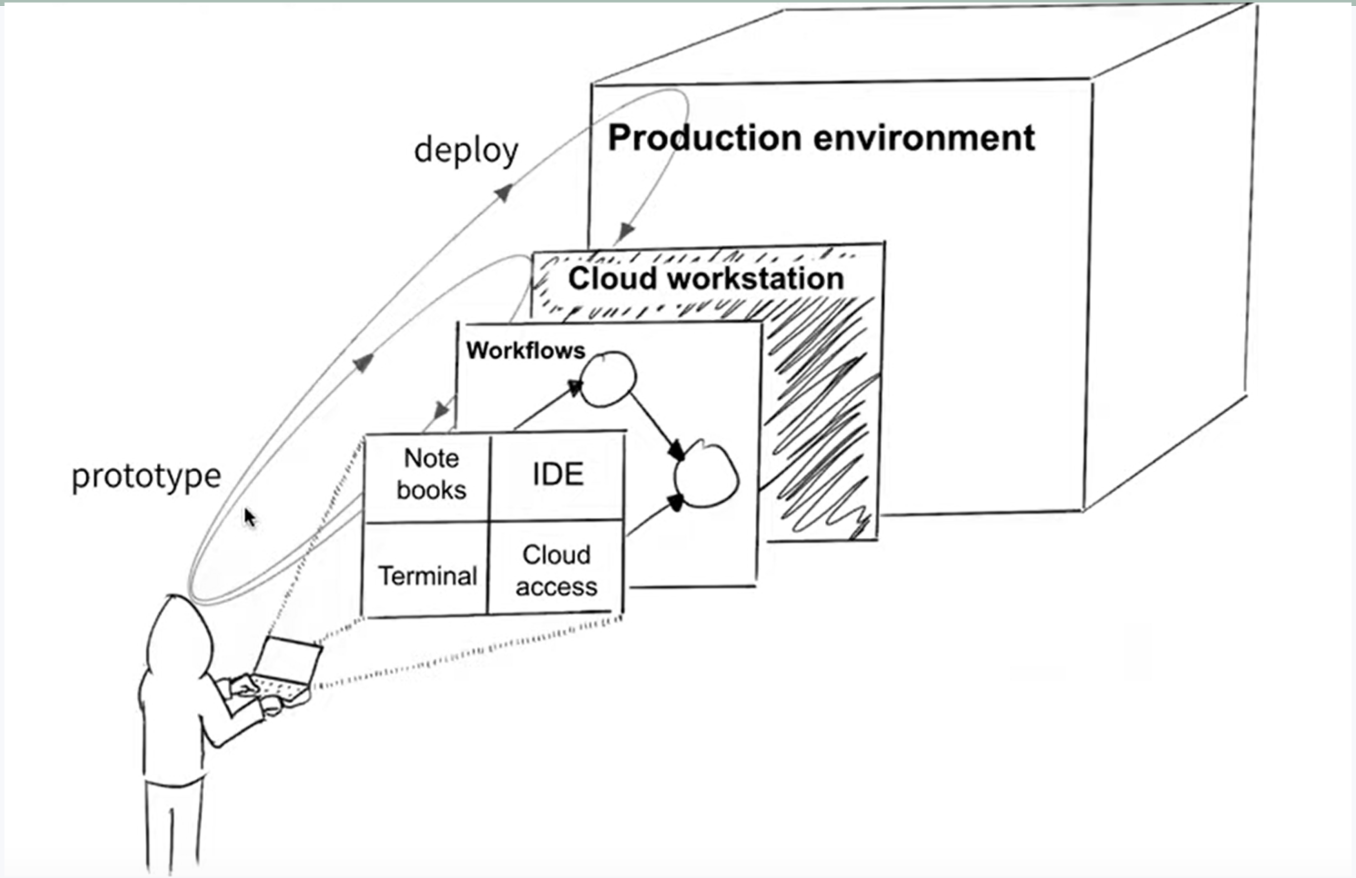
Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

Project lifecycle: Baby-steps towards production



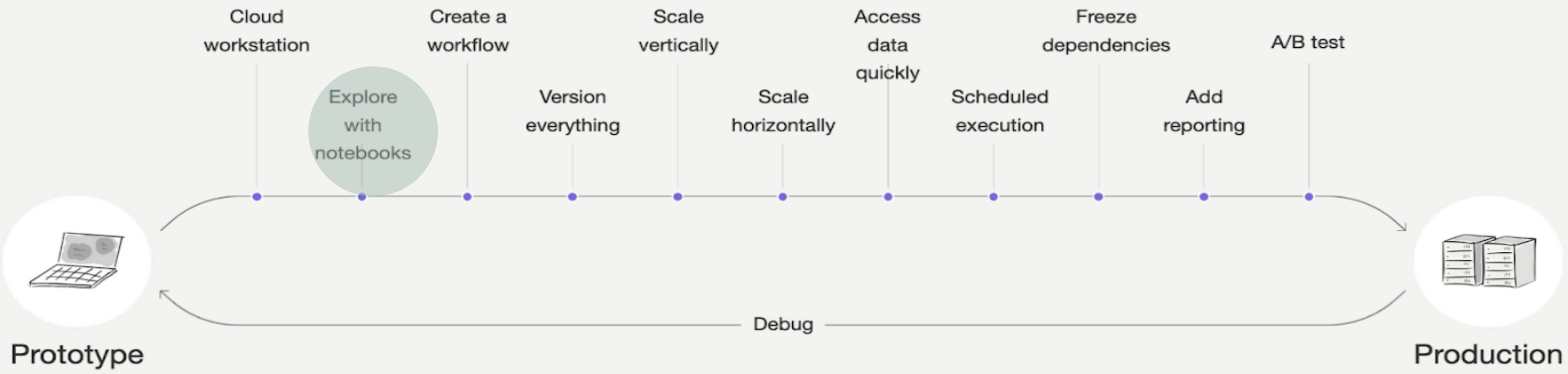
Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

Project lifecycle: Baby-steps towards production



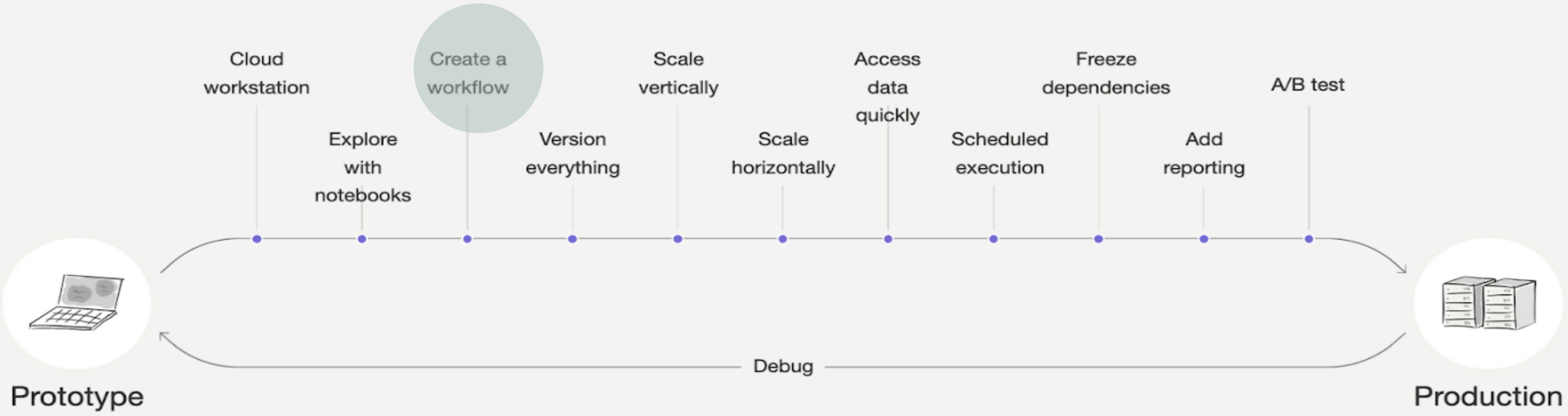
Derived from: outerbound.com

Project lifecycle: Baby-steps towards production



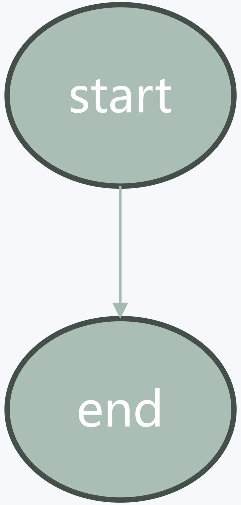
Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

