Serverless Parallel Data Analytics

DS 5110/CS 5501: Big Data Systems

Spring 2024

Lecture 8c

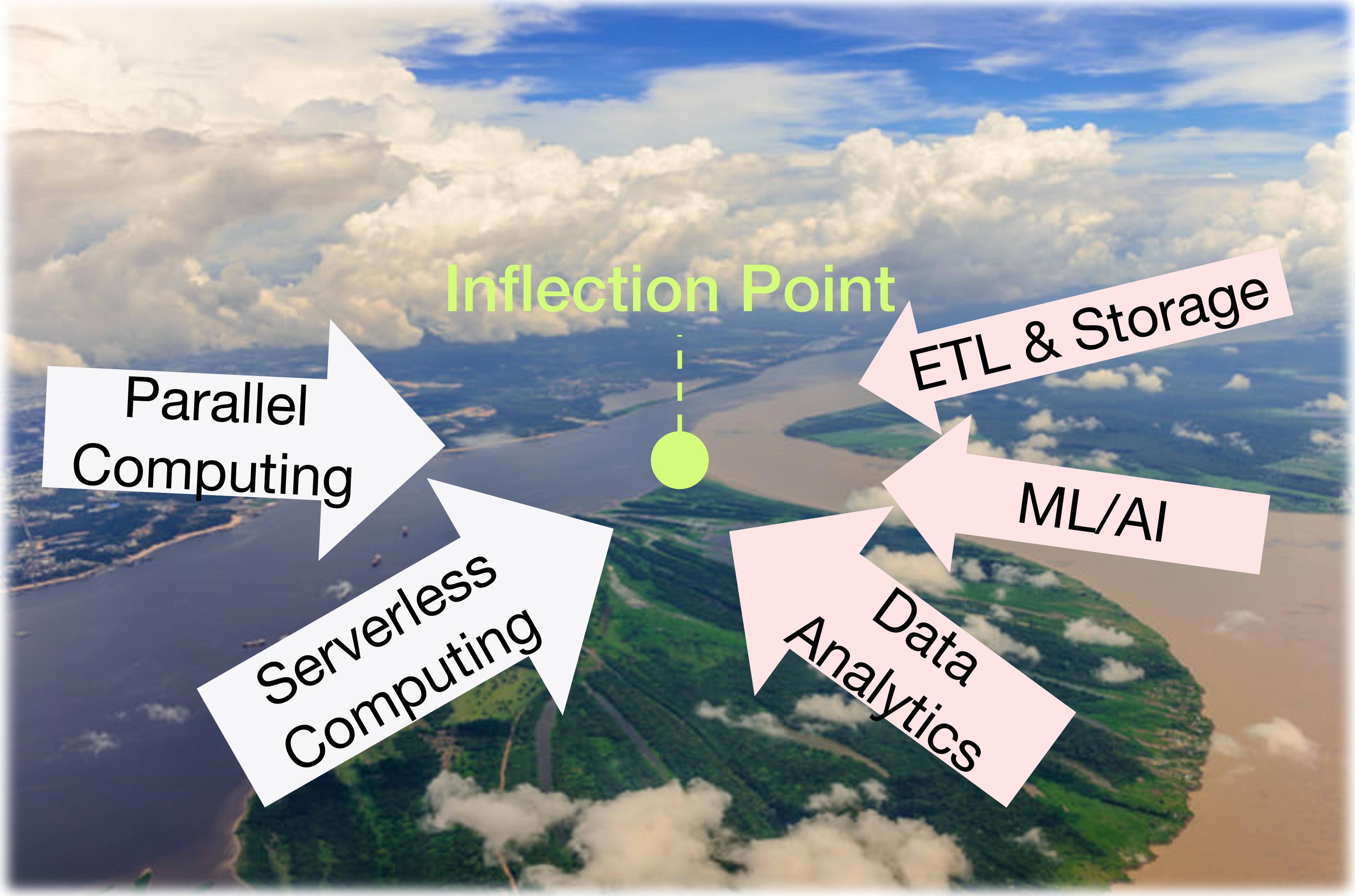
Yue Cheng

UNIVERSITY of VIRGINIA

Learning objectives

- Understand the challenges of supporting “stateful” computations on FaaS
- Know how PyWren works and its limitations
- Know how Wukong addresses some of PyWren’s limitations

Confluence: When stateful apps meet serverless computing



Today’s data analytics landscape

Libraries efficient for O(1MB)

NumPy

pandas

SciPy

scikit learn

matplotlib

Today's data analytics landscape

Libraries efficient for O(1MB)

NumPy

pandas

SciPy

scikit learn

matplotlib

Frameworks for O(100s GB)

APACHE Spark™

DASK

hadoop

MPI

Today’s data analytics landscape

Libraries efficient for O(1MB)

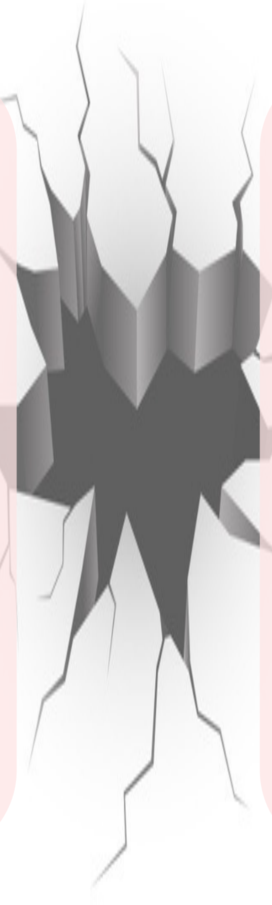
NumPy

pandas

SciPy

scikit learn

matplotlib



Frameworks for O(100s GB)

APACHE Spark™

DASK

hadoop

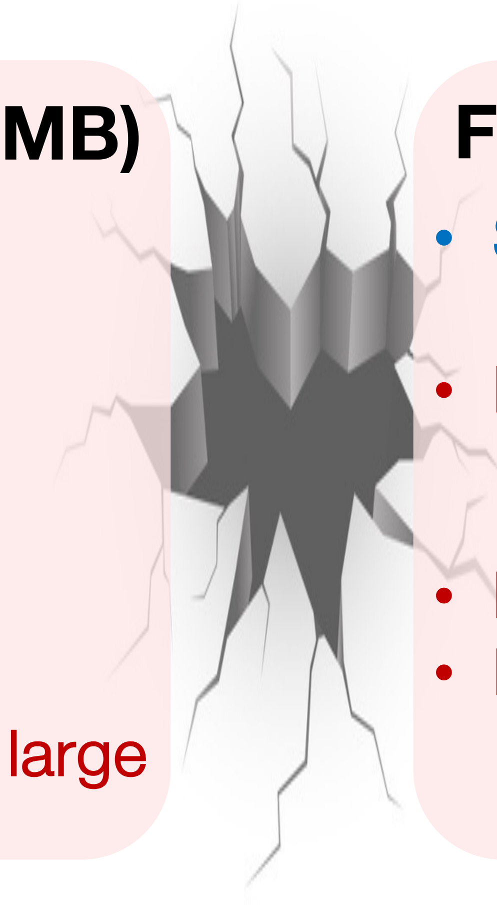
MPI

Today's data analytics landscape

Libraries efficient for O(1MB)

- Easy to program (writing centralized code)
- Low barrier for environment setup (just installing libs)
- Well understood

- No scalability / elasticity
- Not able to efficiently handle large data



Frameworks for O(100s GB)

- Scale to 100s GB data

- Difficult to program and debug
  - Requires distributed systems knowledge
- No elasticity
- High barrier for environment setup
  - Requires low-level administration skills

Today’s data analytics landscape

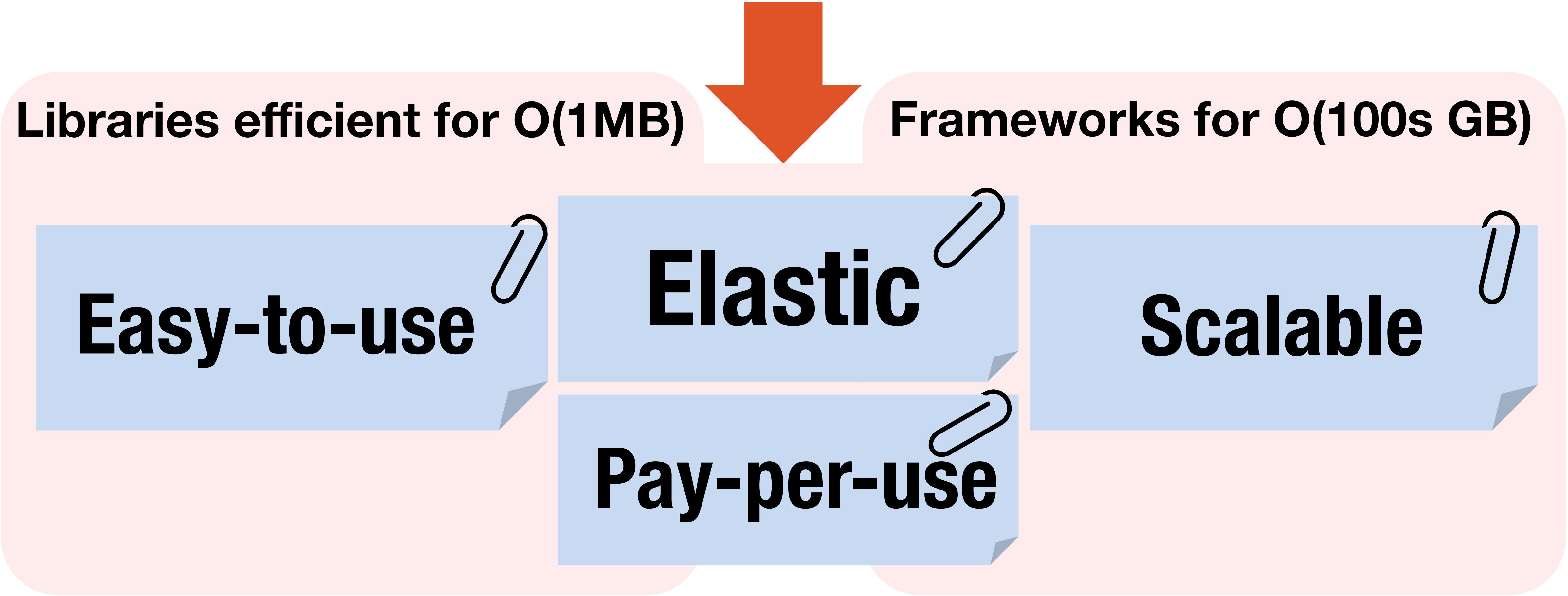
Libraries efficient for O(1MB)

- Easy-to-use
- Not scalable
- Not elastic

Frameworks for O(100s GB)

- Scalable
- Not easy-to-use
- Not elastic

Can we achieve all these desirable properties with **Serverless?**



Libraries efficient for O(1MB)

Frameworks for O(100s GB)

Easy-to-use

Elastic

Pay-per-use

Scalable

Recap: What is serverless computing?

Many people define it many ways

A **programming abstraction** that enables users to upload programs, run them at **virtually** any scale, and pay **only for the resources used**

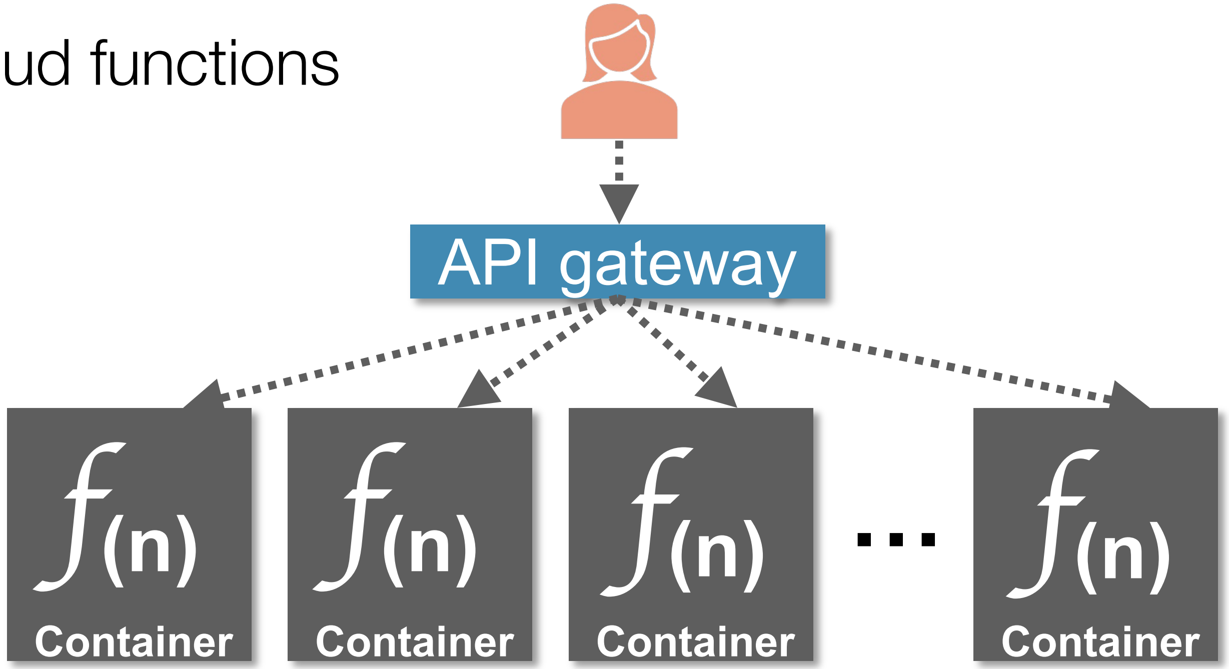
- **Function-as-a-Service (FaaS):** Cloud functions as a basic deployment unit

AWS Lambda

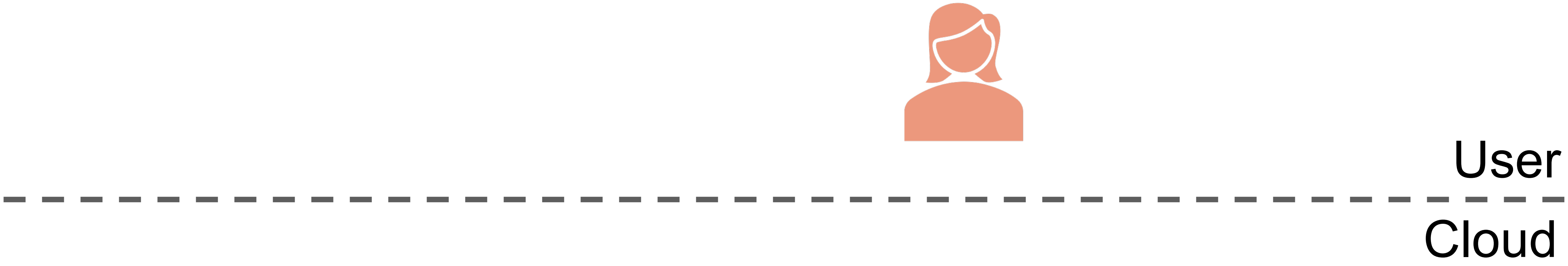
Azure Functions

Google Cloud Functions

Alibaba Function Compute



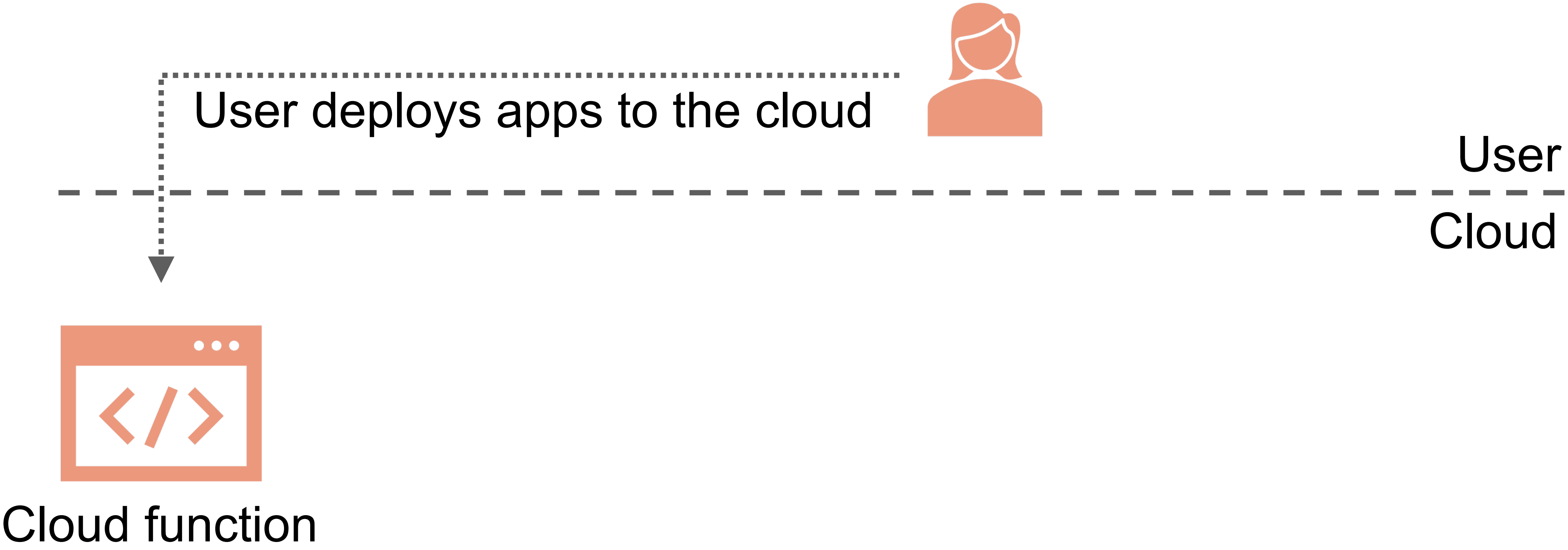
Function-as-a-Service (FaaS)