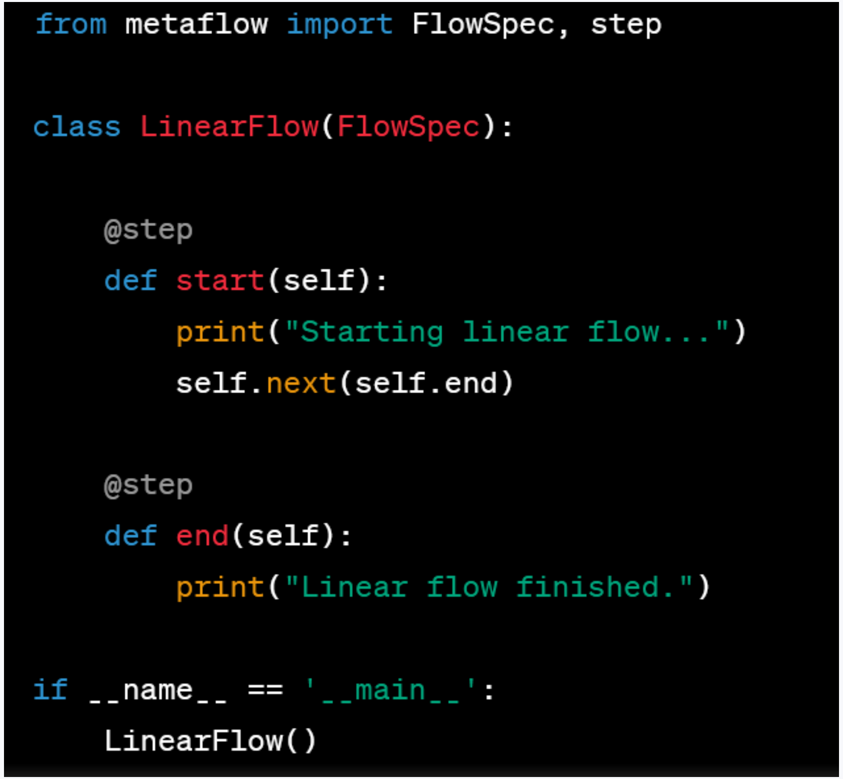
Project lifecycle: Baby-steps towards production



Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

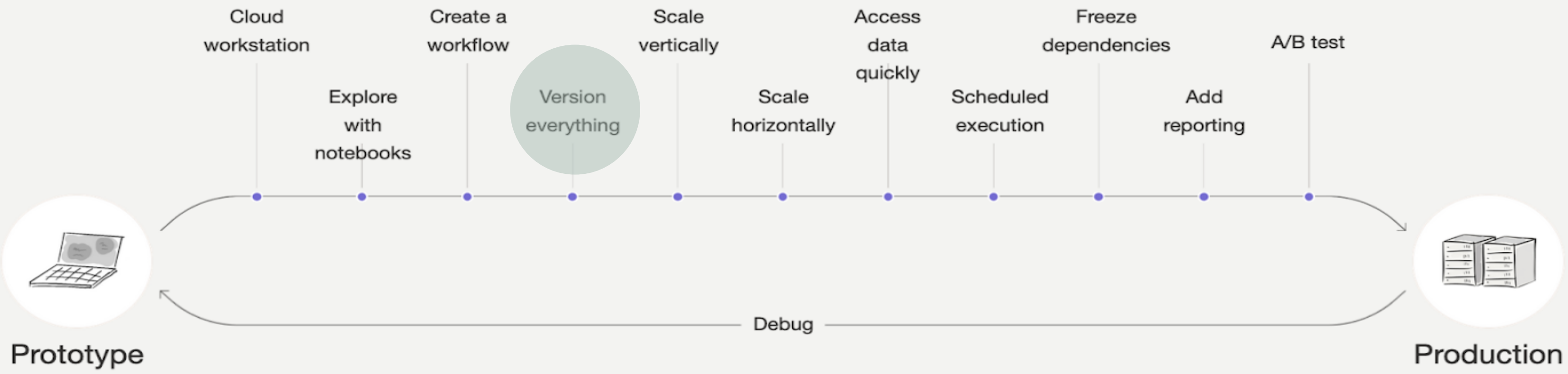
Project lifecycle: Baby-steps towards production





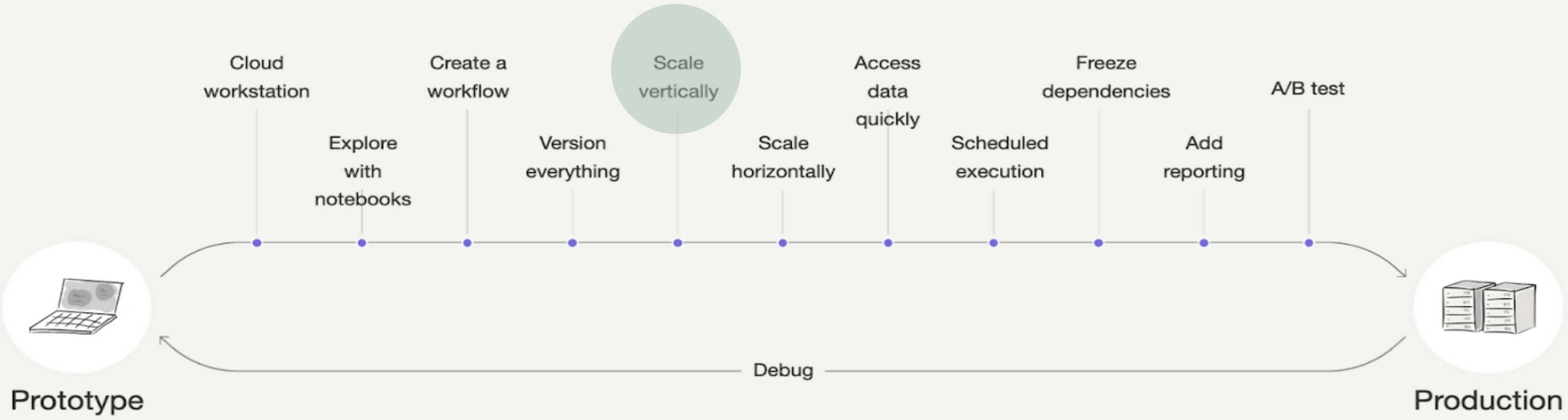
```python
from metaflow import FlowSpec, step

class LinearFlow(FlowSpec):

    @step
    def start(self):
        print("Starting linear flow...")
        self.next(self.end)

    @step
    def end(self):
        print("Linear flow finished.")

if __name__ == '__main__':
    LinearFlow()
```