User

Cloud

Function-as-a-Service (FaaS)



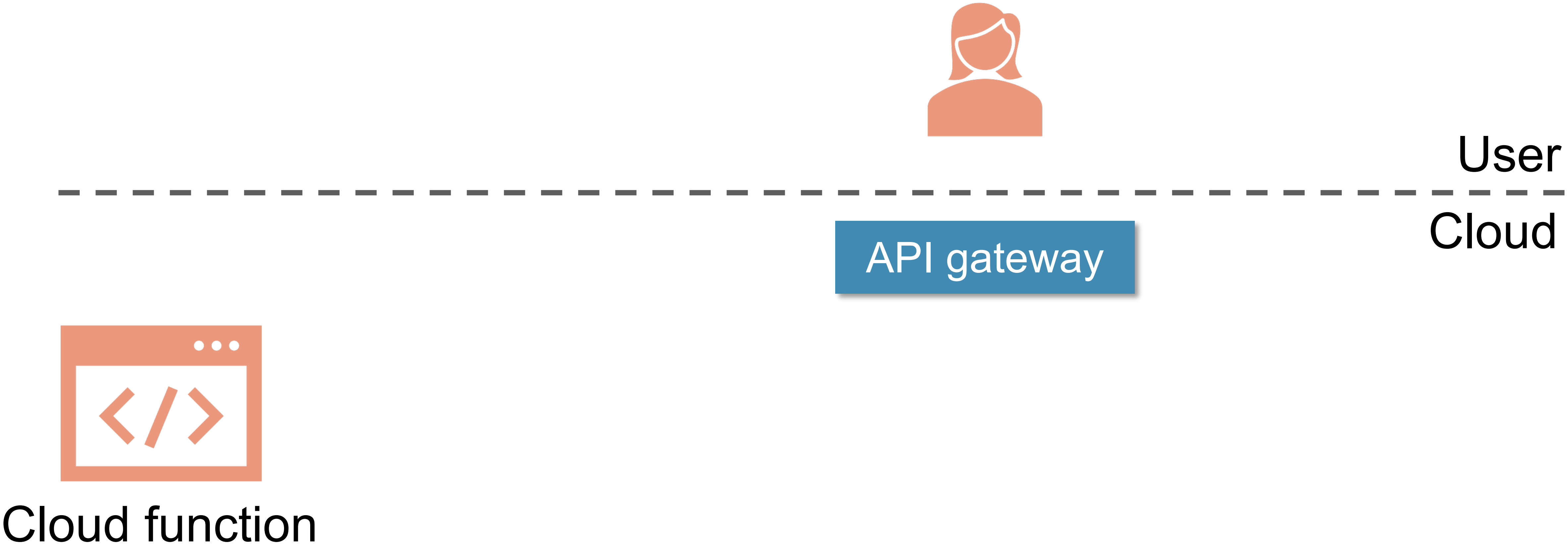
User deploys apps to the cloud

User

Cloud

Cloud function

Function-as-a-Service (FaaS)



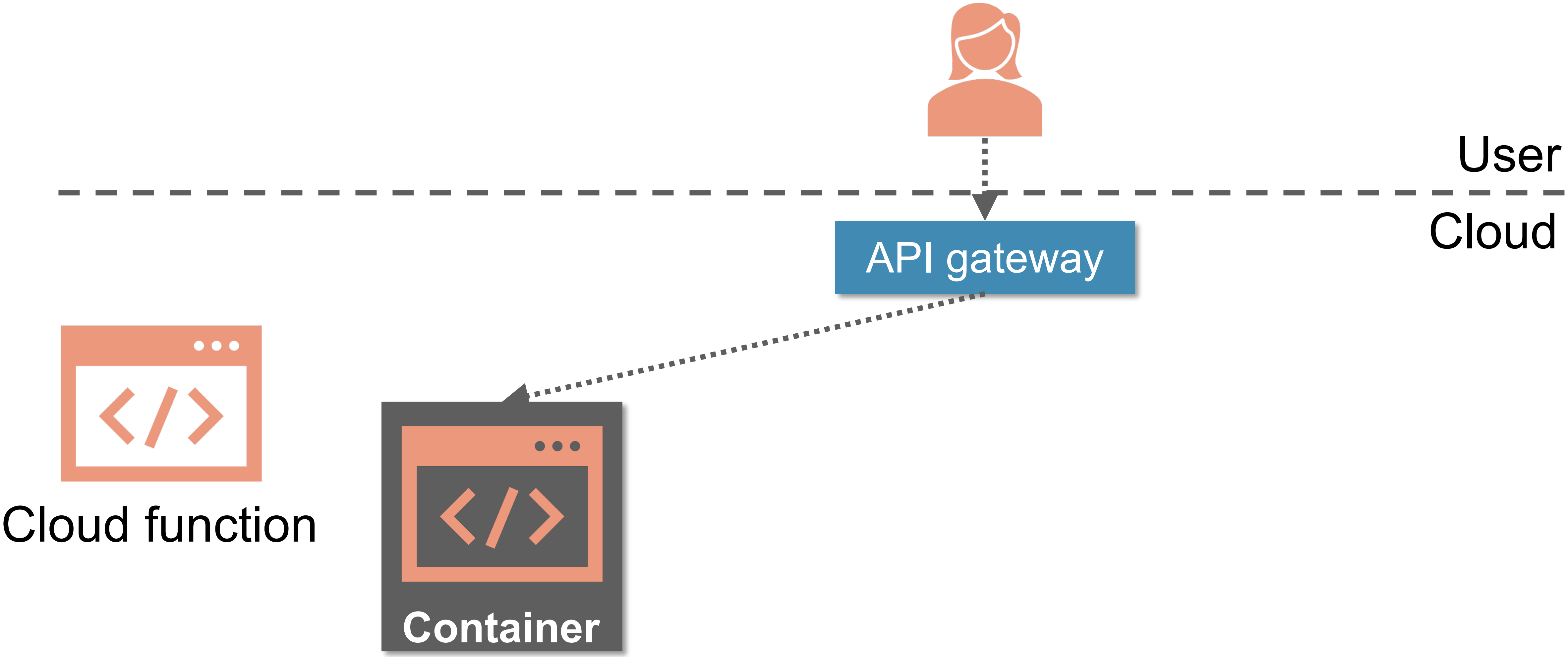
User

Cloud

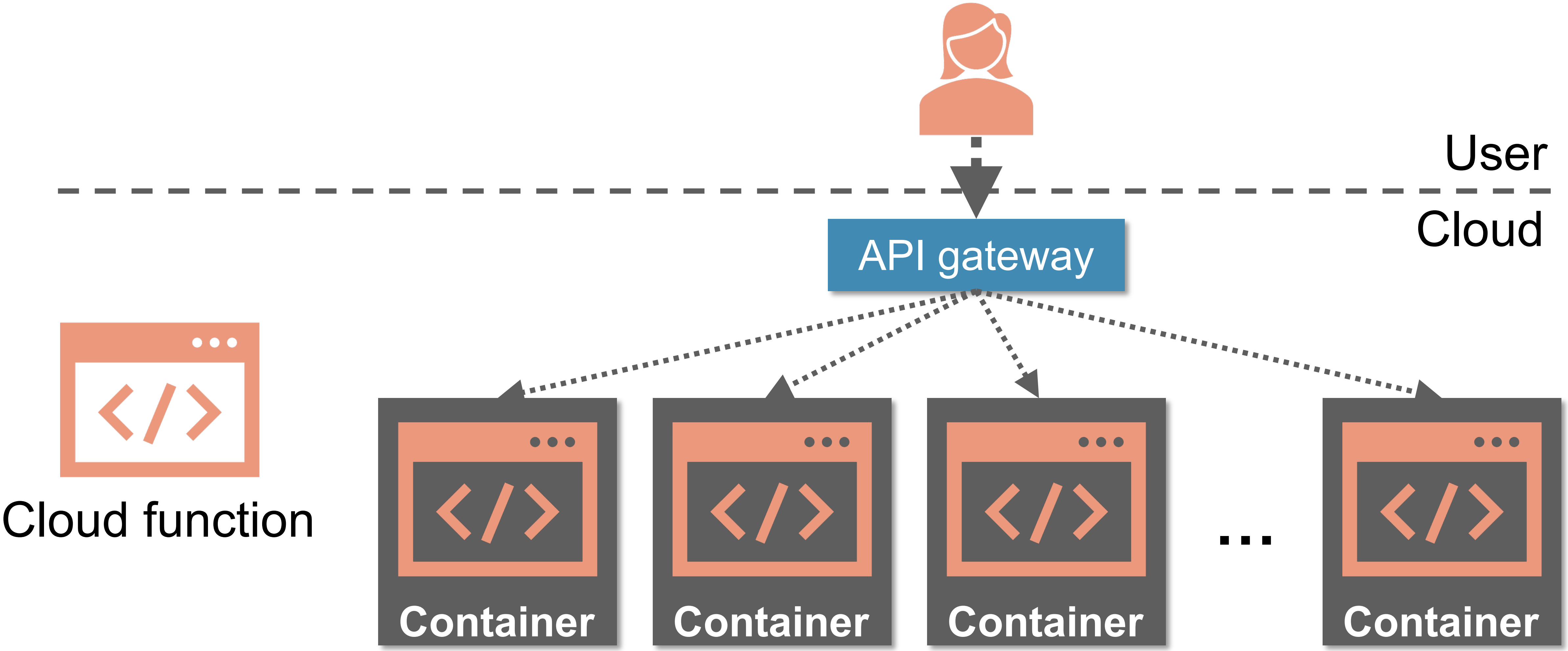
API gateway

Cloud function

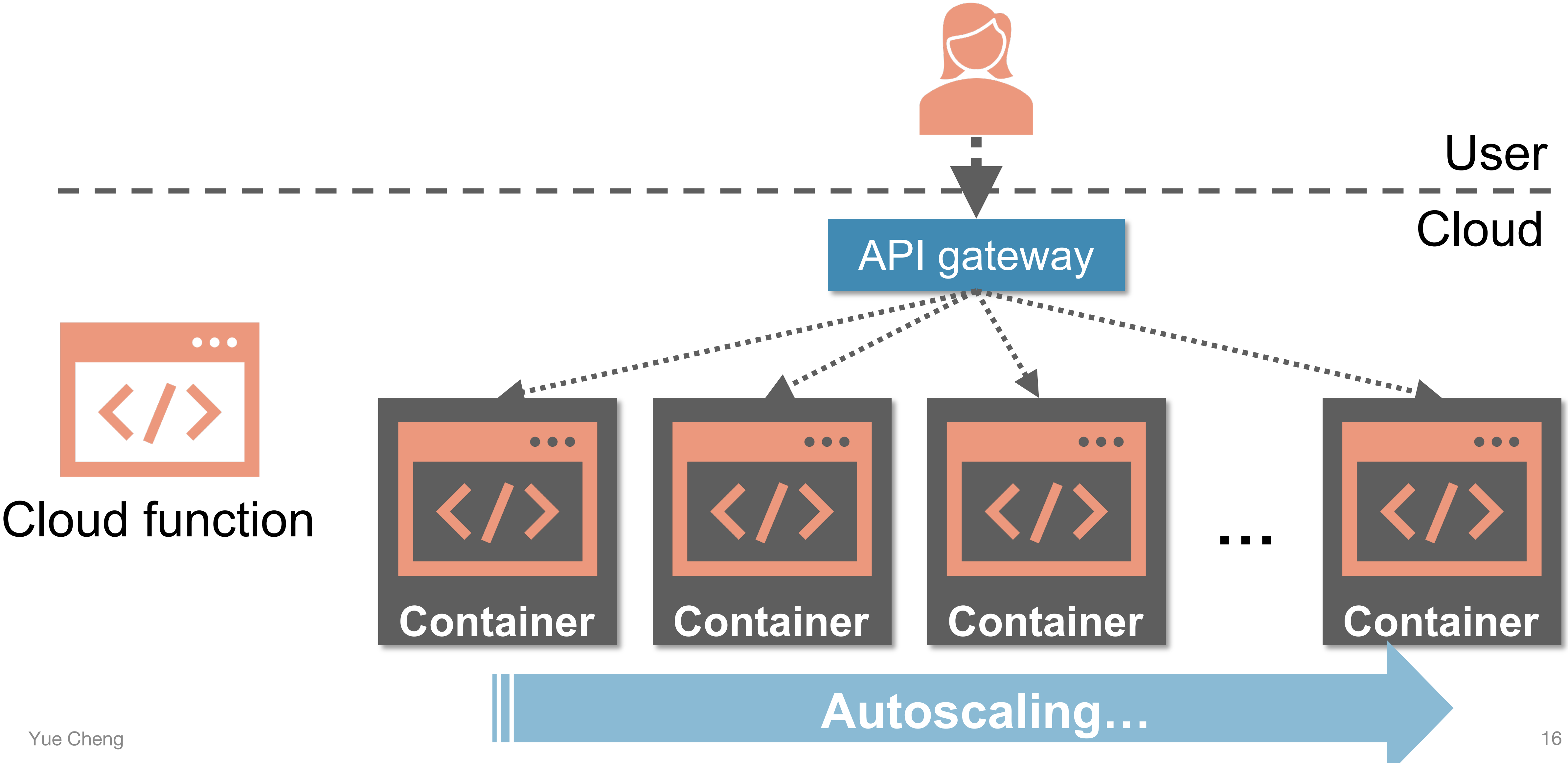
Function-as-a-Service (FaaS)



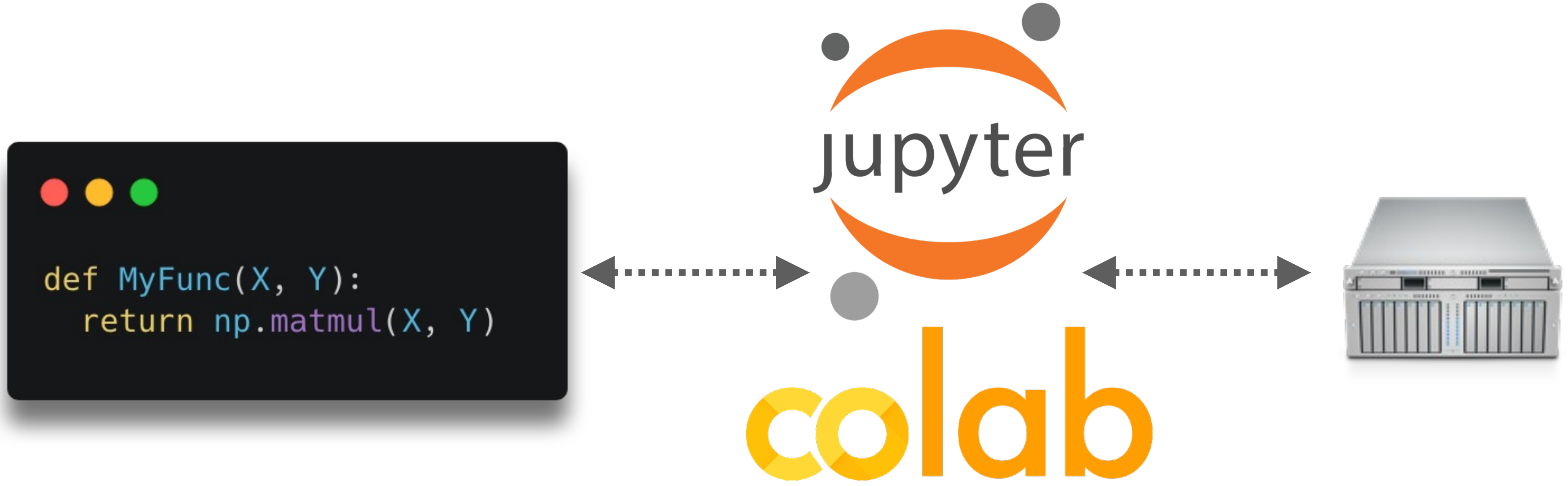
Function-as-a-Service (FaaS)



Function-as-a-Service (FaaS)

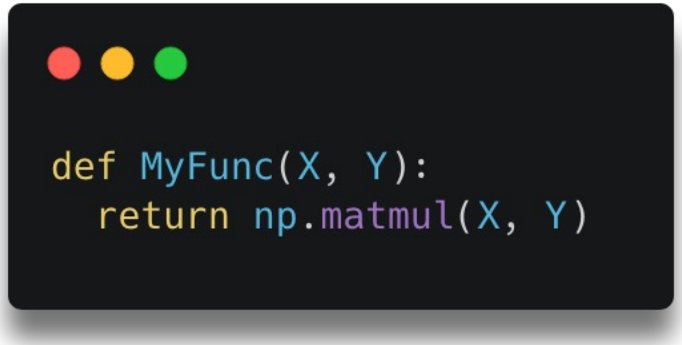


Python analytics: What we have today

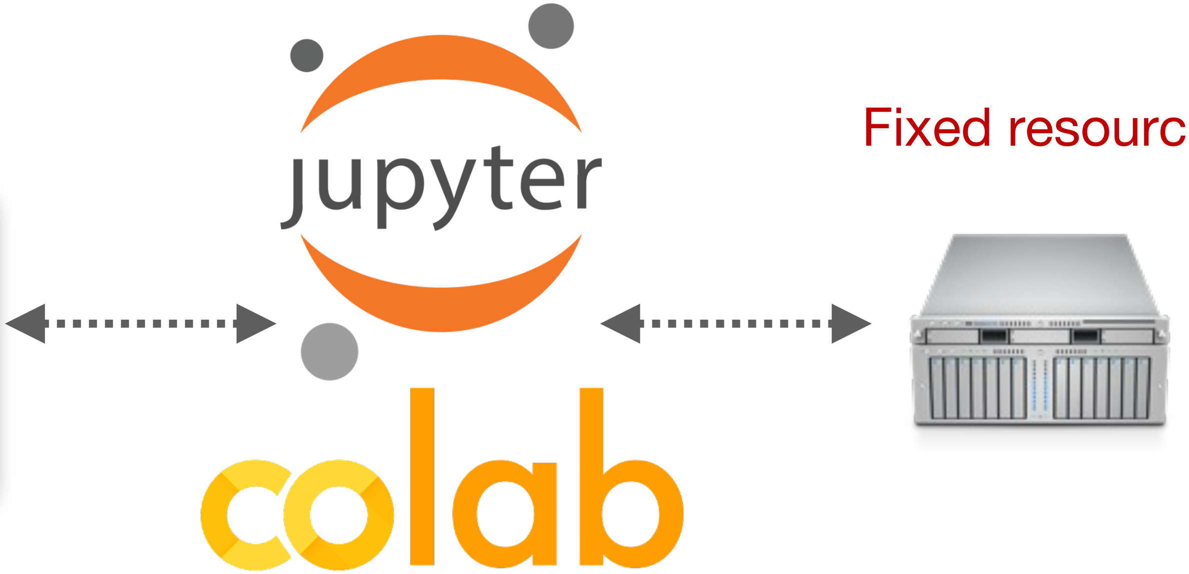


```
def MyFunc(X, Y):
  return np.matmul(X, Y)
```

Python analytics: What we have today



```
def MyFunc(X, Y):
  return np.matmul(X, Y)
```

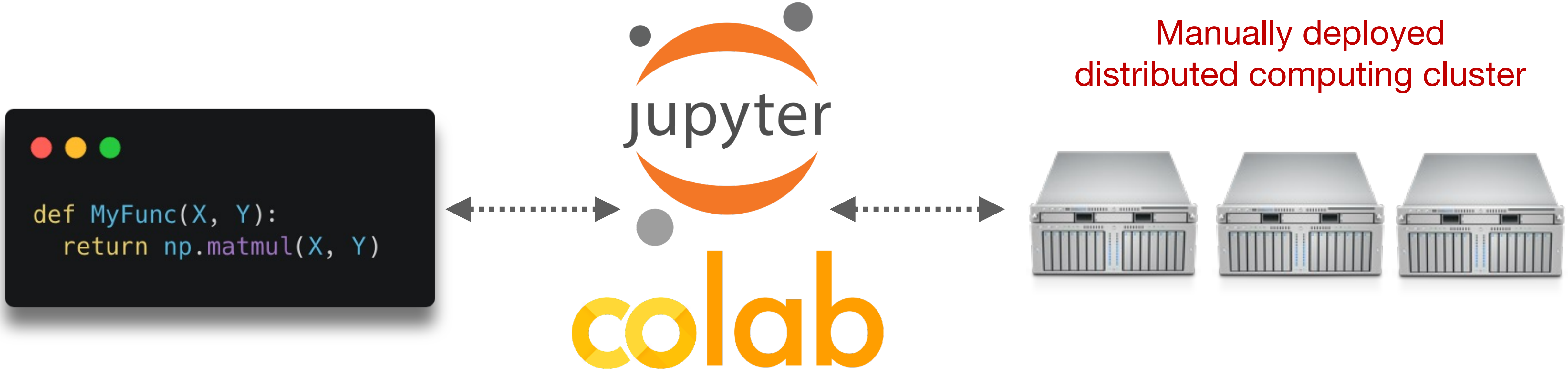


Fixed resource

User writes interactive analytics and runs it on a notebook server

- No autoscaling for large computations
- Too slow? OOM? Need to scale out manually!
- Too expensive? Idled resources charge $$

Python analytics: What we have today



```
def MyFunc(X, Y):
  return np.matmul(X, Y)
```

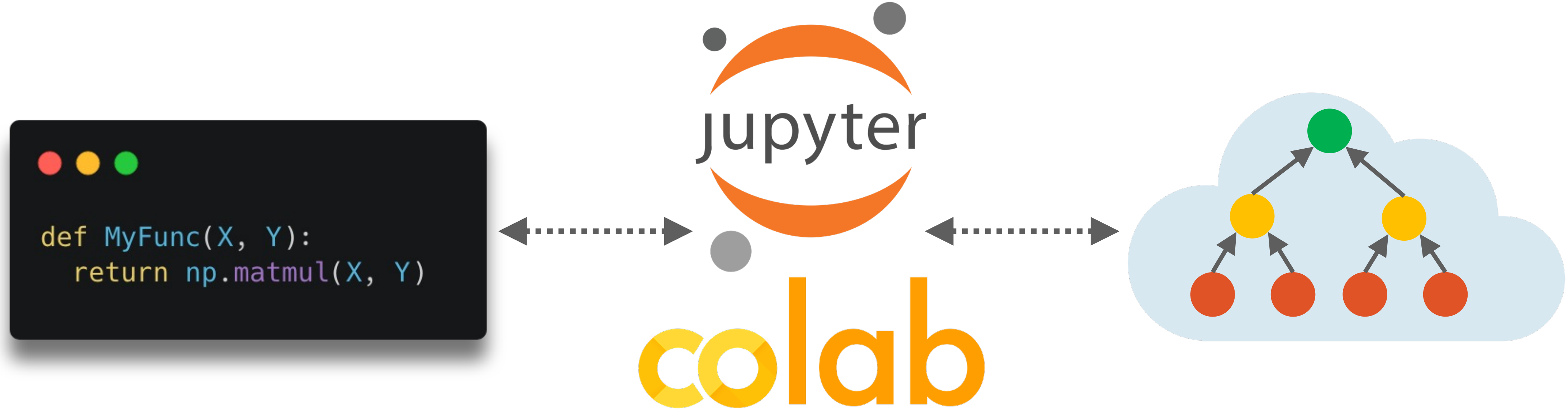
Manually deployed
distributed computing cluster

User writes interactive analytics and runs it on a notebook server

- No autoscaling for large computations
- Too slow? OOM? Need to scale out manually!
- Too expensive? Idled resources charge $$

High barriers to enter for those who lack CS/systems background

Python analytics: What we would like to have

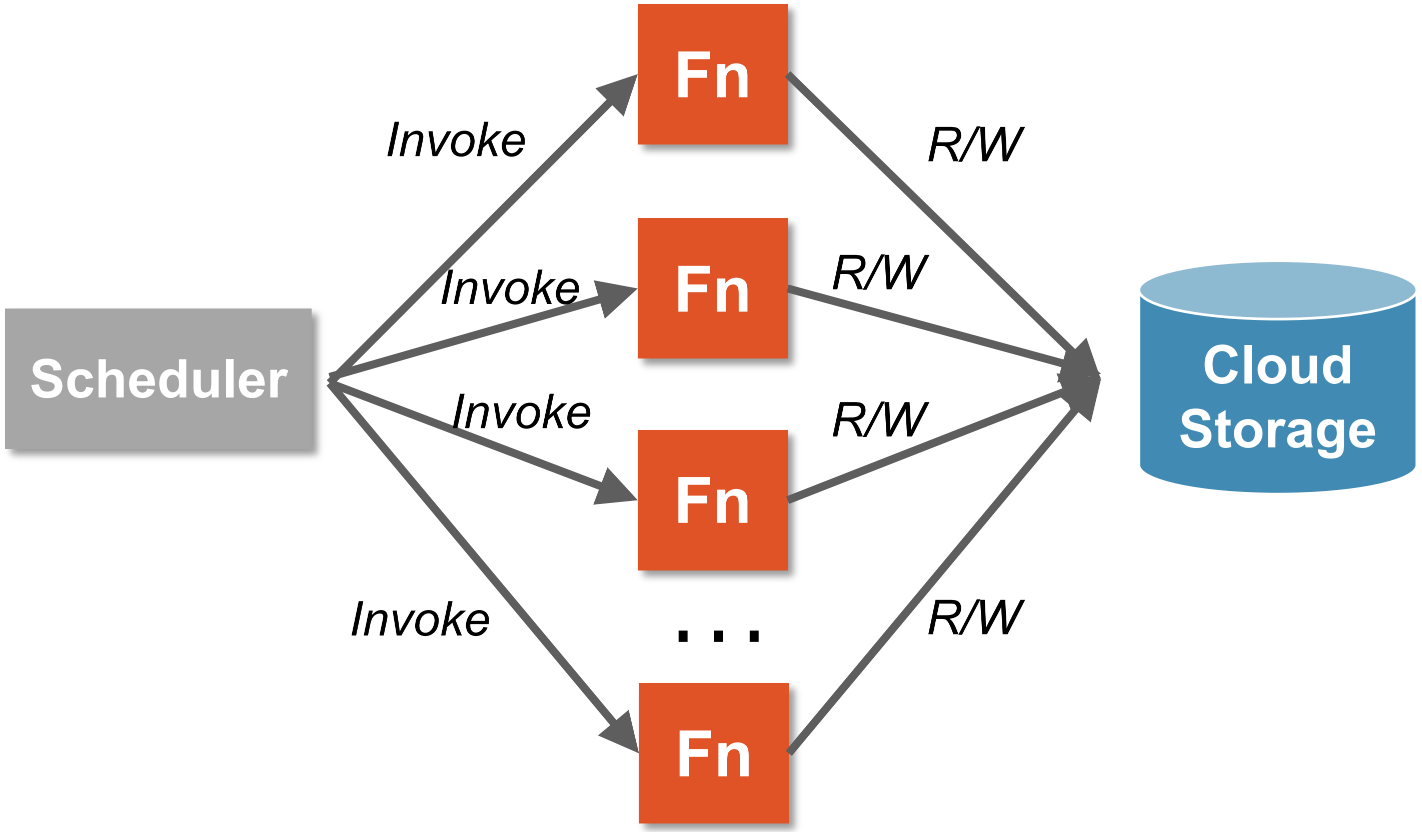


```
def MyFunc(X, Y):
  return np.matmul(X, Y)
```

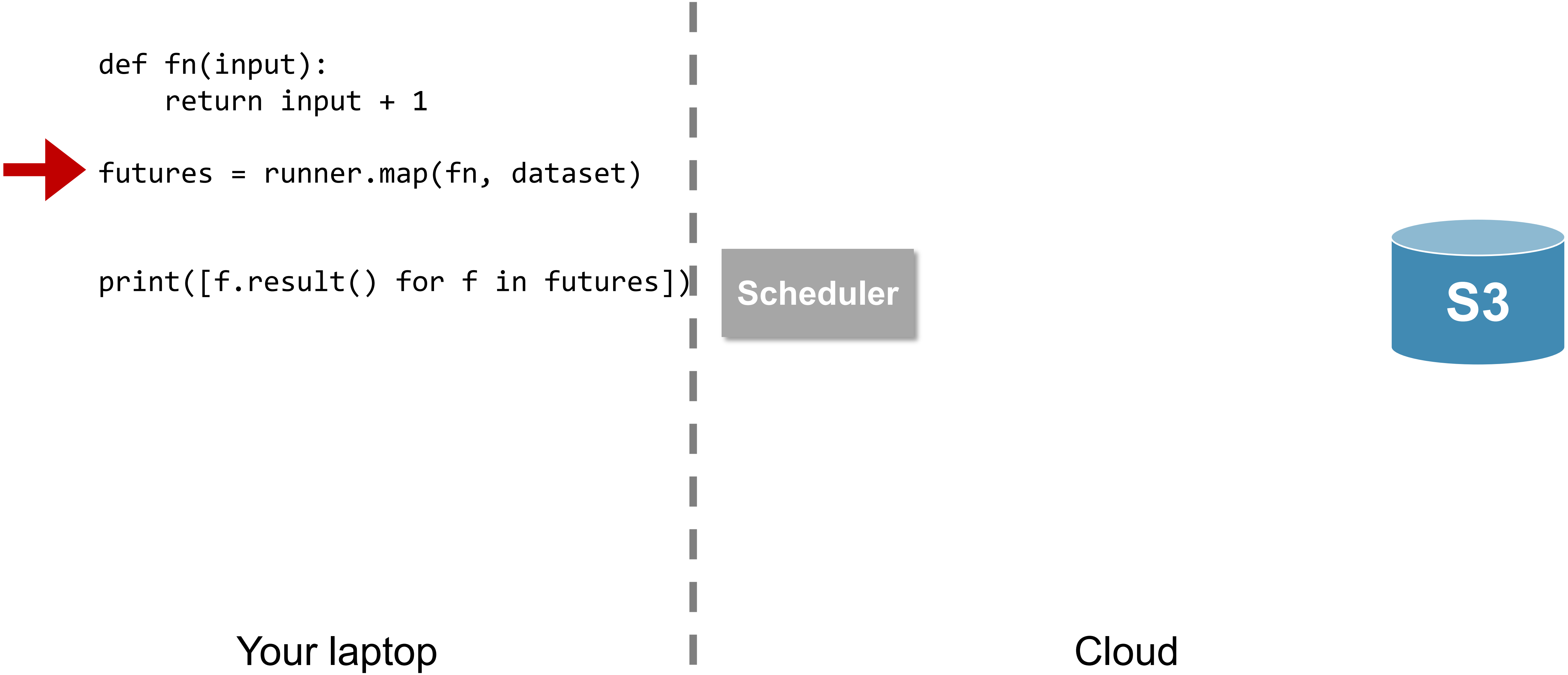
User writes interactive analytics and runs it **on FaaS**

- Elastically and automatically scales to the right size
- Pay-per-use with minimal $$ cost
- Expertise of writing parallel programs **NOT required**
- Manual cluster maintenance **NOT required**

PyWren: Stateful computing over stateless serverless functions

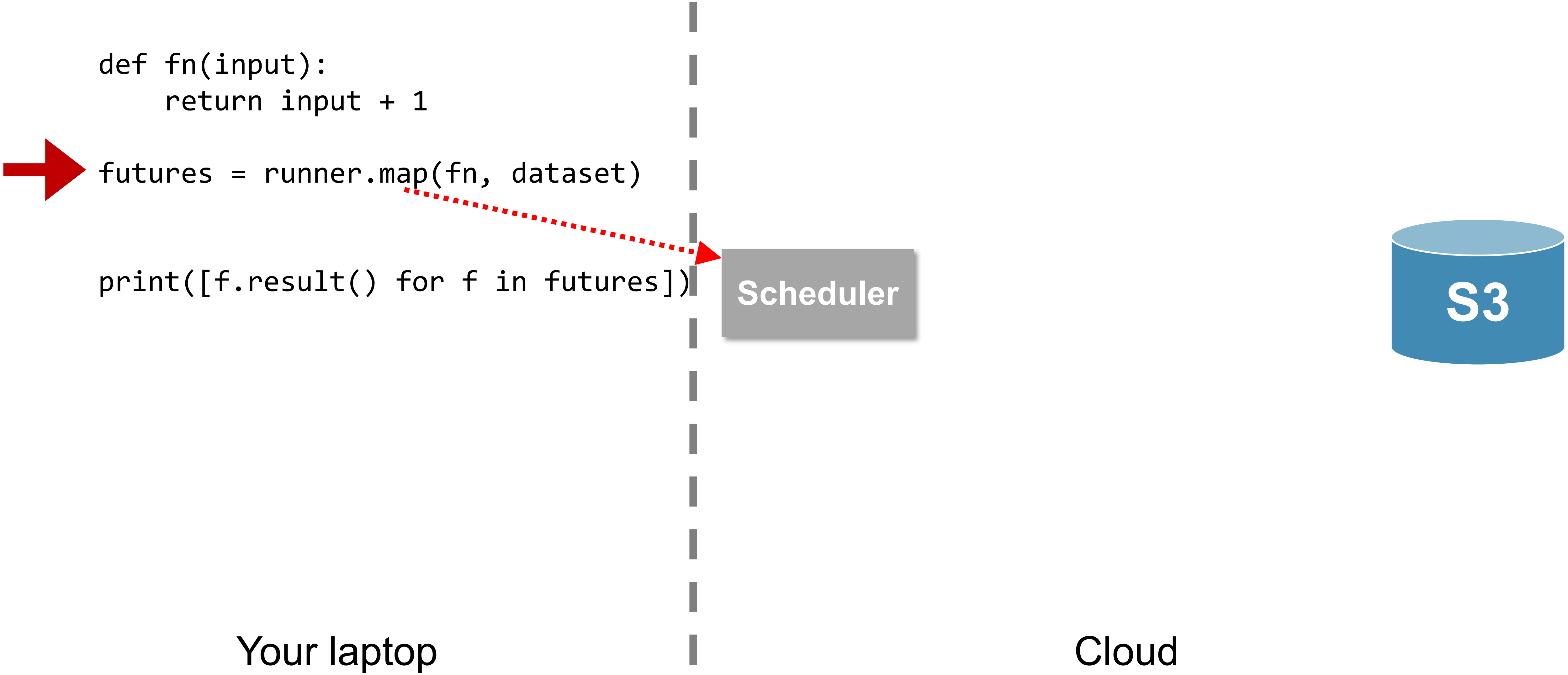


PyWren: How it works



```
def fn(input):
    return input + 1

futures = runner.map(fn, dataset)


print([f.result() for f in futures])
```