Project lifecycle: Baby-steps towards production



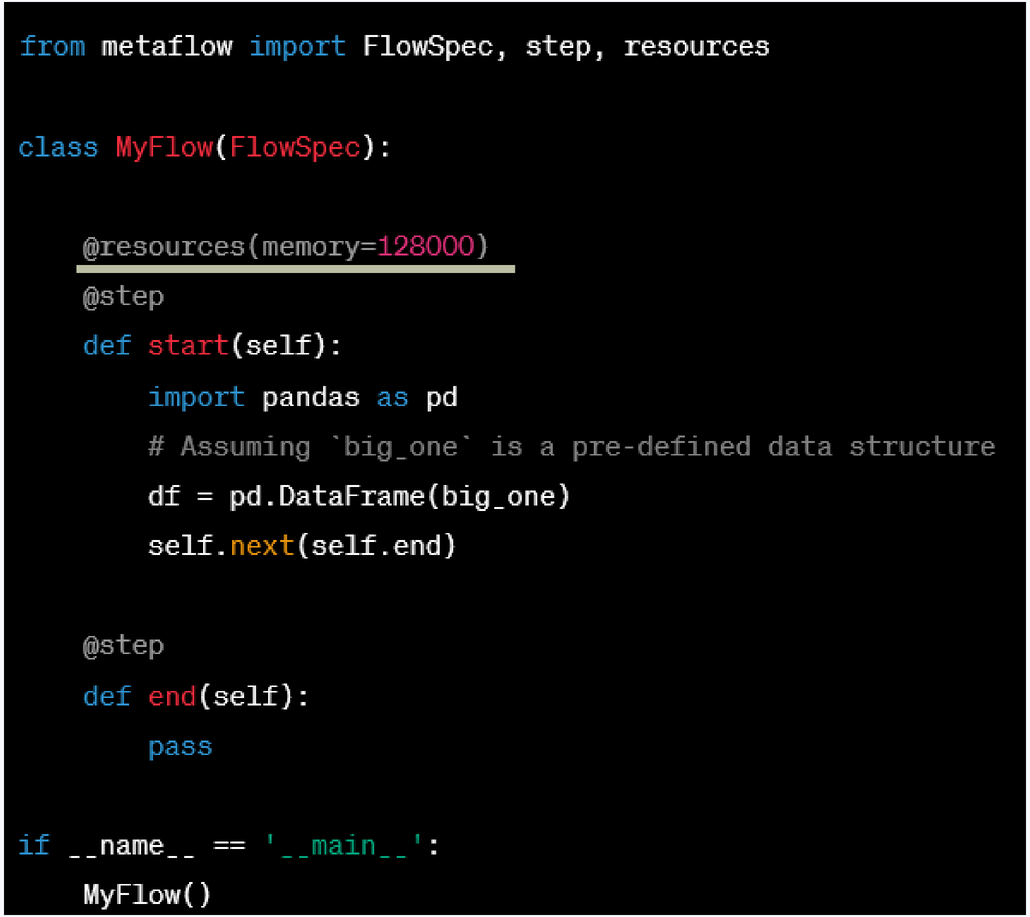
Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

Project lifecycle: Baby-steps towards production



Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

Project lifecycle: Baby-steps towards production

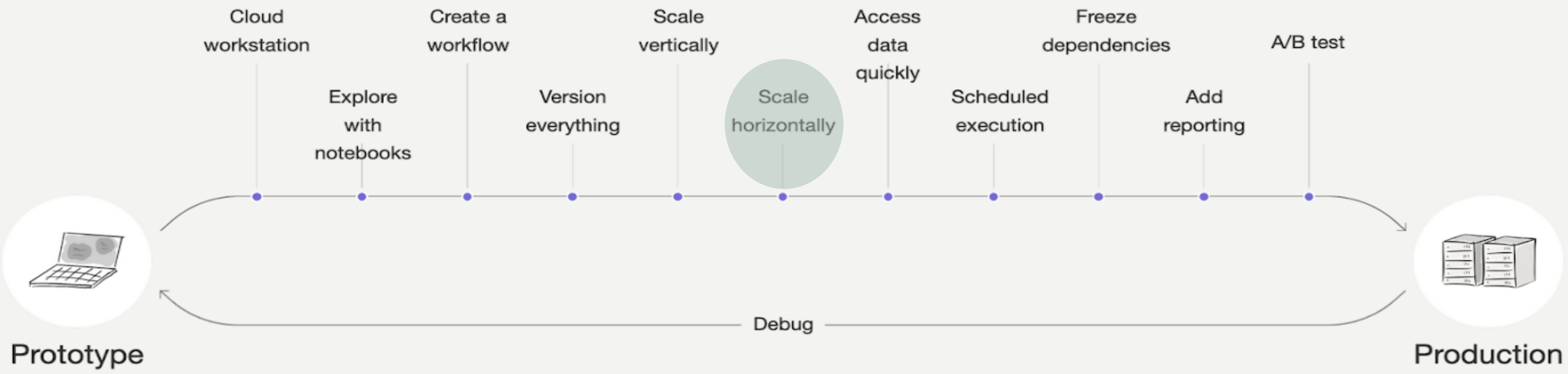


```python
from metaflow import FlowSpec, step, resources

class MyFlow(FlowSpec):

    @resources(memory=128000)
    @step
    def start(self):
        import pandas as pd
        # Assuming `big_one` is a pre-defined data structure
        df = pd.DataFrame(big_one)
        self.next(self.end)

    @step
    def end(self):
        pass

if __name__ == '__main__':
    MyFlow()
```

Terminal command:

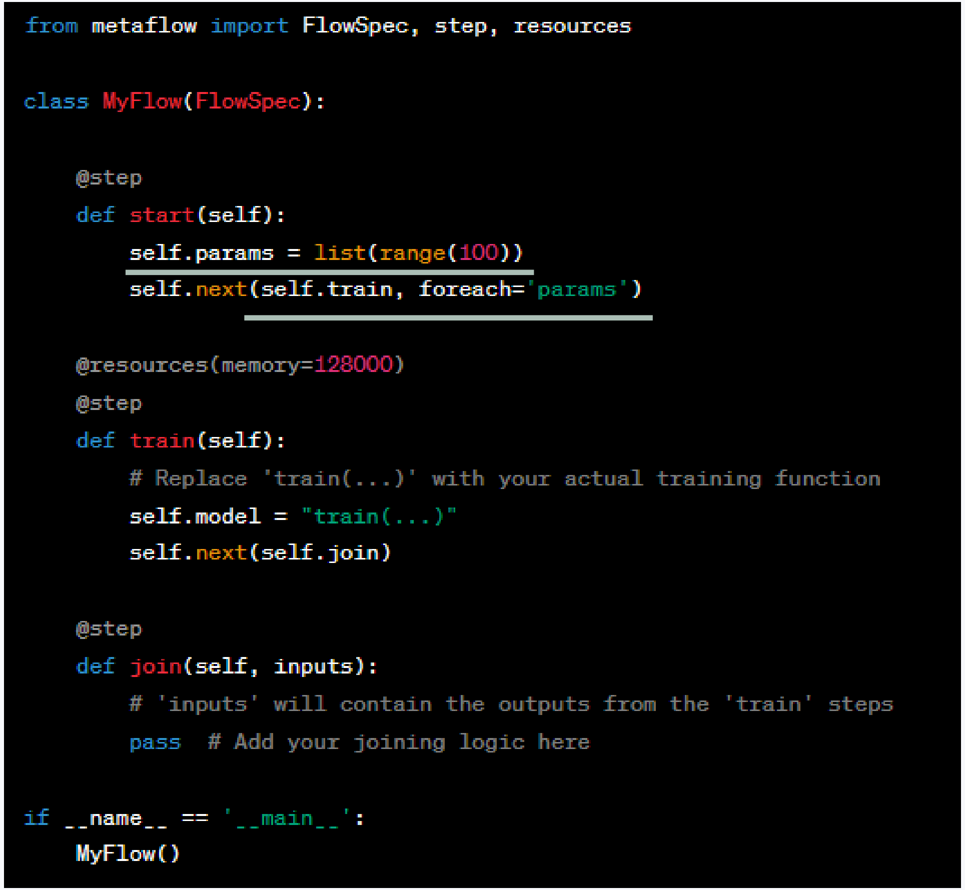
Python myflow.py run – with batch

Project lifecycle: Baby-steps towards production



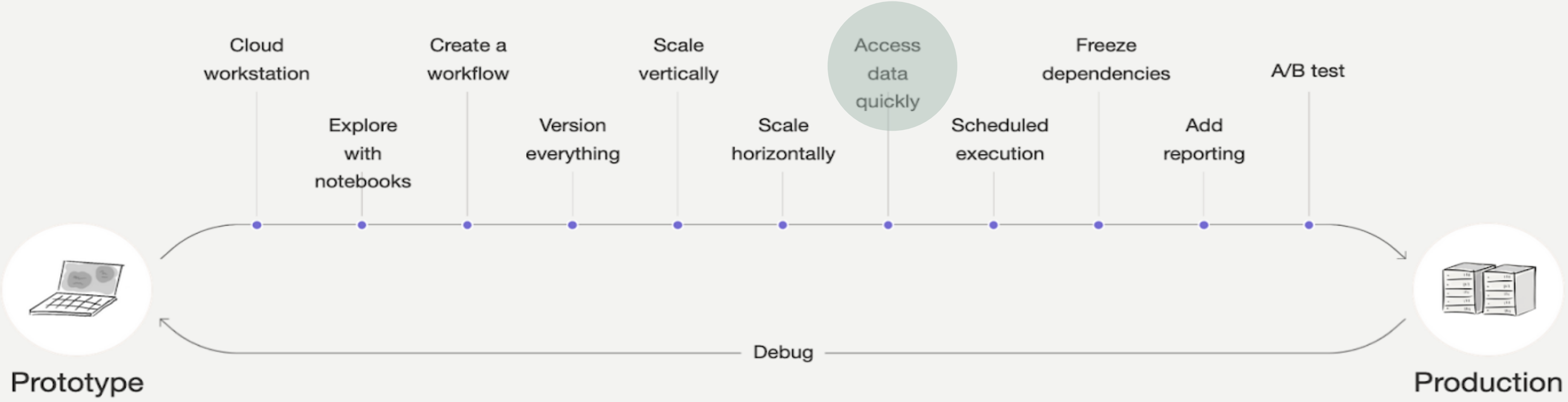
Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

Project lifecycle: Baby-steps towards production



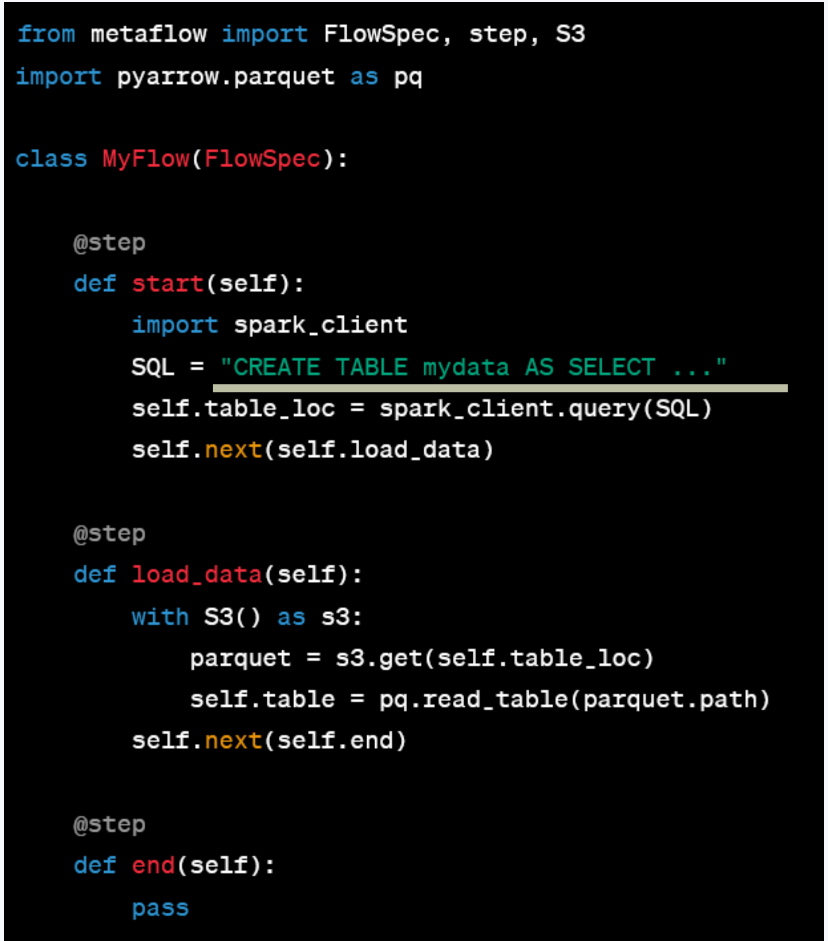
```python
from metaflow import FlowSpec, step, resources

class MyFlow(FlowSpec):

    @step
    def start(self):
        self.params = list(range(100))
        self.next(self.train, foreach='params')

    @resources(memory=128000)
    @step
    def train(self):
        # Replace 'train(...)' with your actual training function
        self.model = "train(...)"
        self.next(self.join)

    @step
    def join(self, inputs):
        # 'inputs' will contain the outputs from the 'train' steps
        pass  # Add your joining logic here

if __name__ == '__main__':
    MyFlow()
```

Project lifecycle: Baby-steps towards production

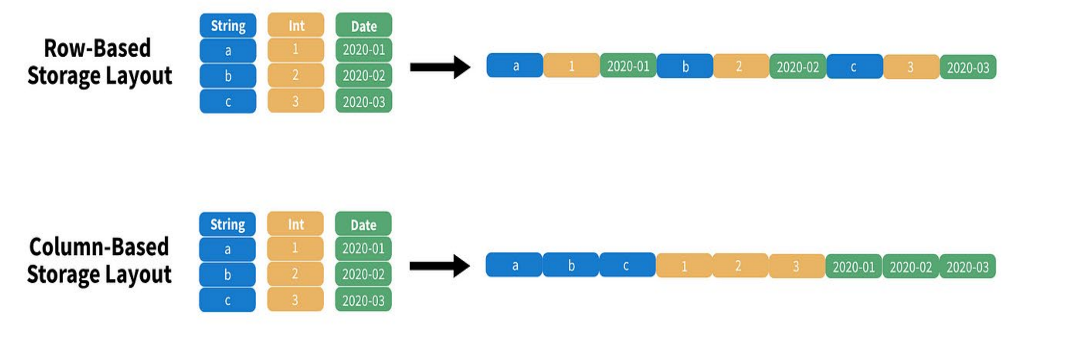


Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

Project lifecycle: Baby-steps towards production



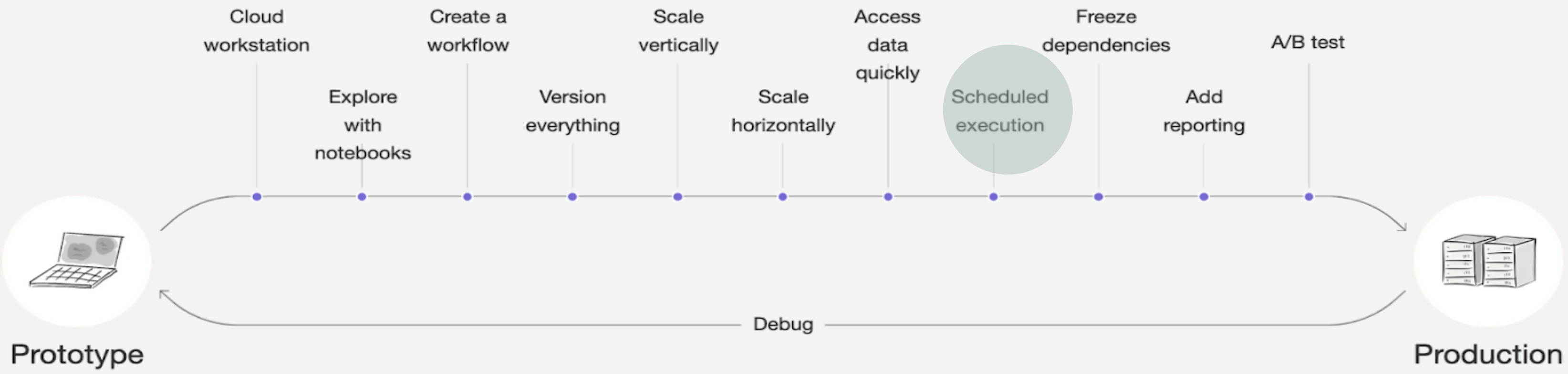
```
from metaflow import FlowSpec, step, S3
import pyarrow.parquet as pq

class MyFlow(FlowSpec):

    @step
    def start(self):
        import spark_client
        SQL = "CREATE TABLE mydata AS SELECT ..."
        self.table_loc = spark_client.query(SQL)
        self.next(self.load_data)

    @step
    def load_data(self):
        with S3() as s3:
            parquet = s3.get(self.table_loc)
            self.table = pq.read_table(parquet.path)
        self.next(self.end)

    @step
    def end(self):
        pass
```