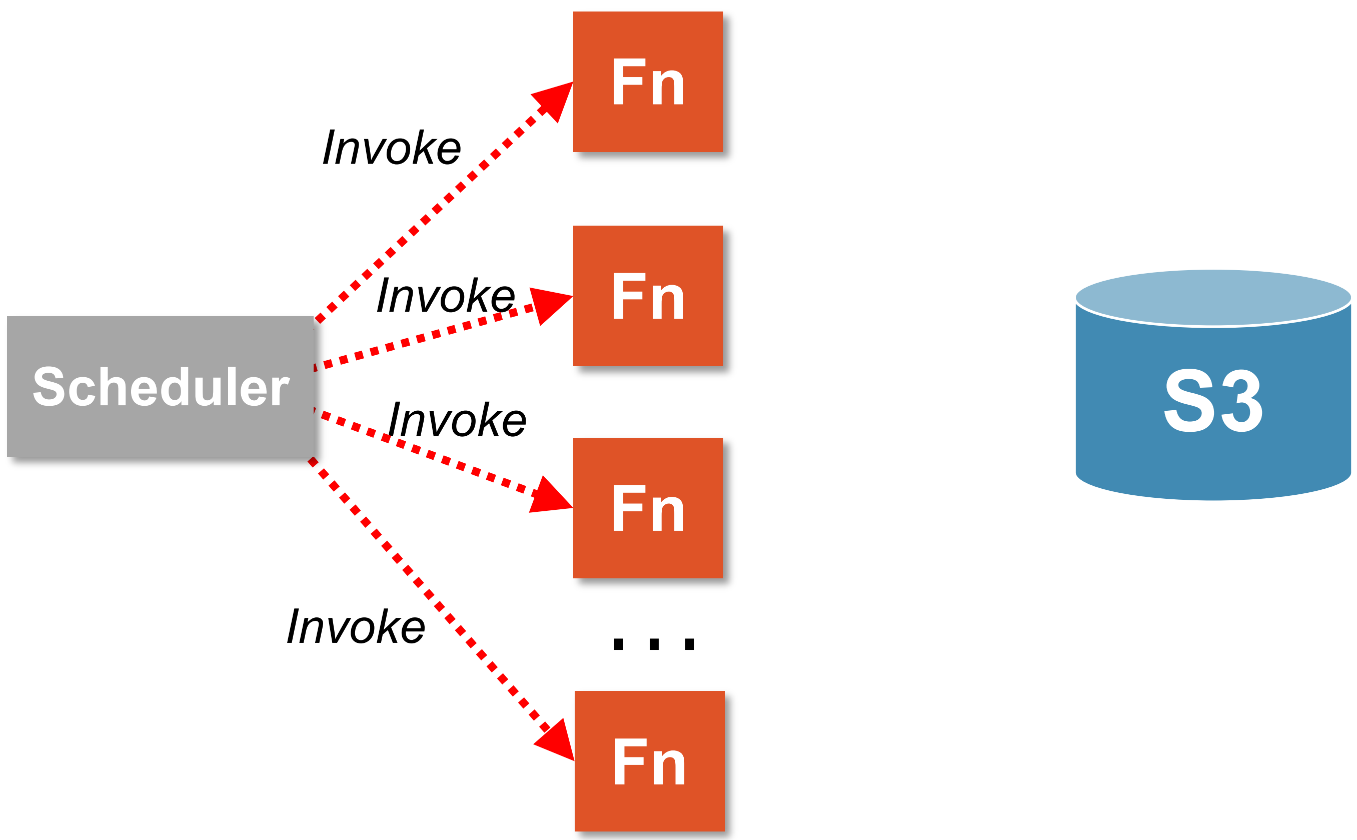
PyWren: How it works



```
def fn(input):
    return input + 1

futures = runner.map(fn, dataset)


print([f.result() for f in futures])
```

PyWren: How it works

```
def fn(input):
    return input + 1

futures = runner.map(fn, dataset)


print([f.result() for f in futures])
```



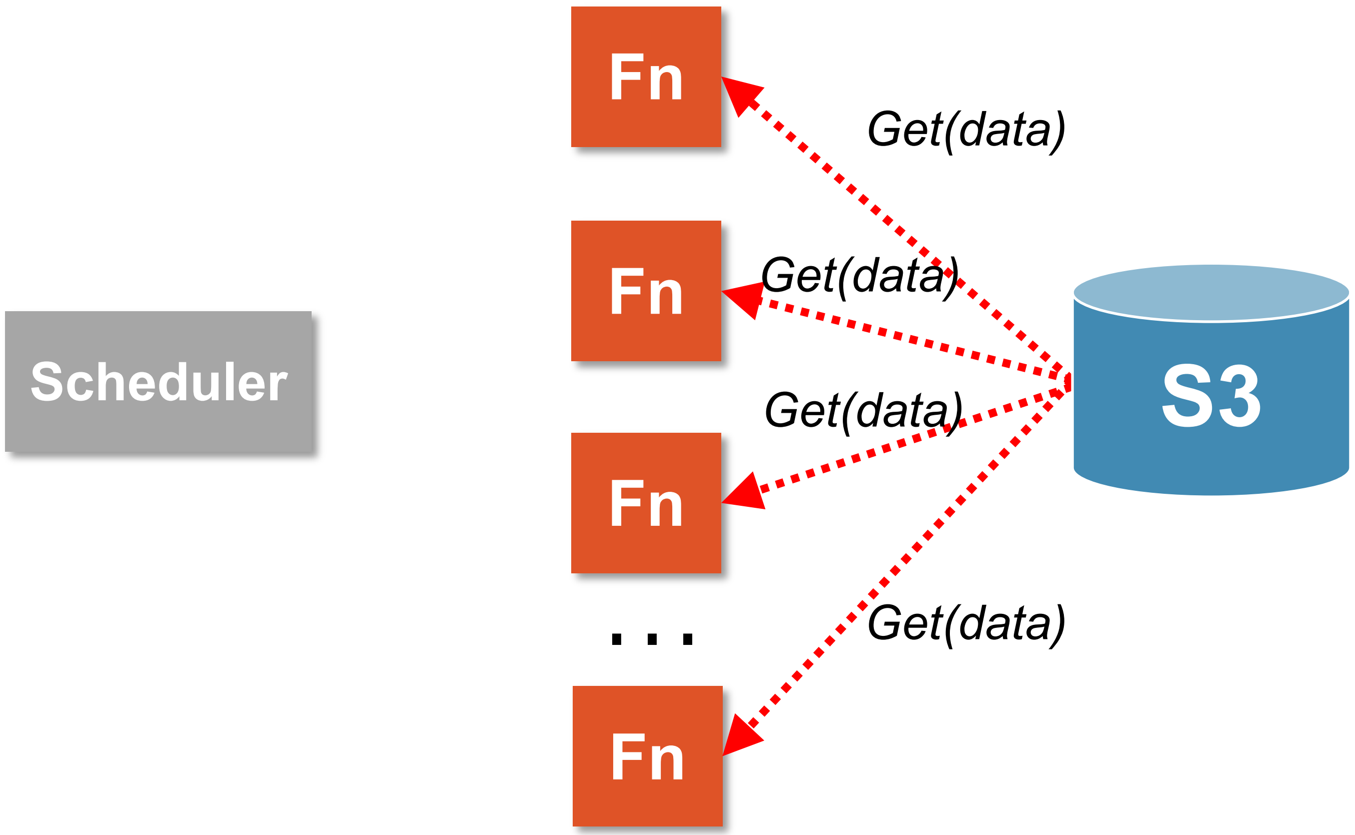
Your laptop

Cloud

PyWren: How it works

```
def fn(input):
    return input + 1

futures = runner.map(fn, dataset)


print([f.result() for f in futures])
```

Your laptop

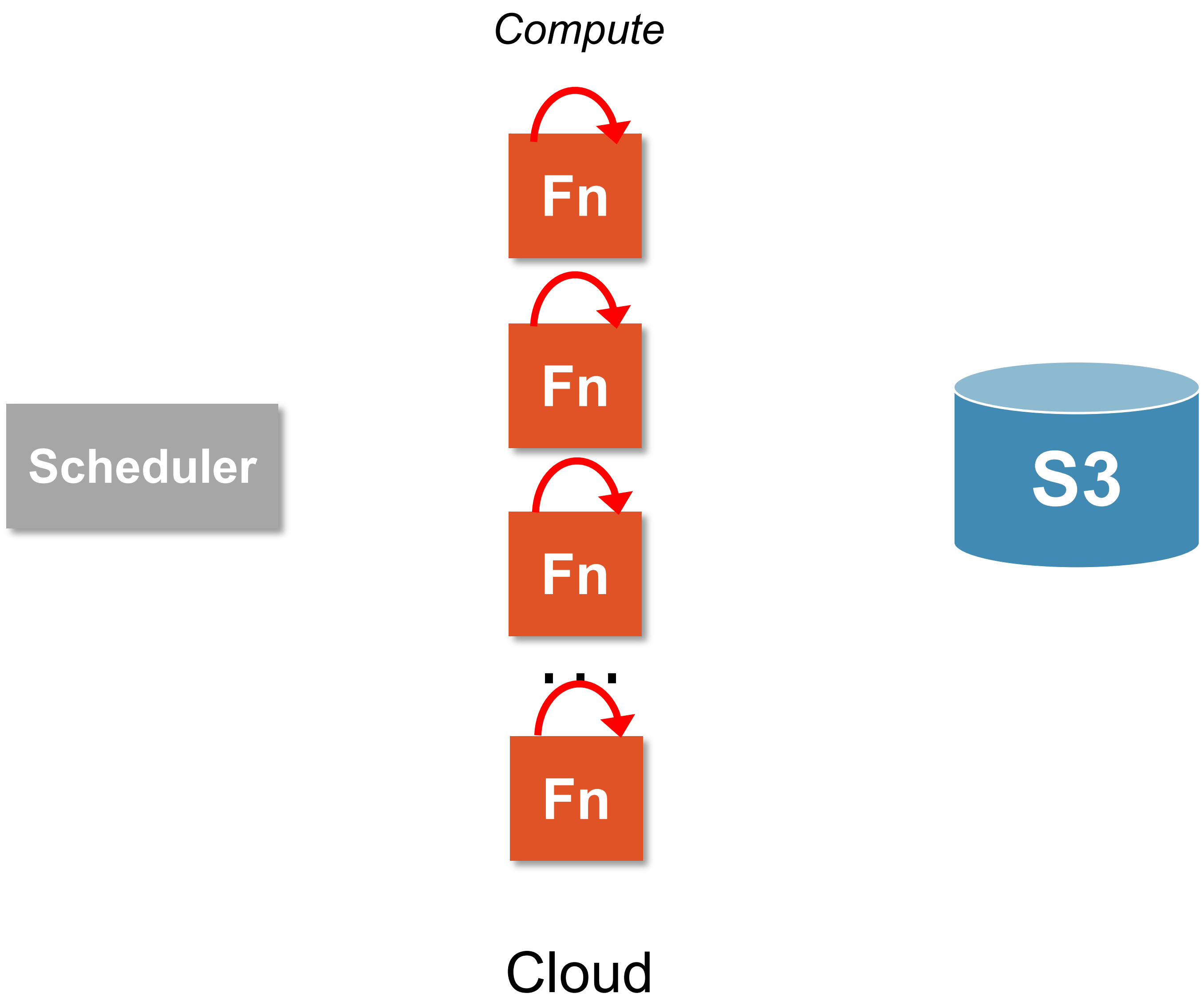


Cloud

PyWren: How it works

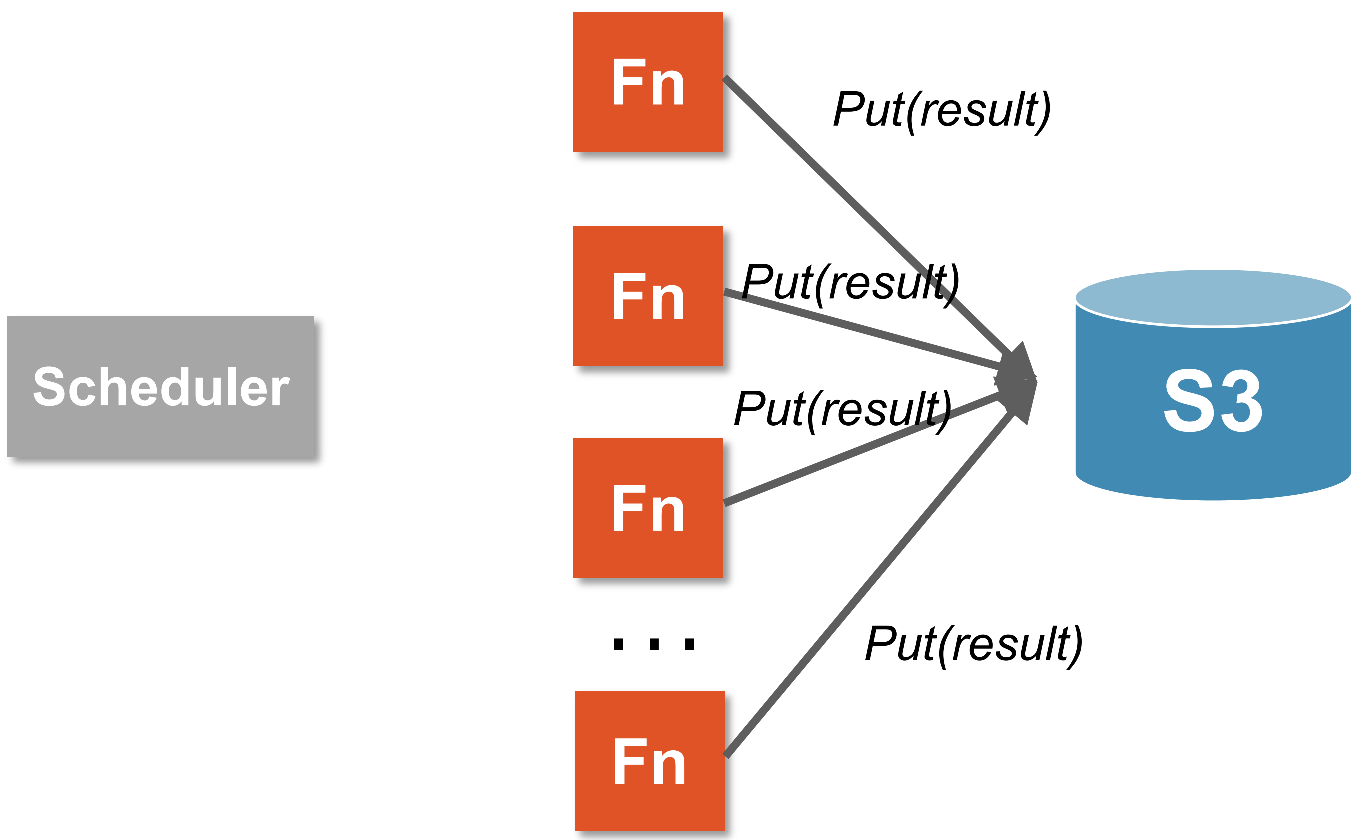
```
def fn(input):
    return input + 1

futures = runner.map(fn, dataset)


print([f.result() for f in futures])
```

Your laptop



PyWren: How it works

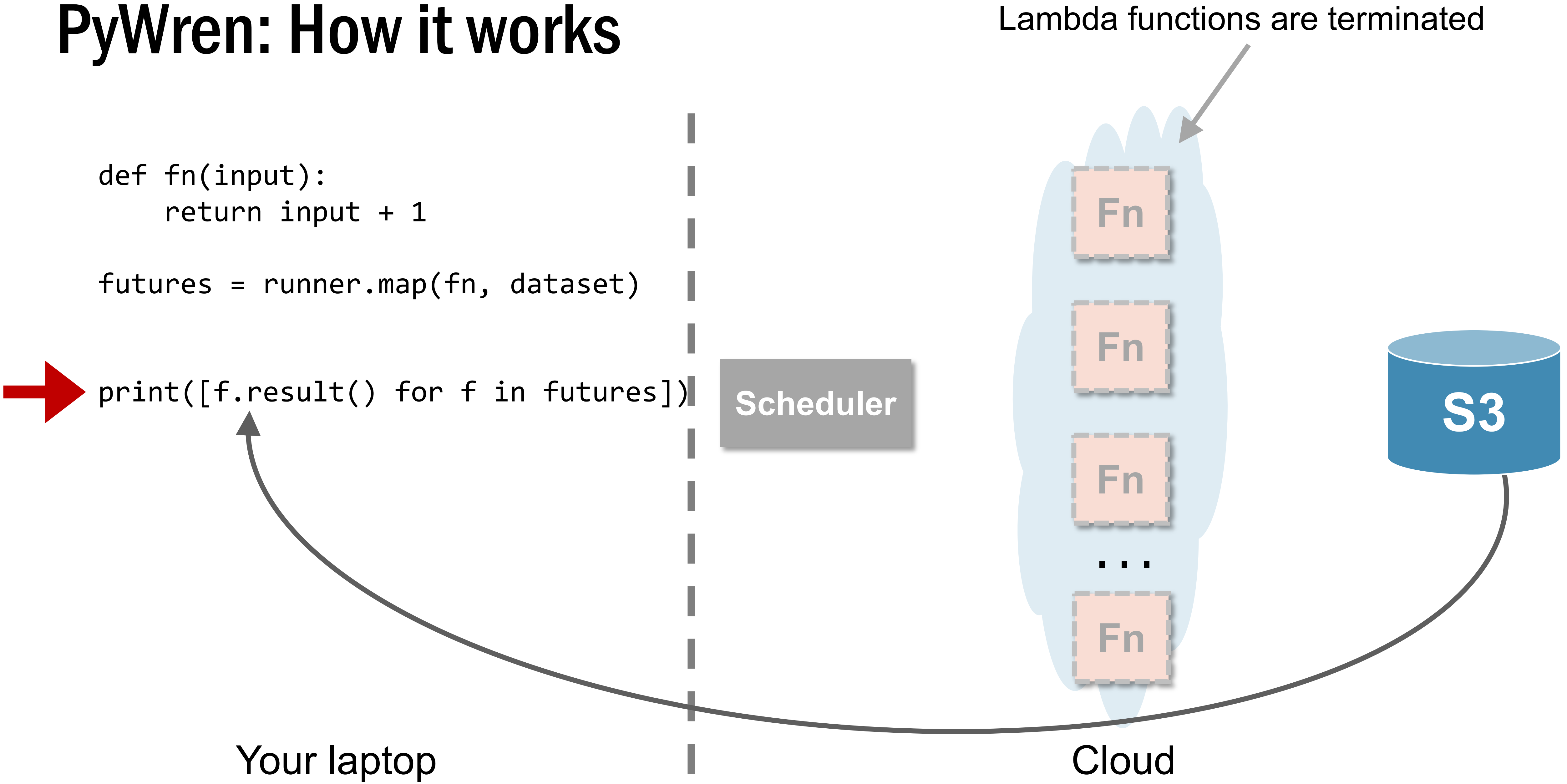
```
def fn(input):
    return input + 1

futures = runner.map(fn, dataset)


print([f.result() for f in futures])
```



Your laptop

Cloud

PyWren: How it works

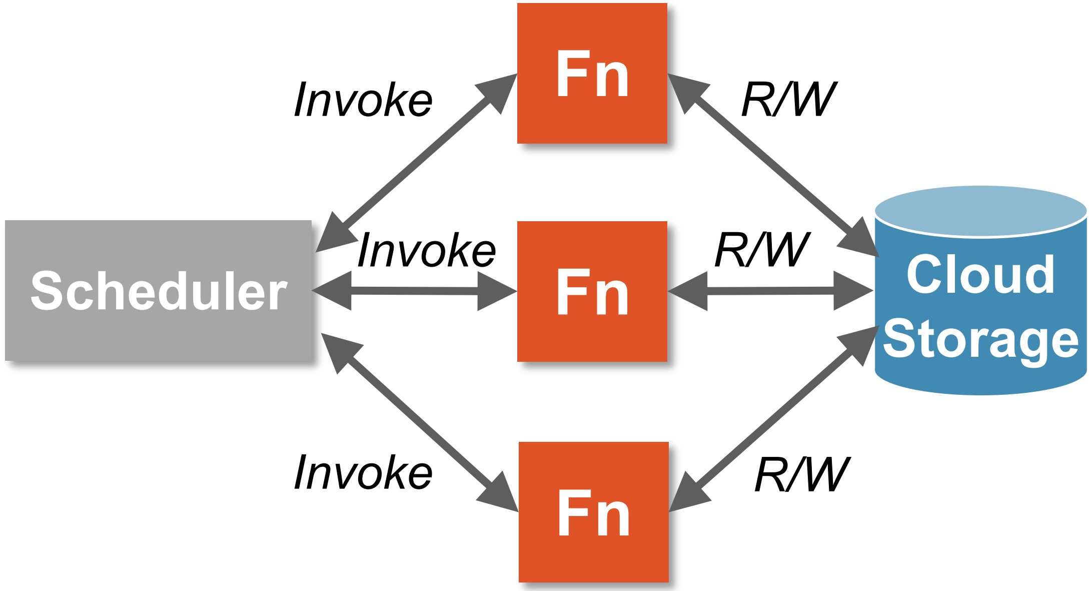


```
def fn(input):
    return input + 1

futures = runner.map(fn, dataset)


print([f.result() for f in futures])
```

Quantifying the pain of FaaS

How FaaS adds huge amounts of **performance taxes**

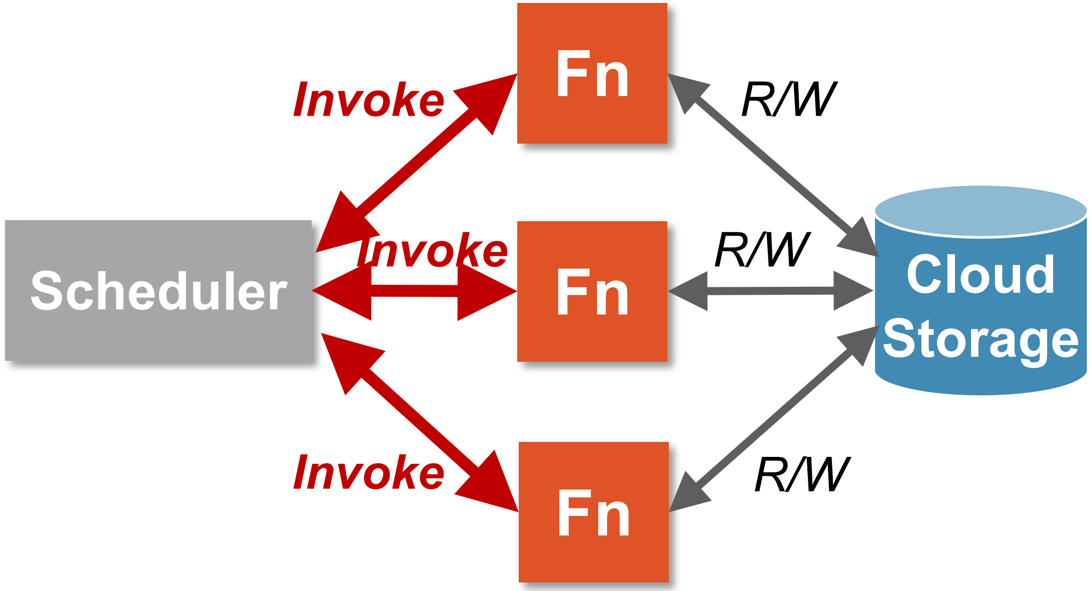
Python analytics on FaaS is slow!

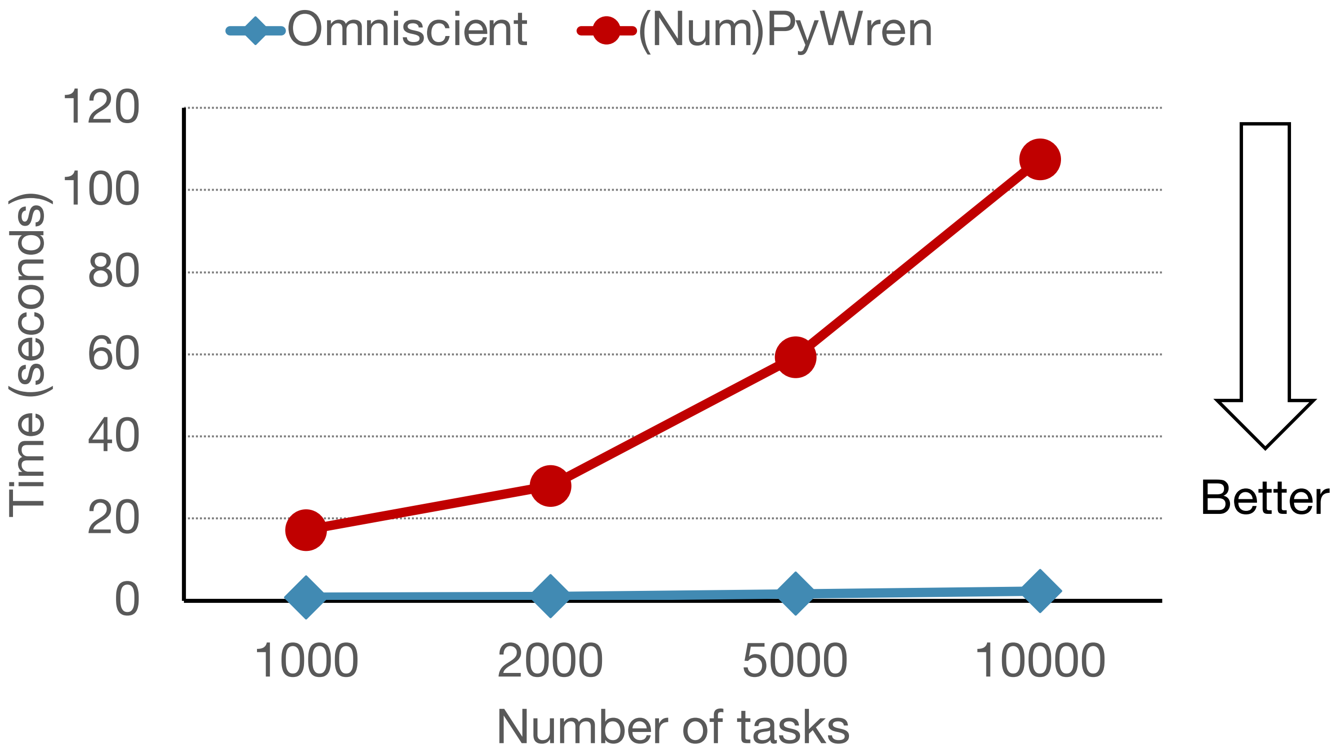


PyWren and numpywren

* [PyWren] Occupy the Cloud: Distributed Computing for the 99%. In ACM SoCC'17.
* [numpywren] Serverless linear algebra. In ACM SoCC'20.

Python analytics on FaaS is slow!





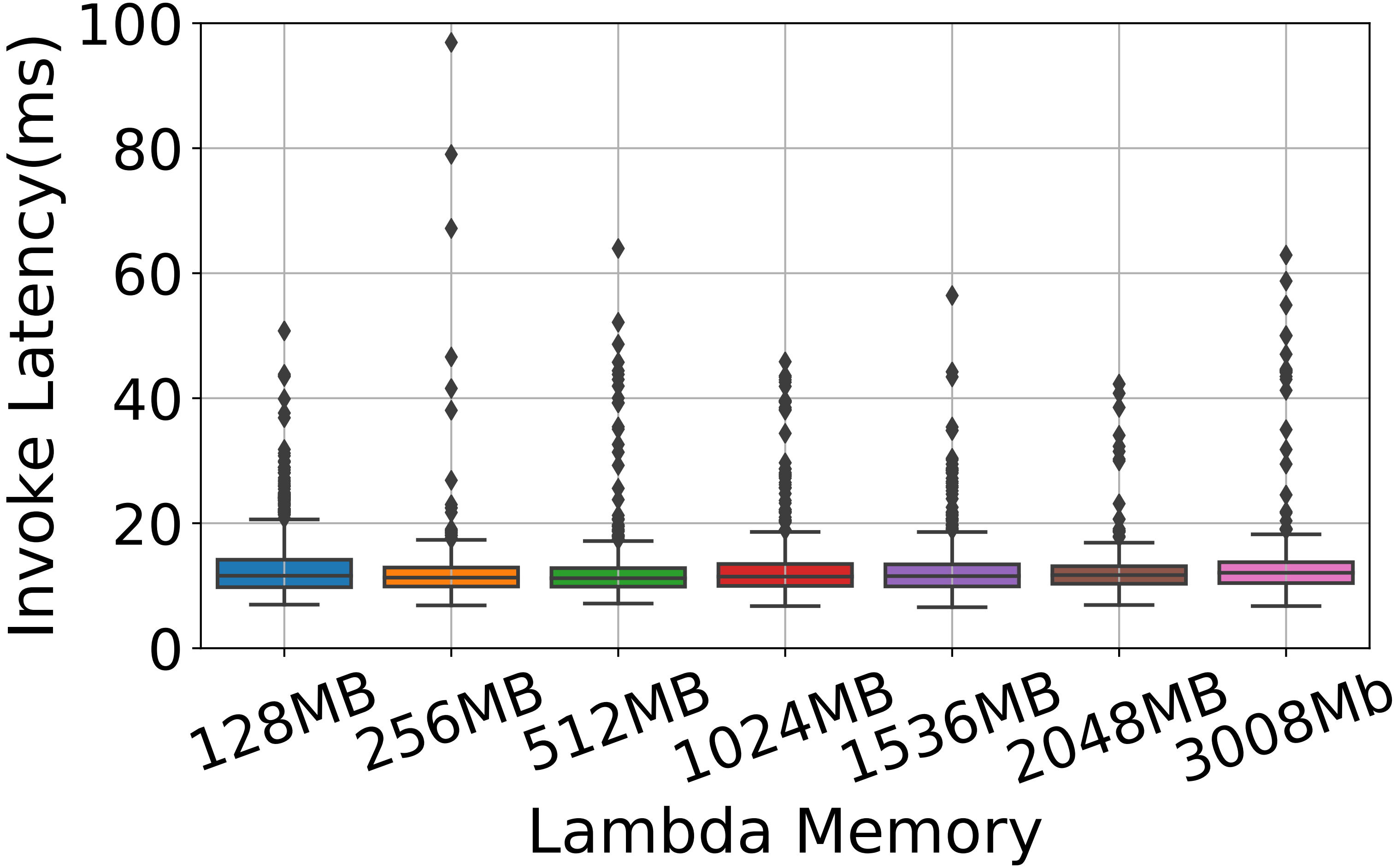
State-of-the-art FaaS frameworks pay huge amounts of FaaS taxes

- **Task scheduling bottleneck:** Too slow to scale to thousands of functions

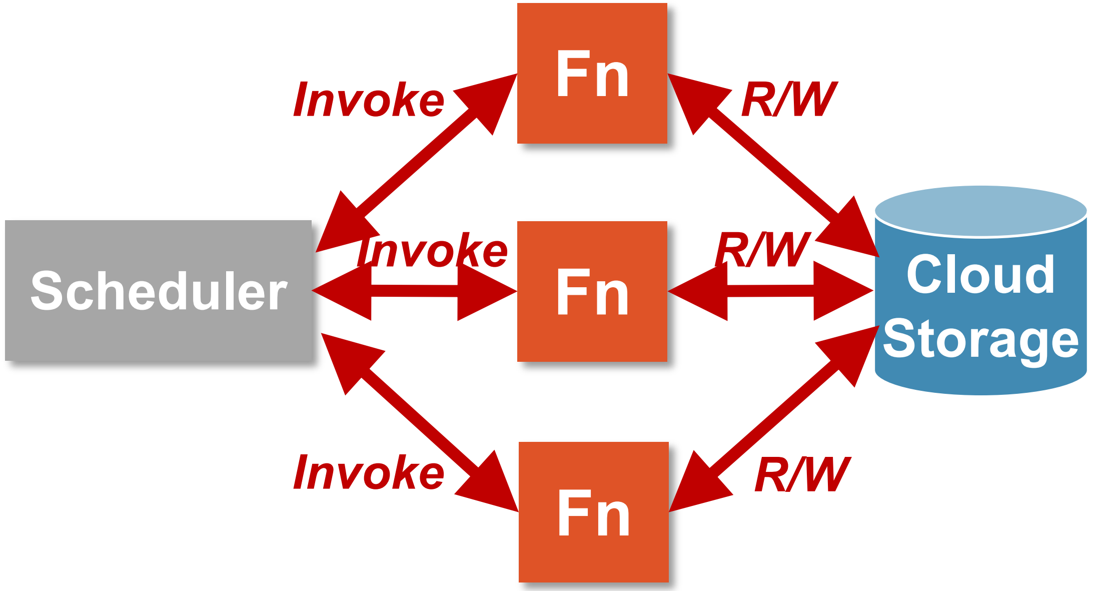
* [PyWren] Occupy the Cloud: Distributed Computing for the 99%. In ACM SoCC'17.