Parquet

SQL

Project lifecycle: Baby-steps towards production



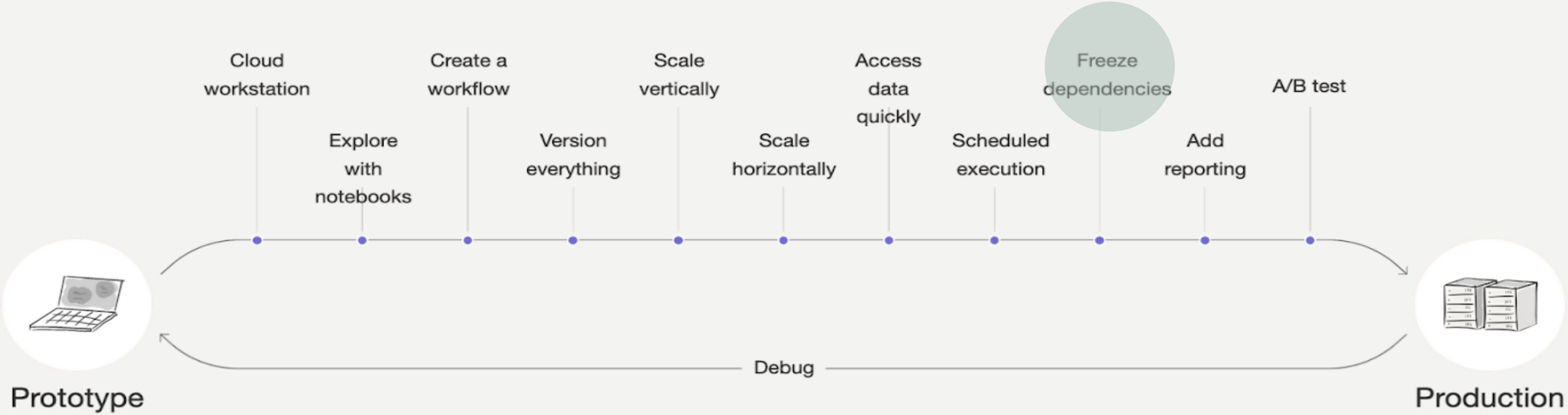
Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

Project lifecycle: Baby-steps towards production

Terminal command:

```
Python myflow.py  step – functions create
```

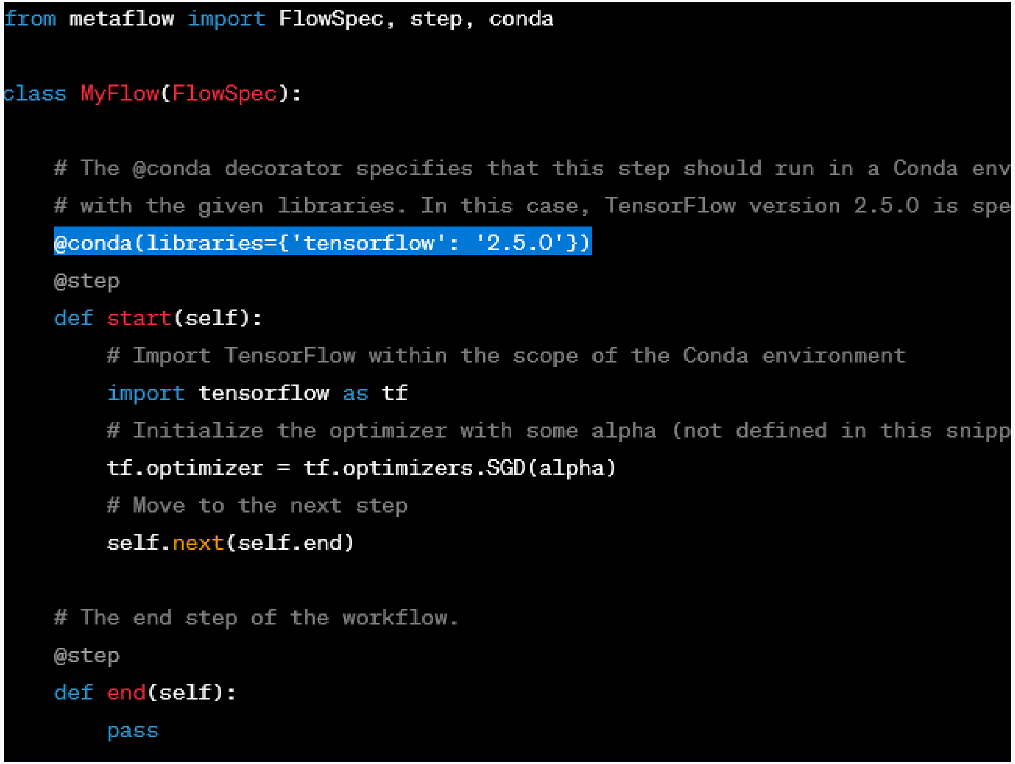
Project lifecycle: Baby-steps towards production



Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

Project lifecycle: Baby-steps towards production

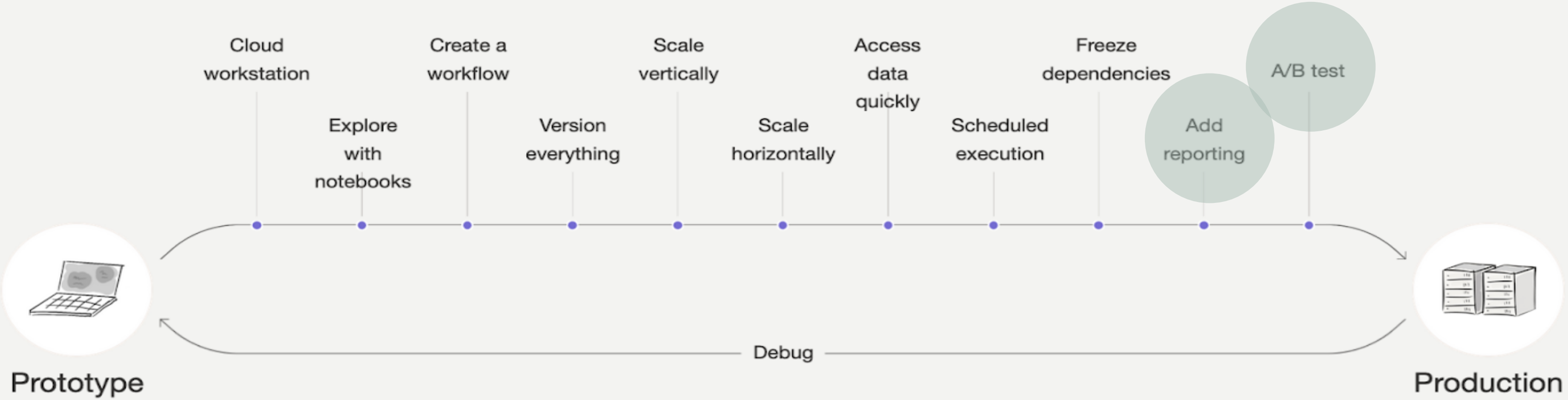
VS



```
from metaflow import FlowSpec, step, conda

class MyFlow(FlowSpec):

    # The @conda decorator specifies that this step should run in a Conda env
    # with the given libraries. In this case, TensorFlow version 2.5.0 is spe
    @conda(libraries={'tensorflow': '2.5.0'})
    @step
    def start(self):
        # Import TensorFlow within the scope of the Conda environment
        import tensorflow as tf
        # Initialize the optimizer with some alpha (not defined in this snipp
        tf.optimizer = tf.optimizers.SGD(alpha)
        # Move to the next step
        self.next(self.end)

    # The end step of the workflow.
    @step
    def end(self):
        pass
```