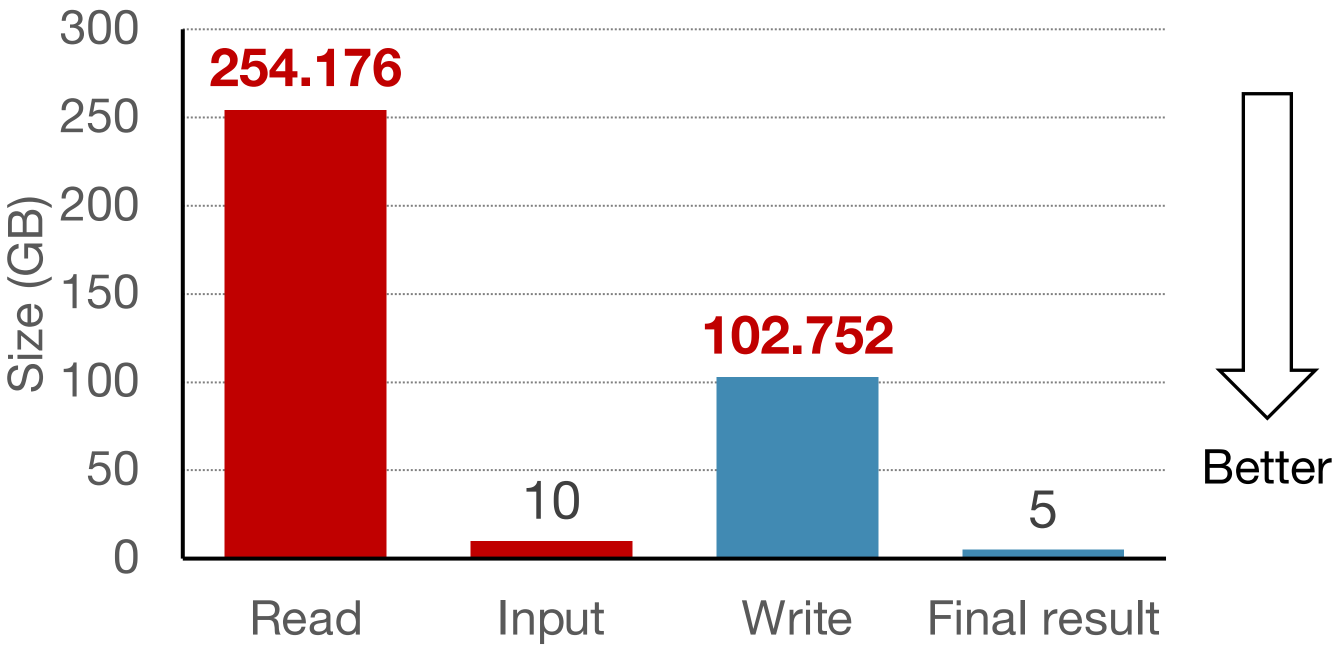
* [numpywren] Serverless linear algebra. In ACM SoCC'20.

High HTTP invocation cost for AWS Lambda



Python analytics on FaaS is slow!





numpywren GEMM read & write amplification

State-of-the-art FaaS frameworks pay huge amounts of FaaS taxes

- **Task scheduling bottleneck:** Too slow to scale to thousands of functions
- **I/O bottleneck:** Excessive data movement cost due to FaaS constraint

* [PyWren] Occupy the Cloud: Distributed Computing for the 99%. In ACM SoCC'17.
* [numpywren] Serverless linear algebra. In ACM SoCC'20.

Naively porting a stateful cluster computing application to FaaS won’t work!

Need a FaaS-centric approach

Insight: A FaaS framework may not care about traditional metrics (load balancing, cluster util.)

Wukong

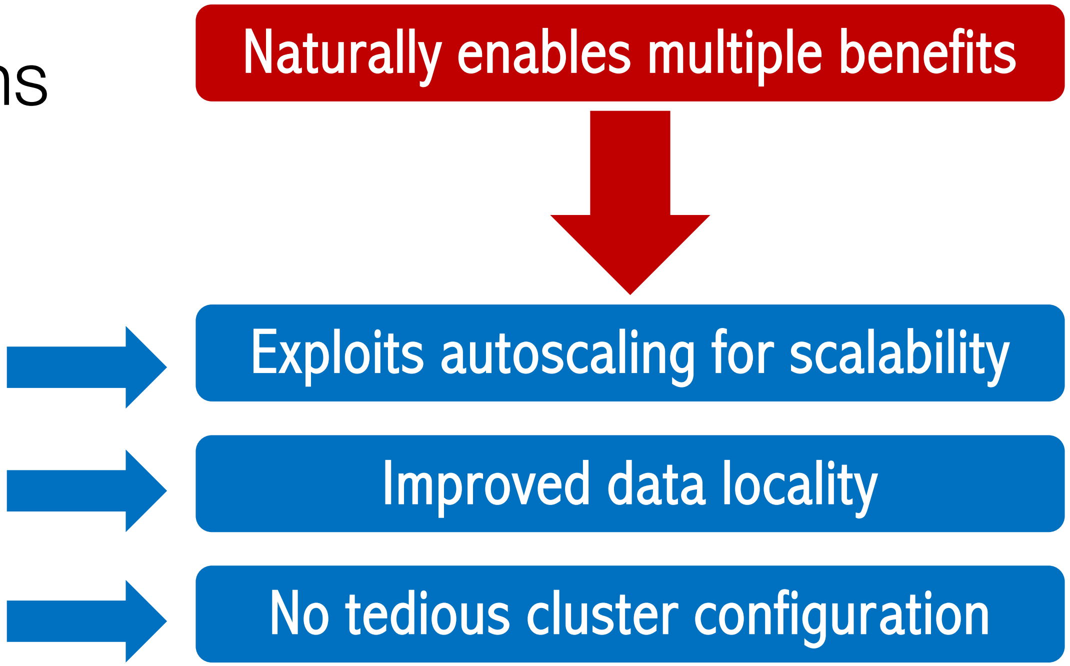
WUKONG 悟空
SERVERLESS DAG ENGINE

https://github.com/ds2-lab/Wukong

Wukong is a **FaaS-centric** parallel computing framework

Key idea: Partitions the work of a centralized scheduler across many functions to take advantage of FaaS elasticity

- Functions schedule tasks by invoking functions
- Functions execute multiple tasks to reduce data movement cost
- Functions scale out / in autonomously

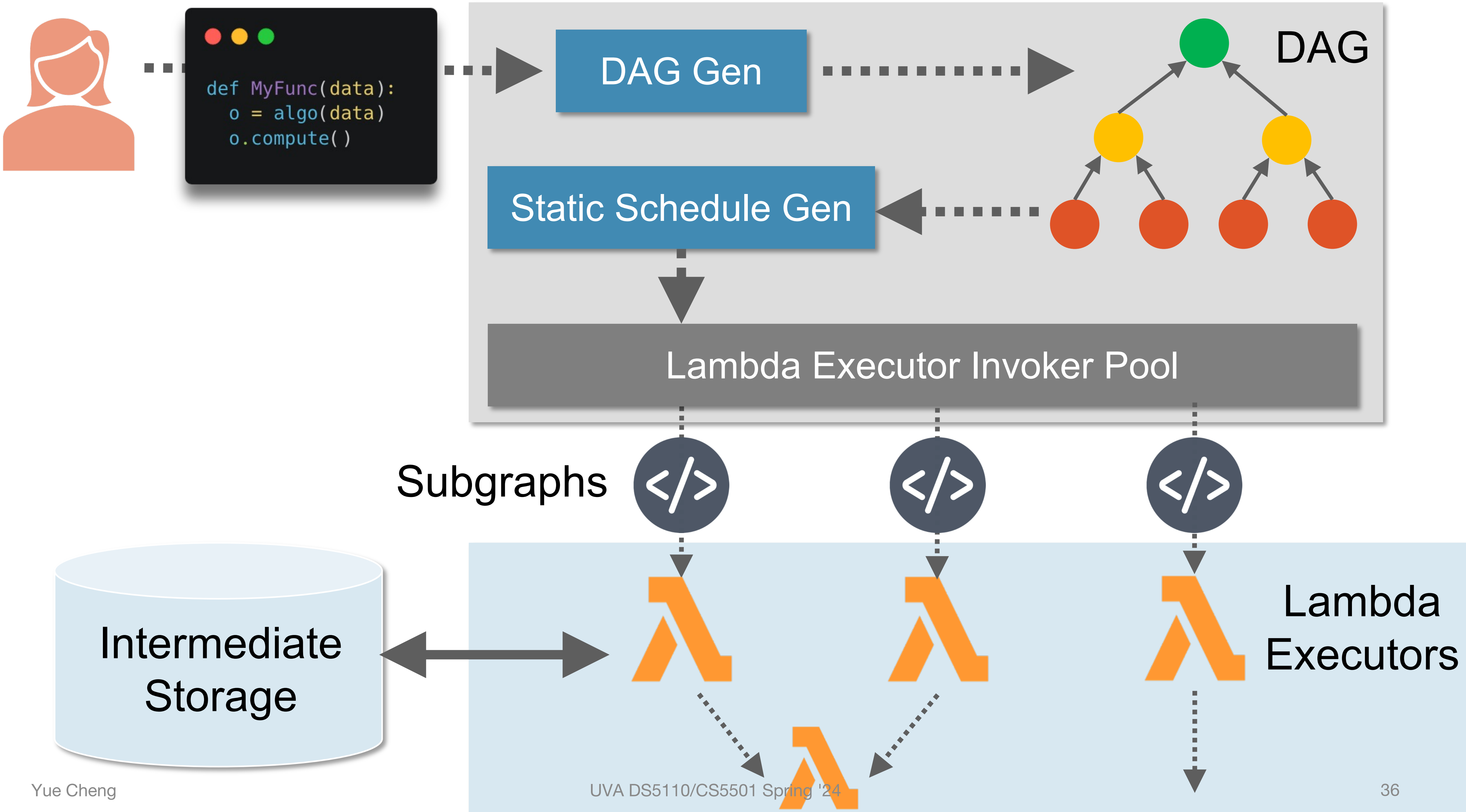


Naturally enables multiple benefits

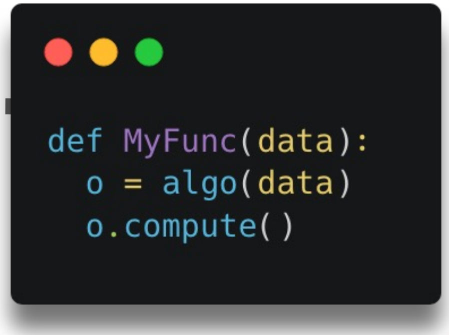
Exploits autoscaling for scalability

Improved data locality

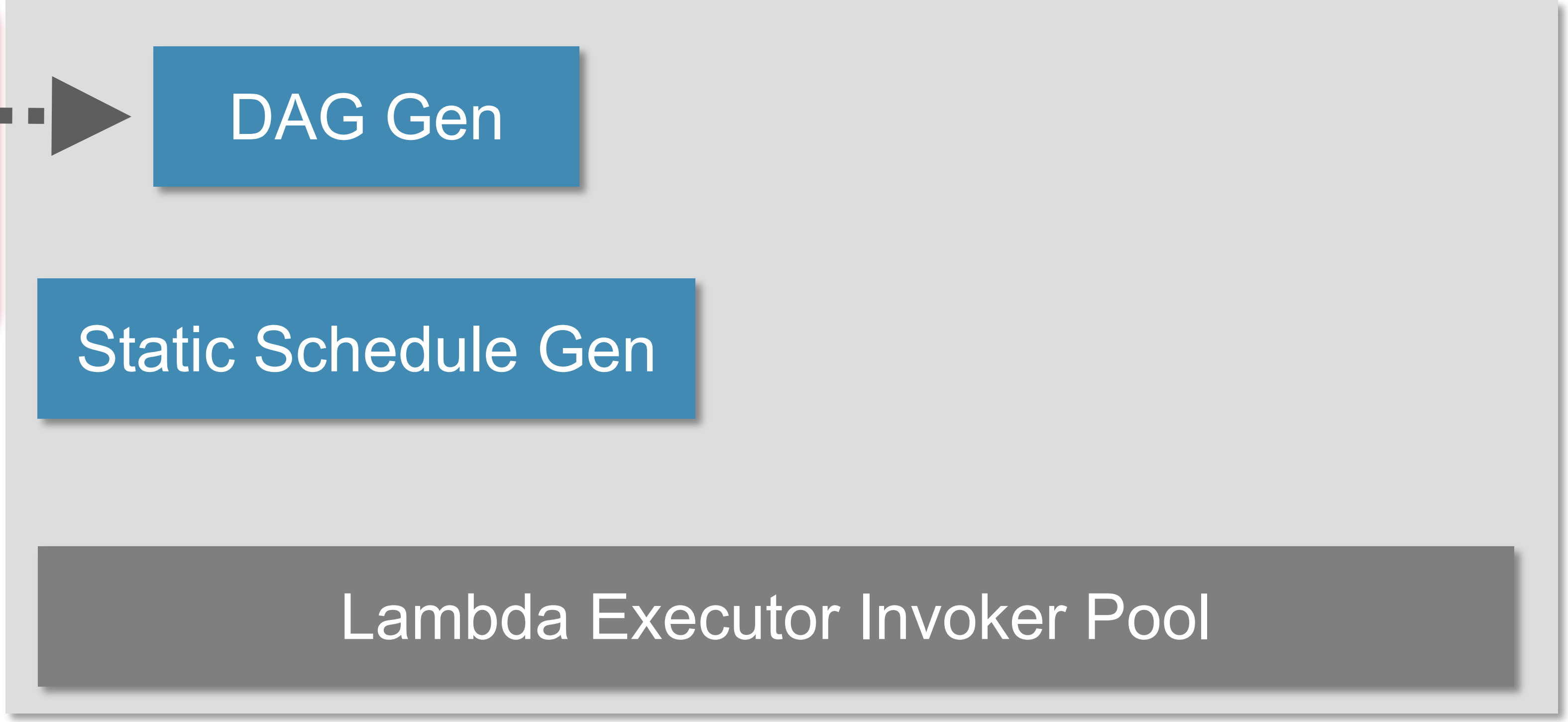
No tedious cluster configuration

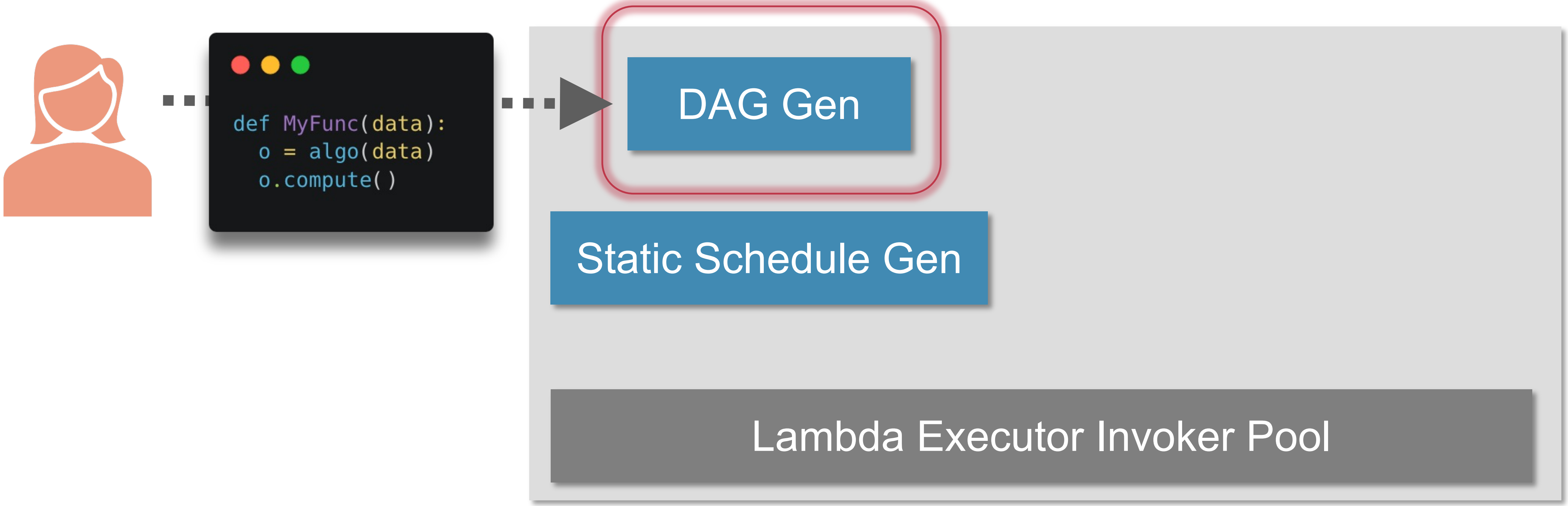


```
def MyFunc(data):
  o = algo(data)
  o.compute()
```

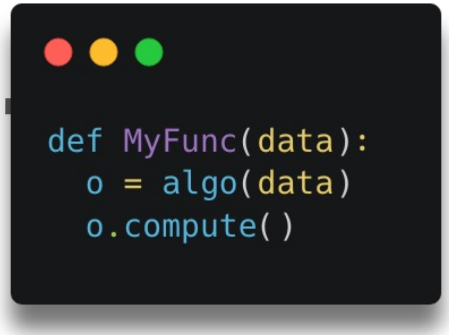


```
def MyFunc(data):
  o = algo(data)
  o.compute()
```

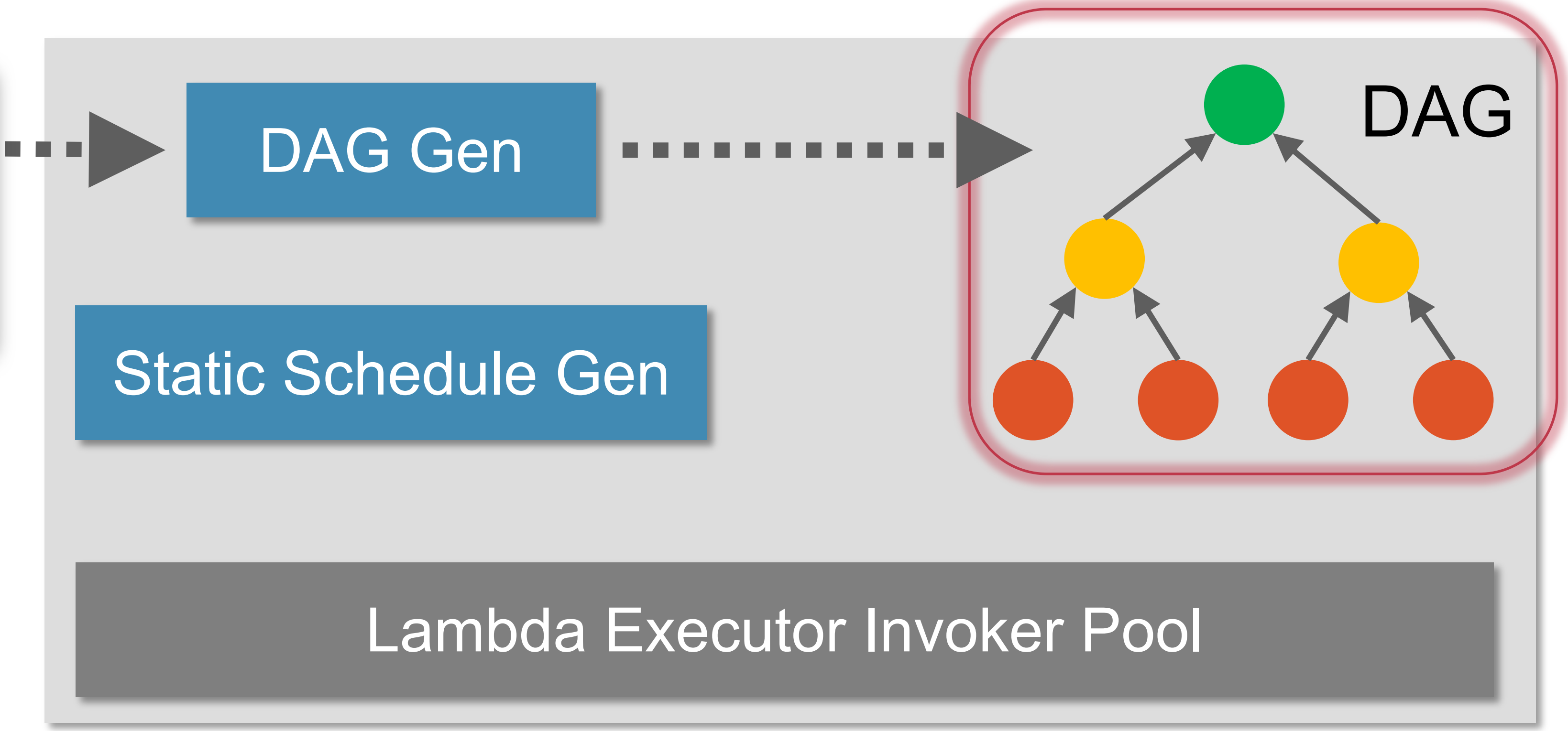


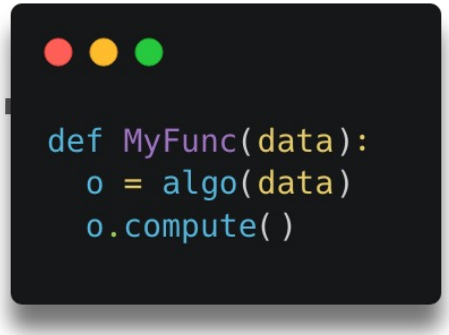


```
def MyFunc(data):
  o = algo(data)
  o.compute()
```

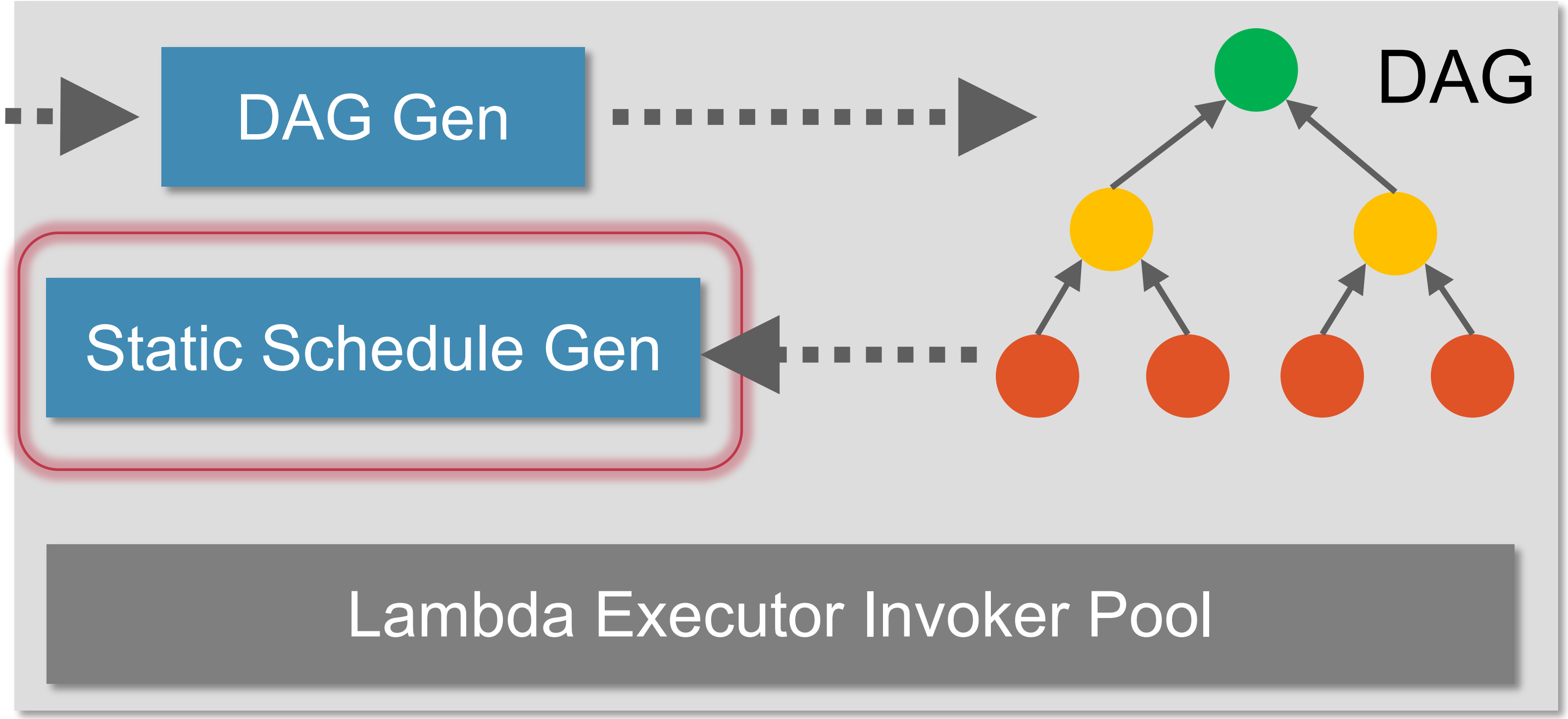


```
def MyFunc(data):
  o = algo(data)
  o.compute()
```





```
def MyFunc(data):
  o = algo(data)
  o.compute()
```