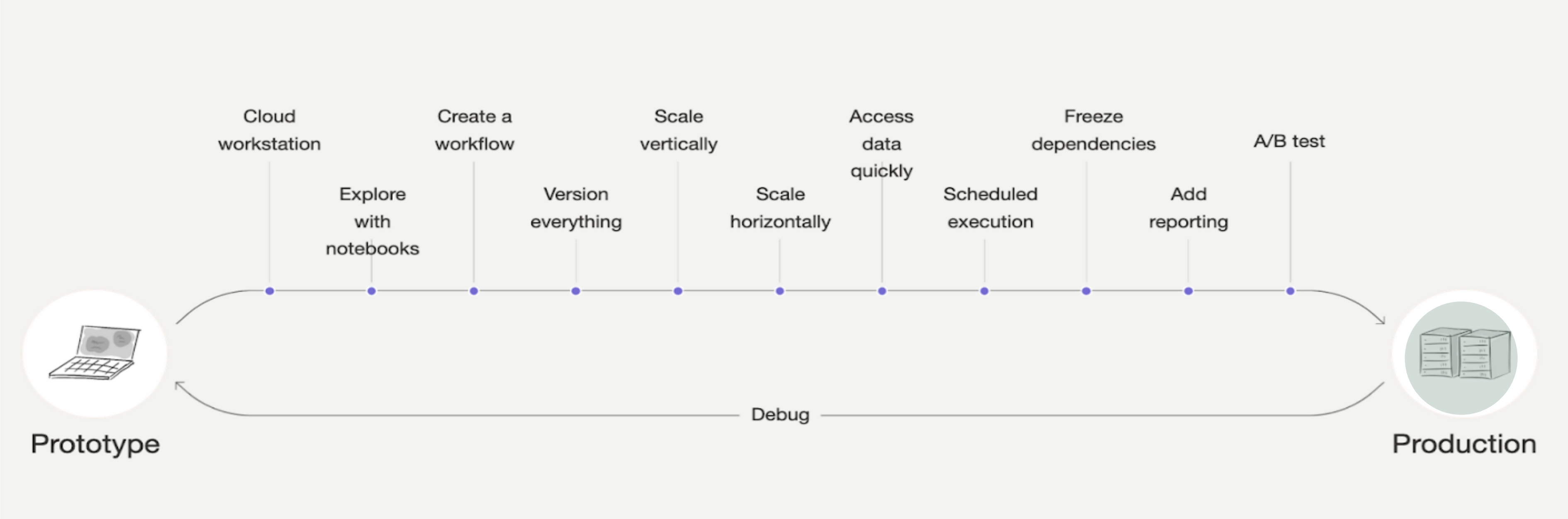
Define a stable execution environment

Project lifecycle: Baby-steps towards production



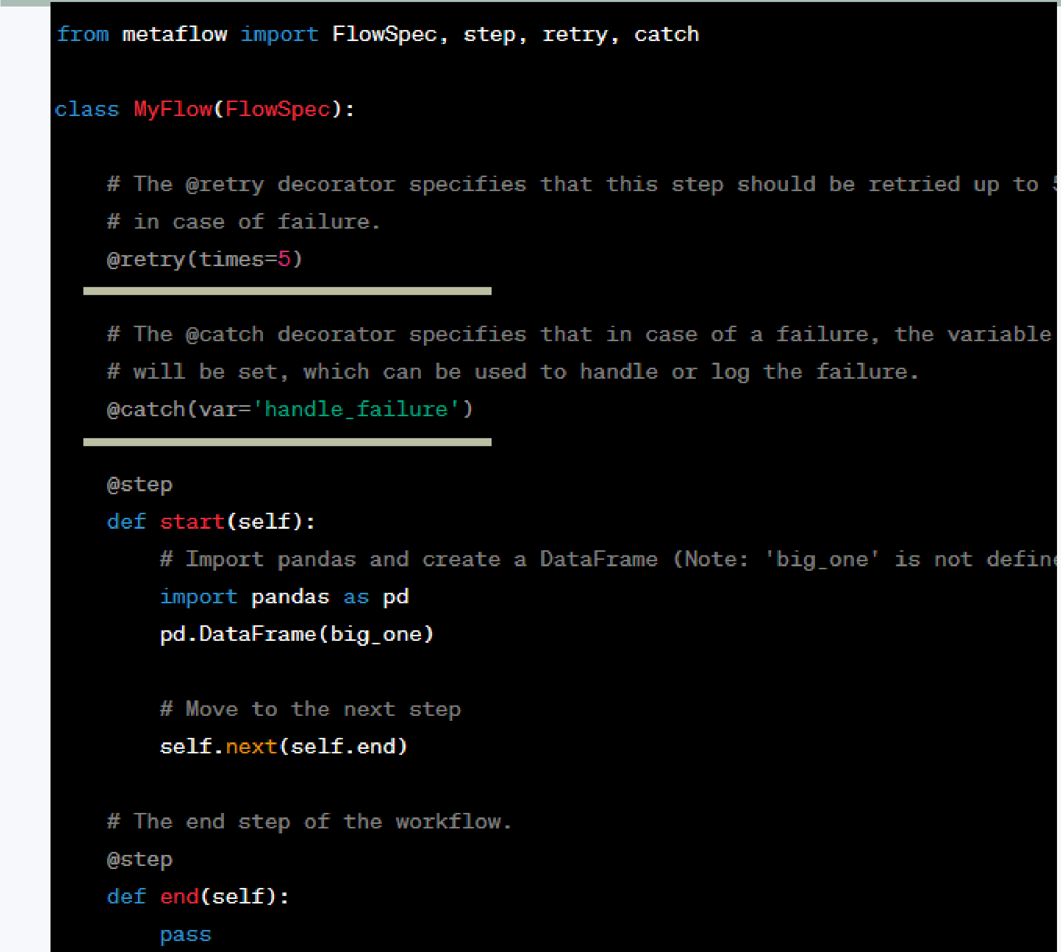
Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

Project lifecycle: Baby-steps towards production



Picture derived from:
https://outerbounds.com/blog/learn-full-stack-ml-corise/

Project lifecycle: Baby-steps towards production



```python
from metaflow import FlowSpec, step, retry, catch

class MyFlow(FlowSpec):

    # The @retry decorator specifies that this step should be retried up to
    # in case of failure.
    @retry(times=5)

    # The @catch decorator specifies that in case of a failure, the variable
    # will be set, which can be used to handle or log the failure.
    @catch(var='handle_failure')

    @step
    def start(self):
        # Import pandas and create a DataFrame (Note: 'big_one' is not define
        import pandas as pd
        pd.DataFrame(big_one)

        # Move to the next step
        self.next(self.end)

    # The end step of the workflow.
    @step
    def end(self):
        pass
```

03

KEY FEATURES

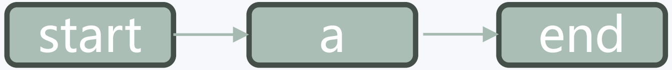
3.1 Key features

1. **Directed Graph of Operations**:
   - Represents a program's flow, making it intuitive for data processing pipelines.
   - Especially suitable for machine learning workflows.
2. **Flow**:
   - The graph of operations.
   - Comprises steps (nodes) and transitions (edges).
3. **Steps**:
   - Operations in the flow.
   - Every flow must have a "start" and an "end" step.
4. **Run**:
   - Execution of the flow.
   - Begins at "start" and concludes successfully at "end".

Summary: Metaflow offers a structured yet flexible approach to designing data-driven workflows, ensuring clarity from start to finish.

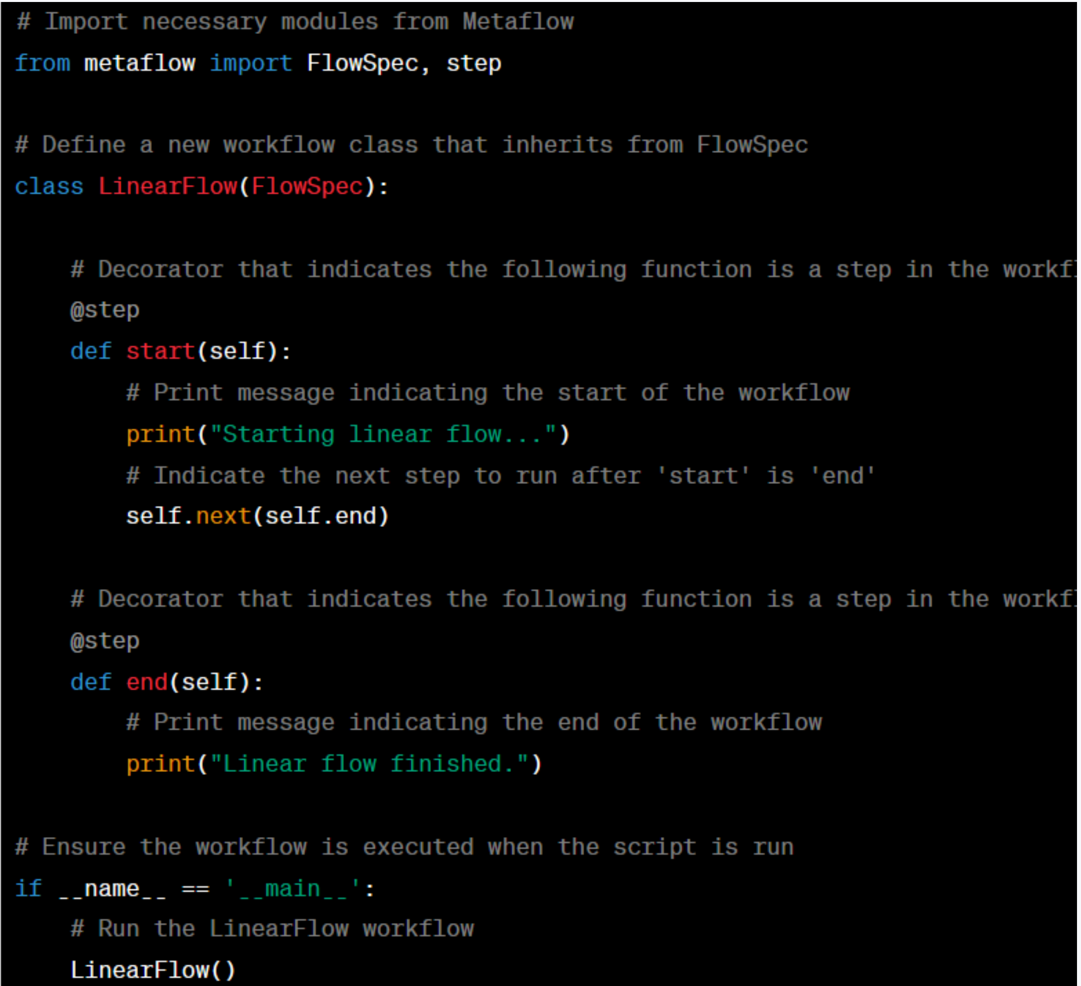
3.1 Key features

1.Linear Transition:
Represents a direct flow
from one operation to the ne



A graph with two linear transitions

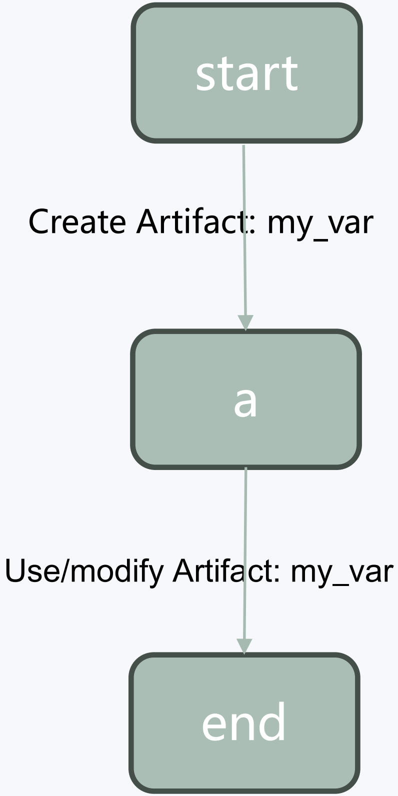
The journey from
"start" to "end" is customizable.



```
# Import necessary modules from Metaflow
from metaflow import FlowSpec, step

# Define a new workflow class that inherits from FlowSpec
class LinearFlow(FlowSpec):

    # Decorator that indicates the following function is a step in the workf
    @step
    def start(self):
        # Print message indicating the start of the workflow
        print("Starting linear flow...")
        # Indicate the next step to run after 'start' is 'end'
        self.next(self.end)

    # Decorator that indicates the following function is a step in the workf
    @step
    def end(self):
        # Print message indicating the end of the workflow
        print("Linear flow finished.")

# Ensure the workflow is executed when the script is run
if __name__ == '__main__':
    # Run the LinearFlow workflow
    LinearFlow()
```

3.1 Key features

2. **Artifacts**

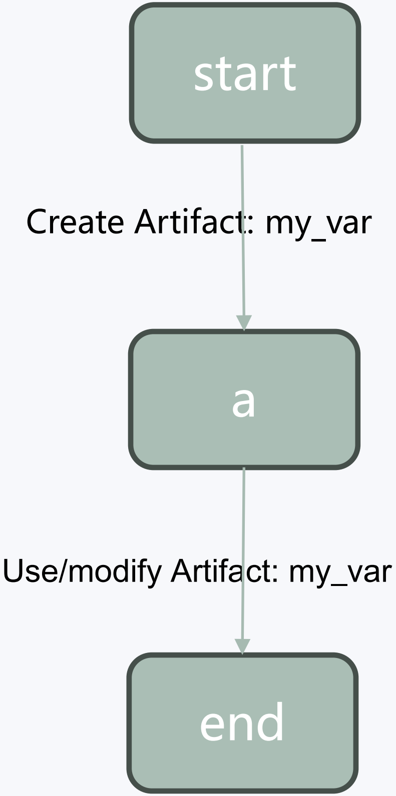


Key Benefits of Artifacts:

1. **Automated Data Management**:
   1. Manage data flow effortlessly.
   2. No manual data loading or storing.
2. **Persistence for Future Use**:
   1. Analyze later using the Client API.
   2. Visualize with Cards or use across different flows.
3. **Consistency Across Environments**:
   1. Seamlessly transition between local and cloud environments.
   2. No explicit data transfer required.
4. **Debugging & Recovery**:
   1. Access past artifacts to inspect data before failures.
   2. Resume past executions post bug fixes.

3.1 Key features

2. **Artifacts**



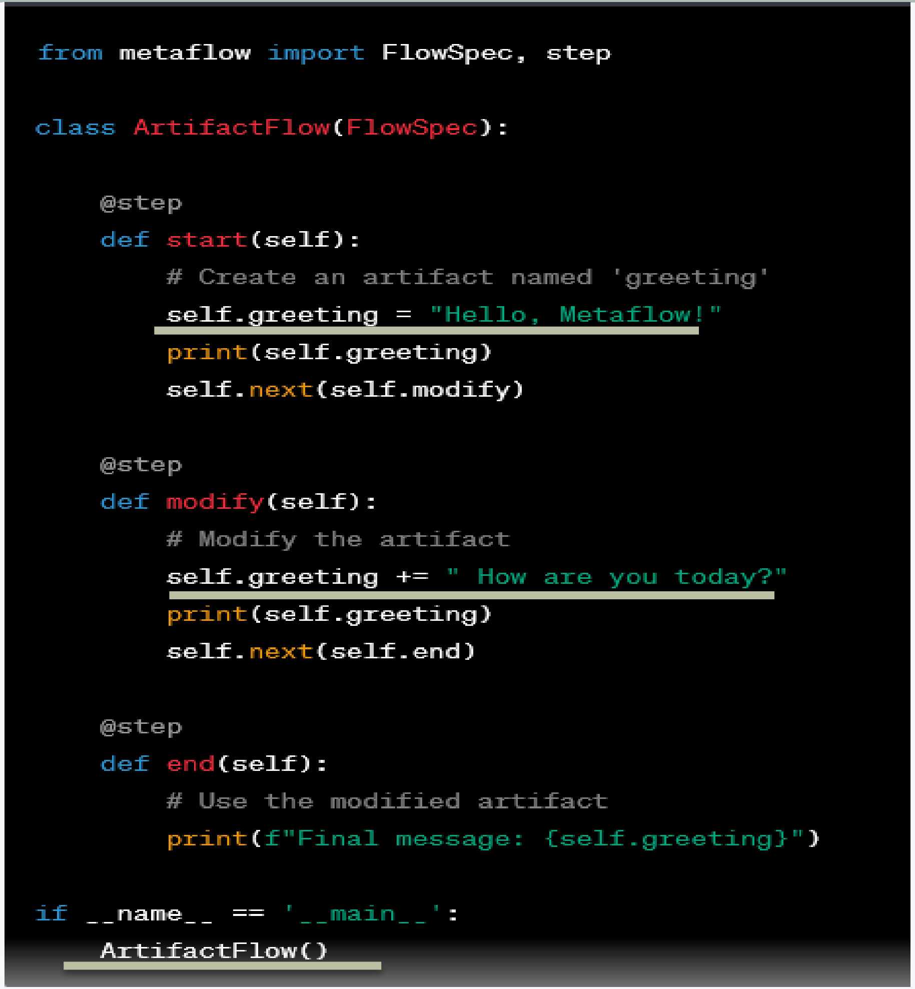
start

Create Artifact: my_var

a

Use/modify Artifact: my_var

end

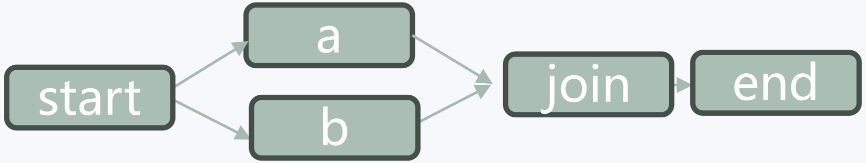


```python
from metaflow import FlowSpec, step

class ArtifactFlow(FlowSpec):

    @step
    def start(self):
        # Create an artifact named 'greeting'
        self.greeting = "Hello, Metaflow!"
        print(self.greeting)
        self.next(self.modify)

    @step
    def modify(self):
        # Modify the artifact
        self.greeting += " How are you today?"
        print(self.greeting)
        self.next(self.end)

    @step
    def end(self):
        # Use the modified artifact
        print(f"Final message: {self.greeting}")

if __name__ == '__main__':
    ArtifactFlow()
```

3.1 Key features

3. Branch Transition

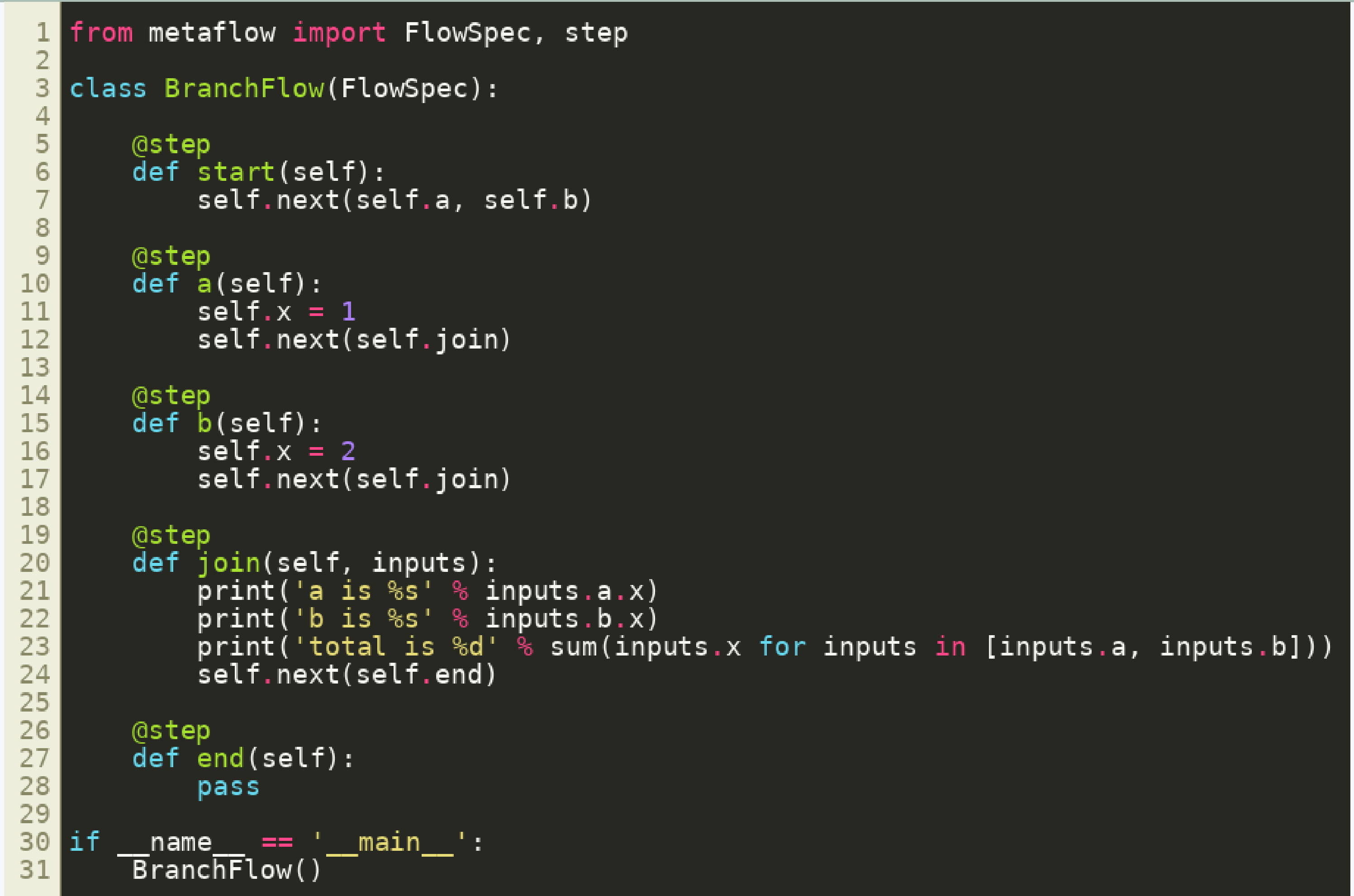
We can express parallel steps with **a branch**.
In the figure below, start transitions to two parallel steps, a and b.



A benefit of a branch is performance: Metaflow can execute a and b over multiple CPU cores or over multiple instances in the cloud.

3.1 Key features

3. Branch



```python
from metaflow import FlowSpec, step

class BranchFlow(FlowSpec):

    @step
    def start(self):
        self.next(self.a, self.b)

    @step
    def a(self):
        self.x = 1
        self.next(self.join)

    @step
    def b(self):
        self.x = 2
        self.next(self.join)

    @step
    def join(self, inputs):
        print('a is %s' % inputs.a.x)
        print('b is %s' % inputs.b.x)
        print('total is %d' % sum(inputs.x for inputs in [inputs.a, inputs.b]))
        self.next(self.end)

    @step
    def end(self):
        pass

if __name__ == '__main__':
    BranchFlow()
```

3.1 Key features

➢ Card

- **What is a Card?**
  - A UI component for visualizing data and results.
- **Key Features**
  - Interactive Visualizations
  - Shareable Insights
  - Traceability
- **Use Cases**
  - Debugging
  - Data Exploration
  - Reporting

04

DEMOS

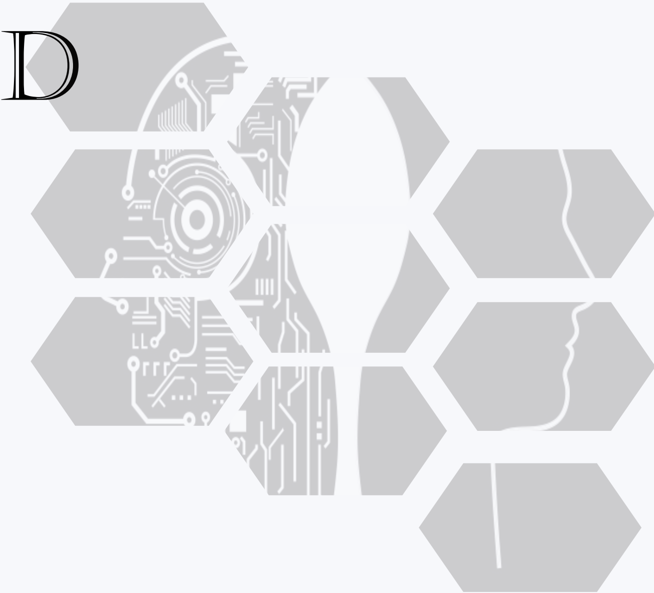
4. Demo

➢ Sandbox

- **Purpose**: Exclusive for testing Metaflow in data science.
  Not for production or general computation.
- **Data Caution**: Test with datasets,
  avoid confidential, personal, or sensitive data.
- **Duration**: Default access is 7 days.
  Post-expiry, data is deleted.
  Extend by request.
- **Connectivity**: No internet in the Sandbox.
  Common R libraries pre-installed.
- **Capabilities**: Use up to 8 instances with 8 cores & 30GB RAM using the batch decorator.

05

RESOURCES AND REFERENCES



7.1 References and Resources

- **https://app.slack.com/**

- **https://docs.metaflow.org/**

- **https://github.com/Netflix/metaflow/tree/master/metaflow/tutorials/00-helloworld**
- **Tutorial code on Github.**

- **https://outerbounds.com/**
- **Sandbox**

THANKS FOR LISTENING!