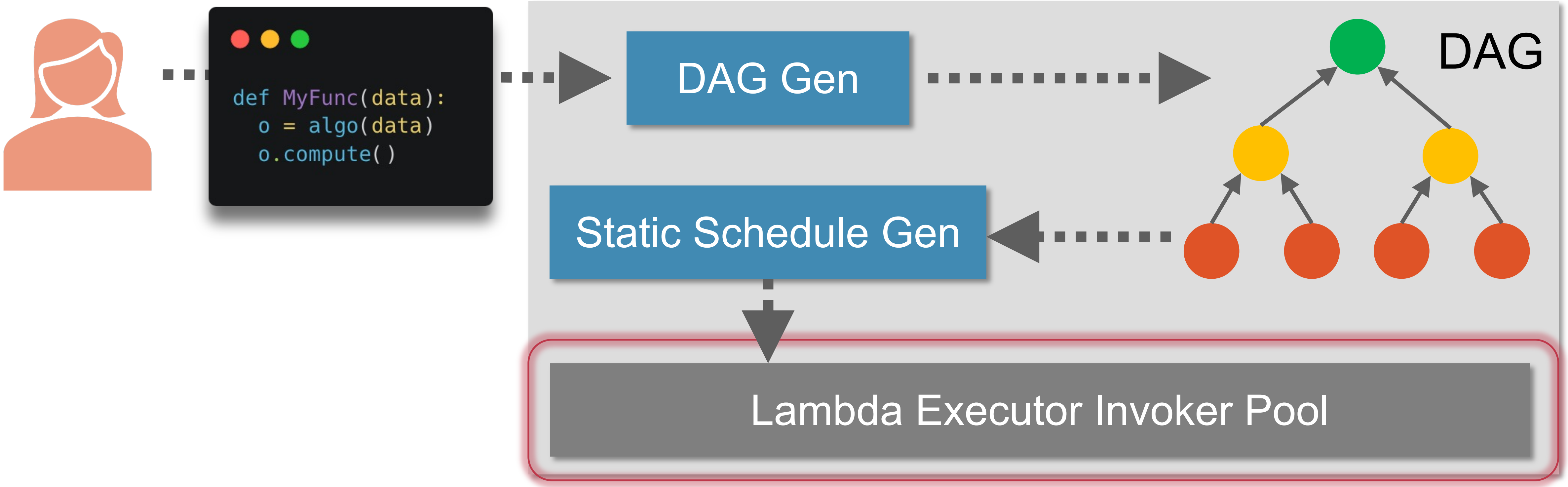


DAG Gen

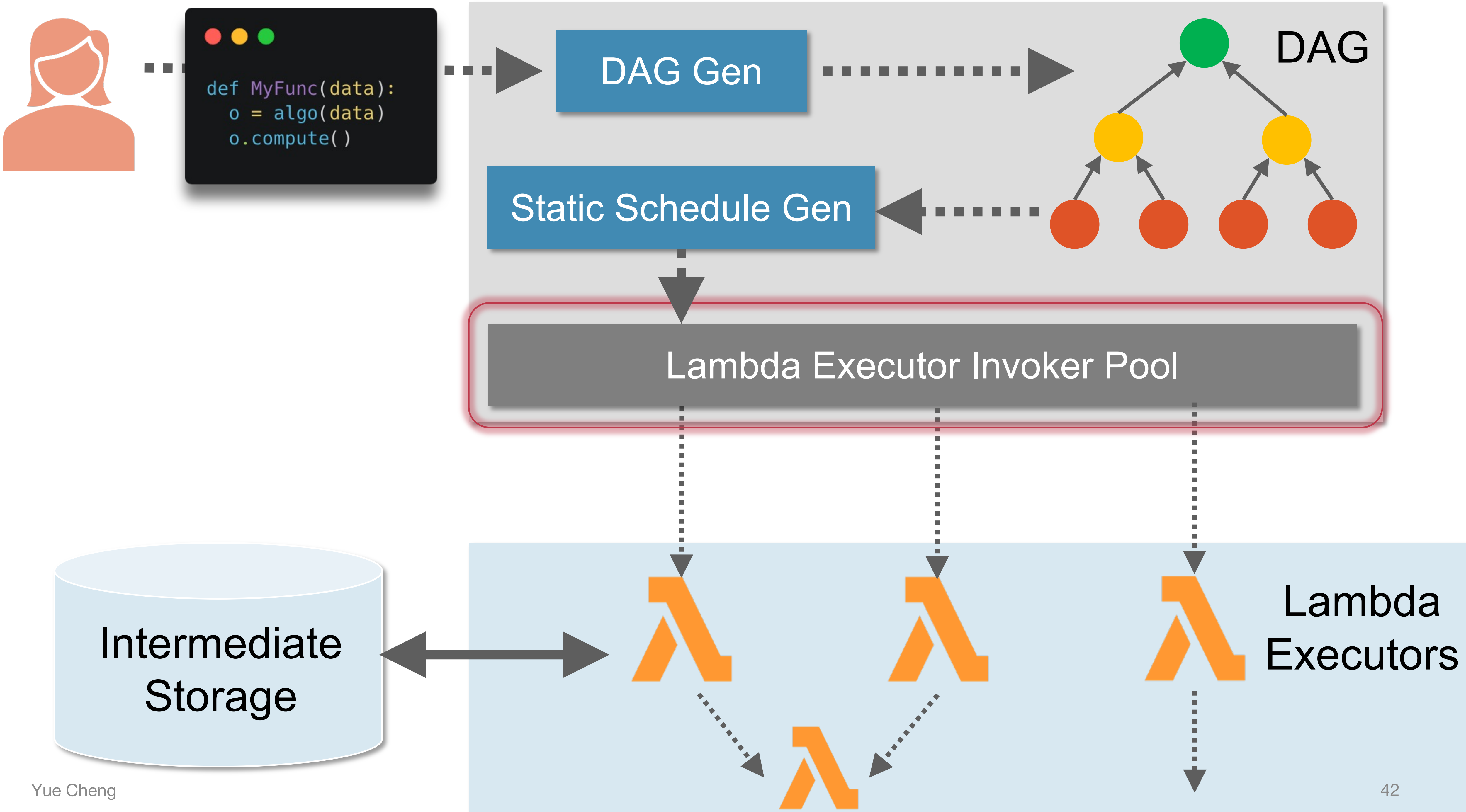
DAG

Static Schedule Gen

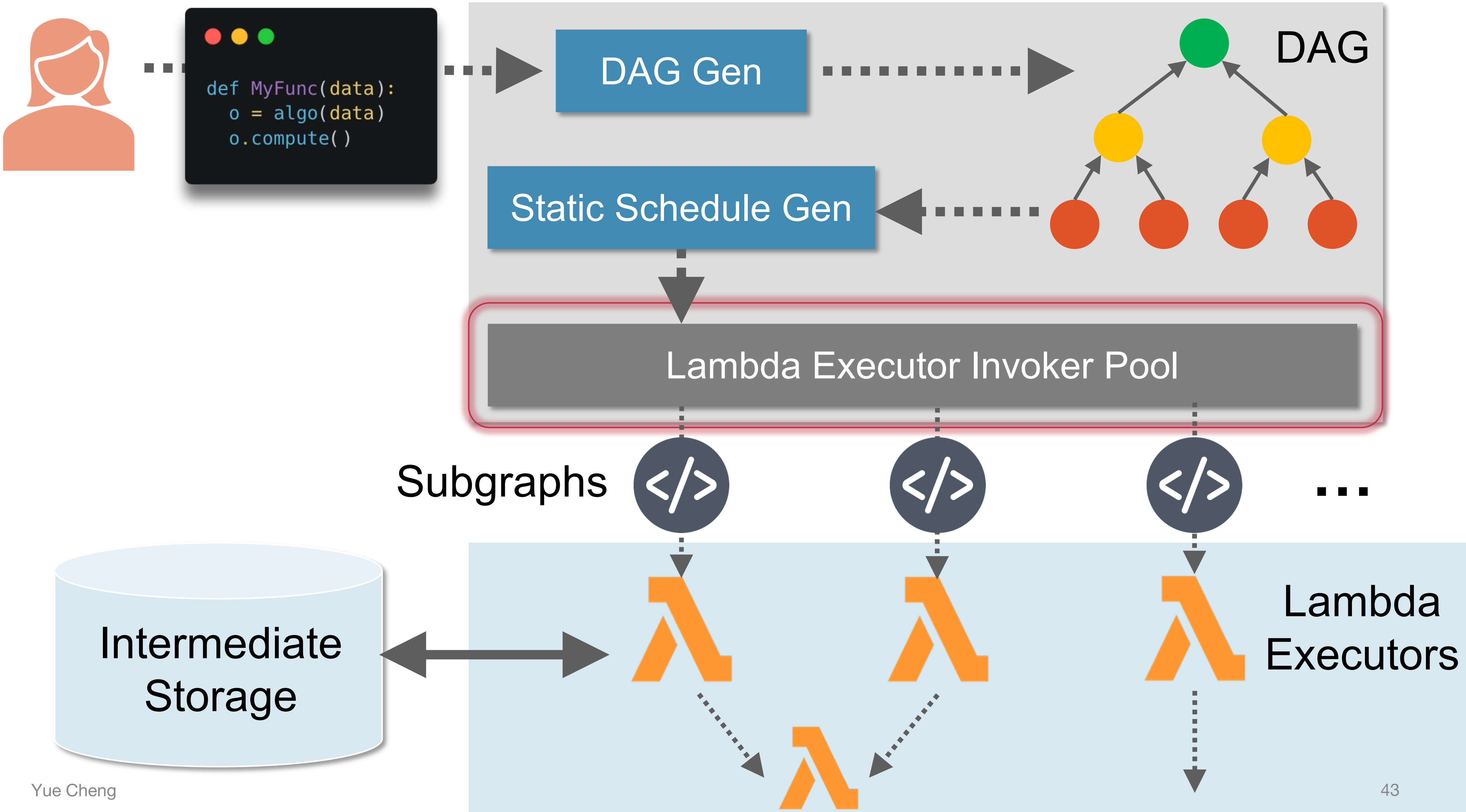
Lambda Executor Invoker Pool



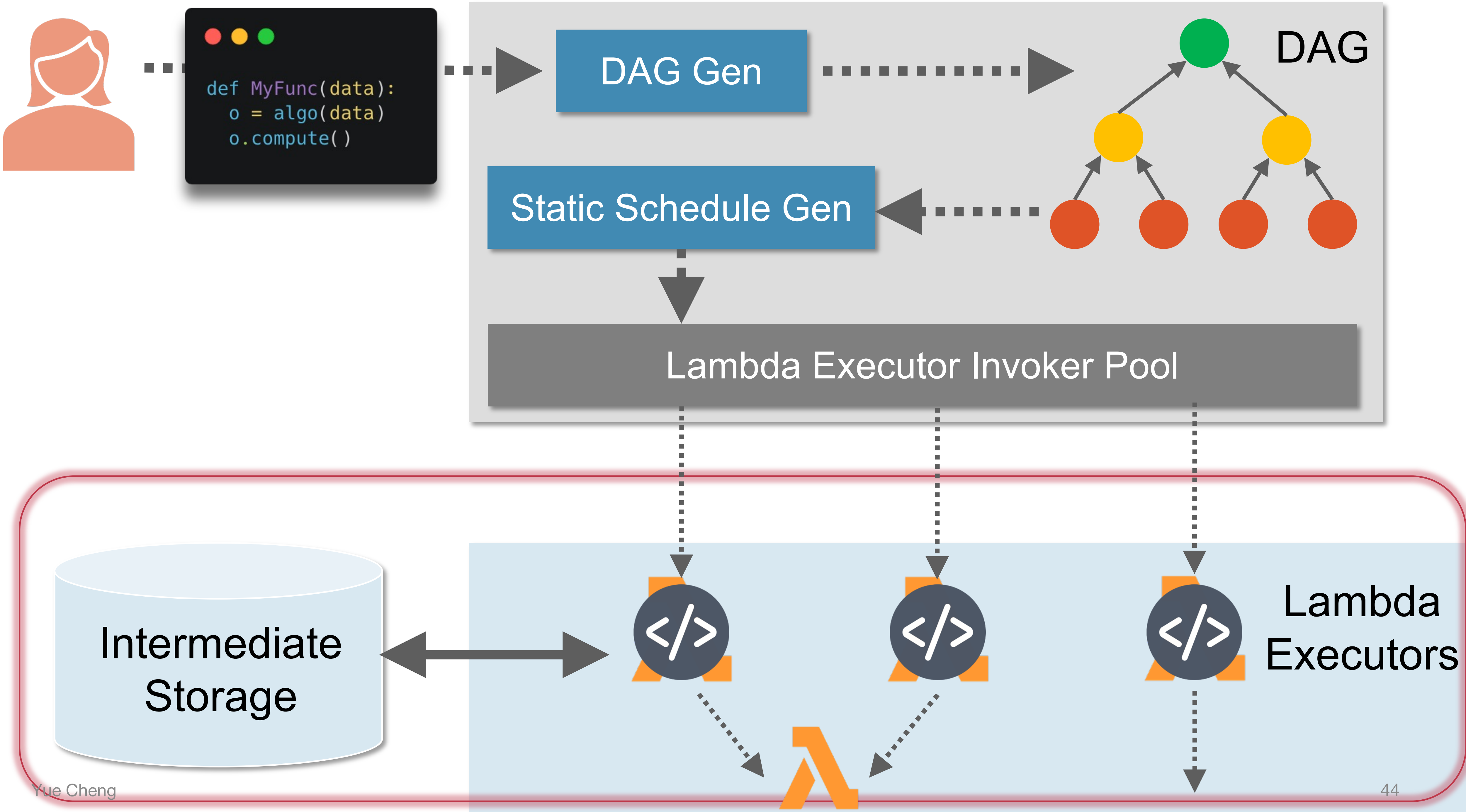
```
def MyFunc(data):
  o = algo(data)
  o.compute()
```



```
def MyFunc(data):
  o = algo(data)
  o.compute()
```



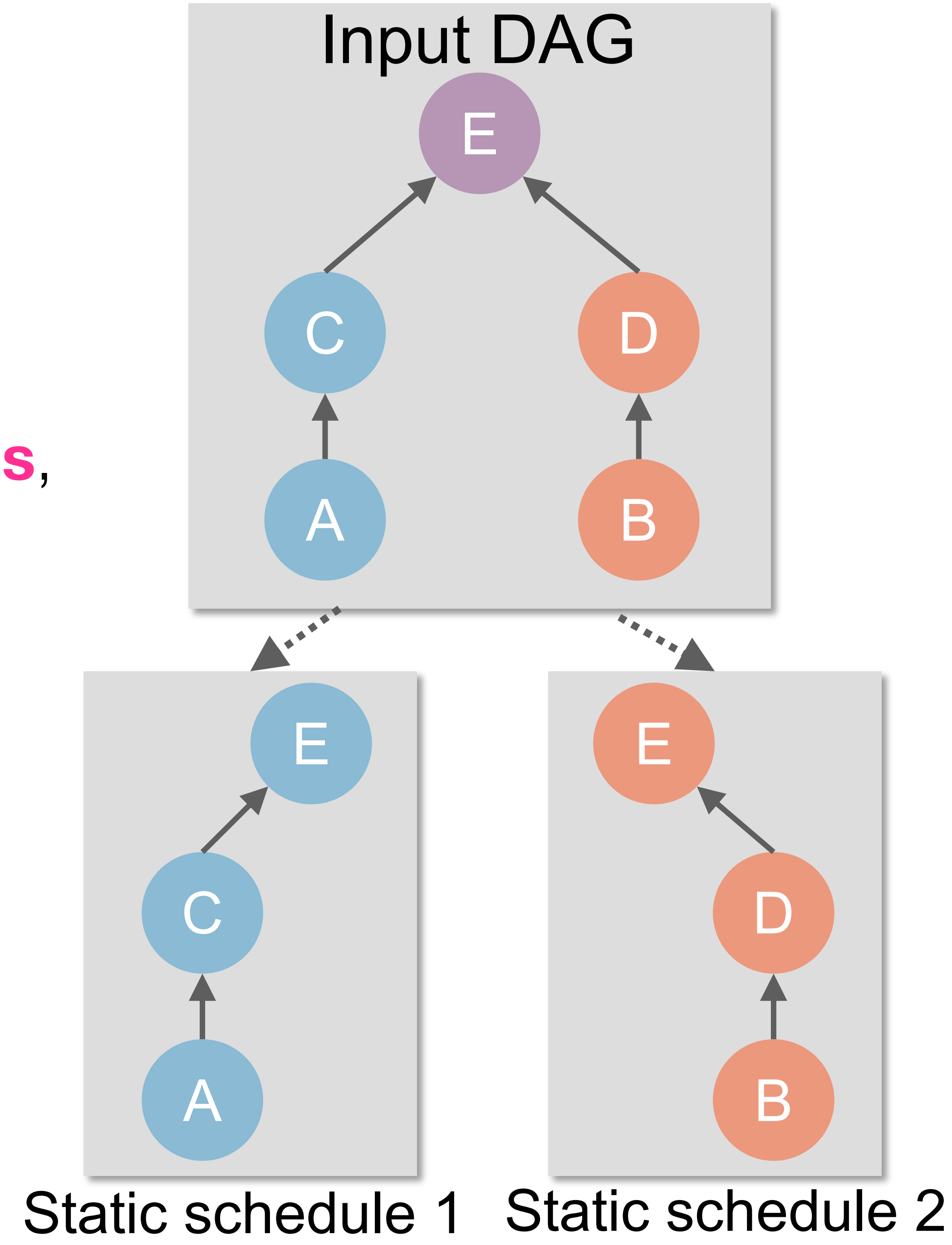
```
def MyFunc(data):
  o = algo(data)
  o.compute()
```

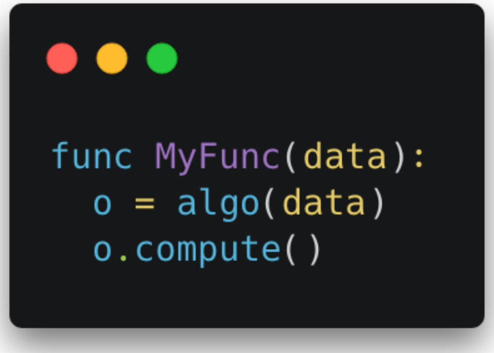


```
def MyFunc(data):
  o = algo(data)
  o.compute()
```

Scheduling in Wukong

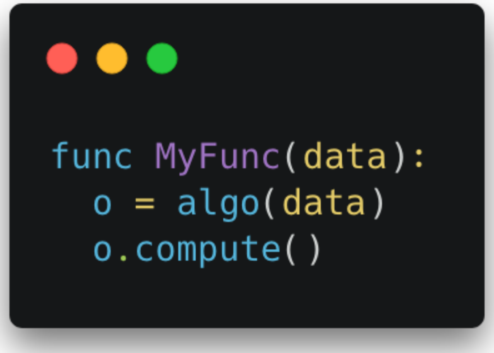
- Combination of **static** and **dynamic** scheduling
- Input DAG partitioned into **static schedules**, or subgraphs of the original DAG
- Serverless executors are assigned a **static schedule**
- Executors use **dynamic scheduling** to enforce data dependencies and **cooperatively** schedule tasks found in multiple static schedules





```
func MyFunc(data):
  o = algo(data)
  o.compute()
```

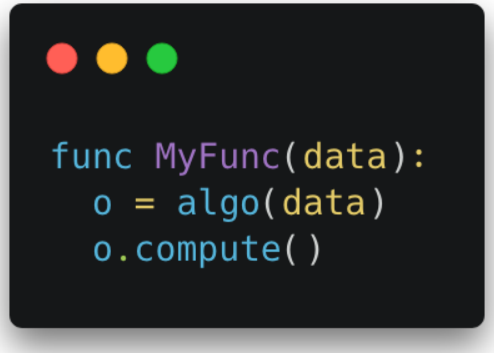
Static scheduling



```
func MyFunc(data):
  o = algo(data)
  o.compute()
```

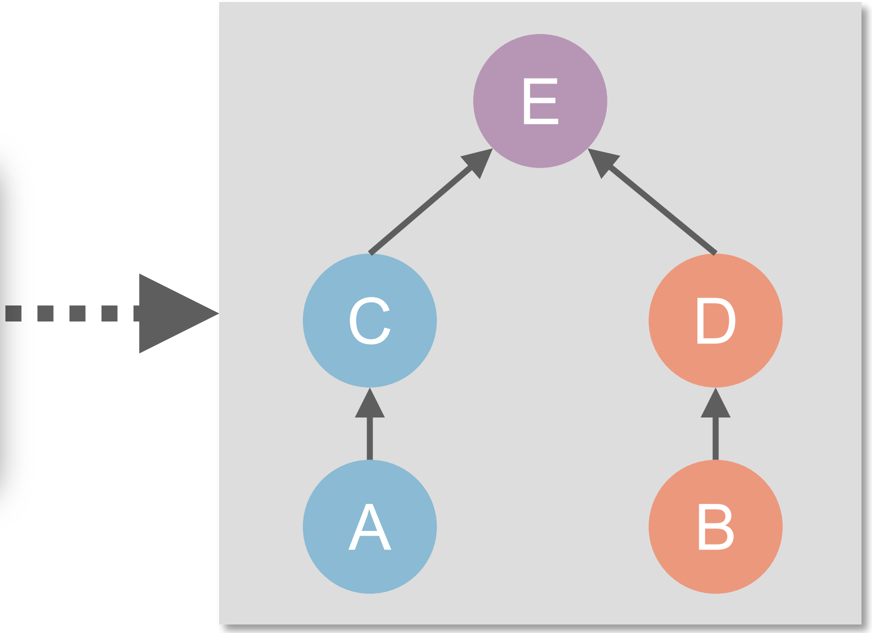
Static scheduling

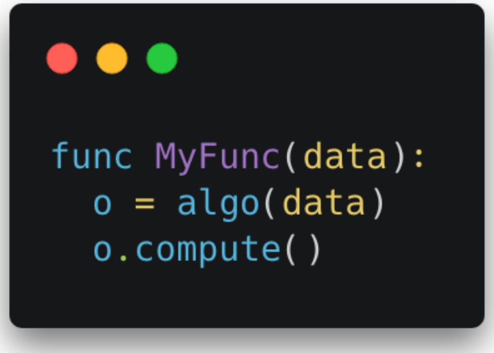
Static scheduling



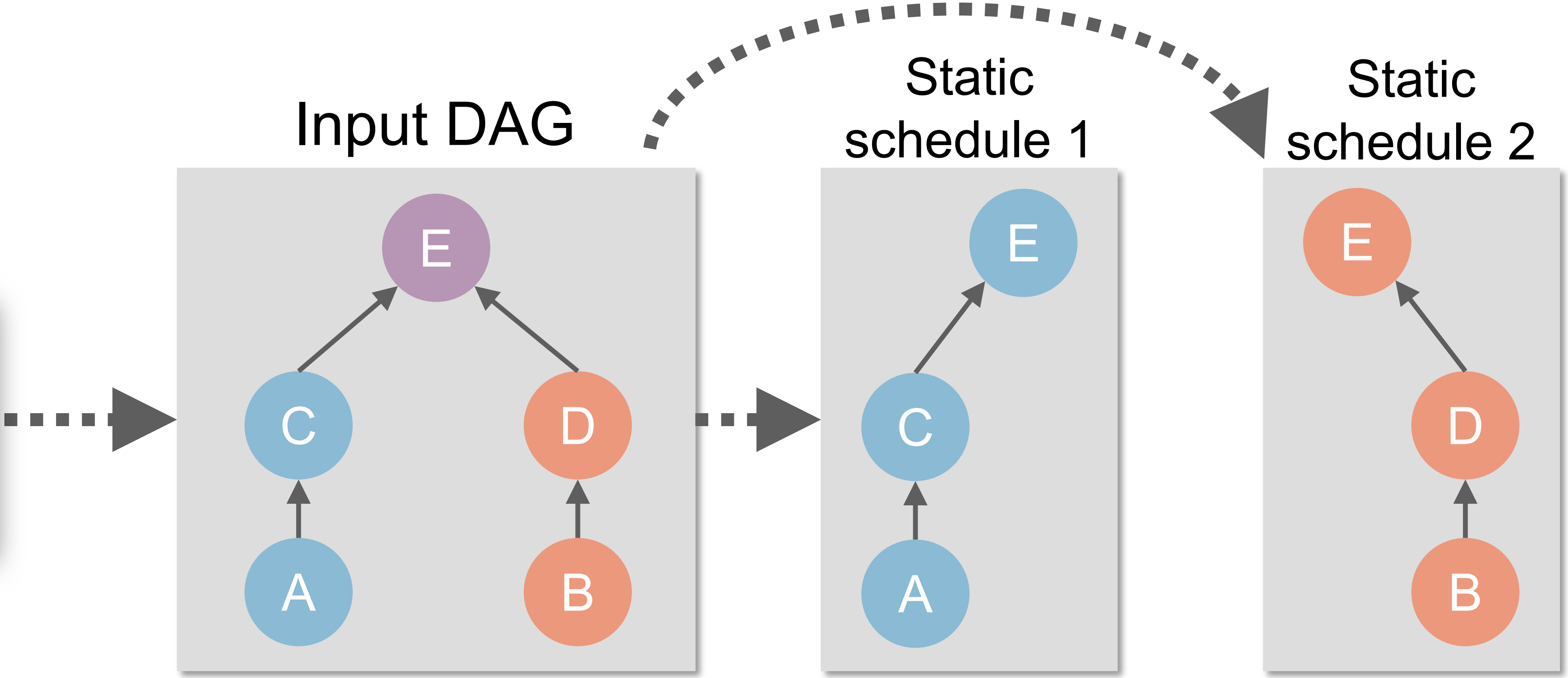
```
func MyFunc(data):
  o = algo(data)
  o.compute()
```

Input DAG

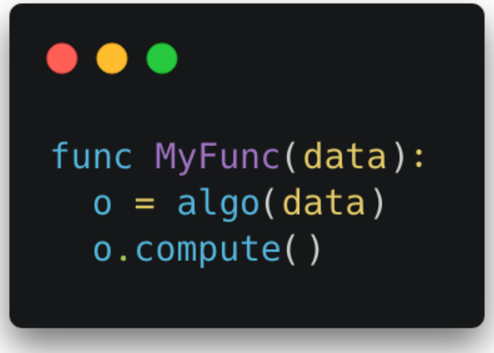




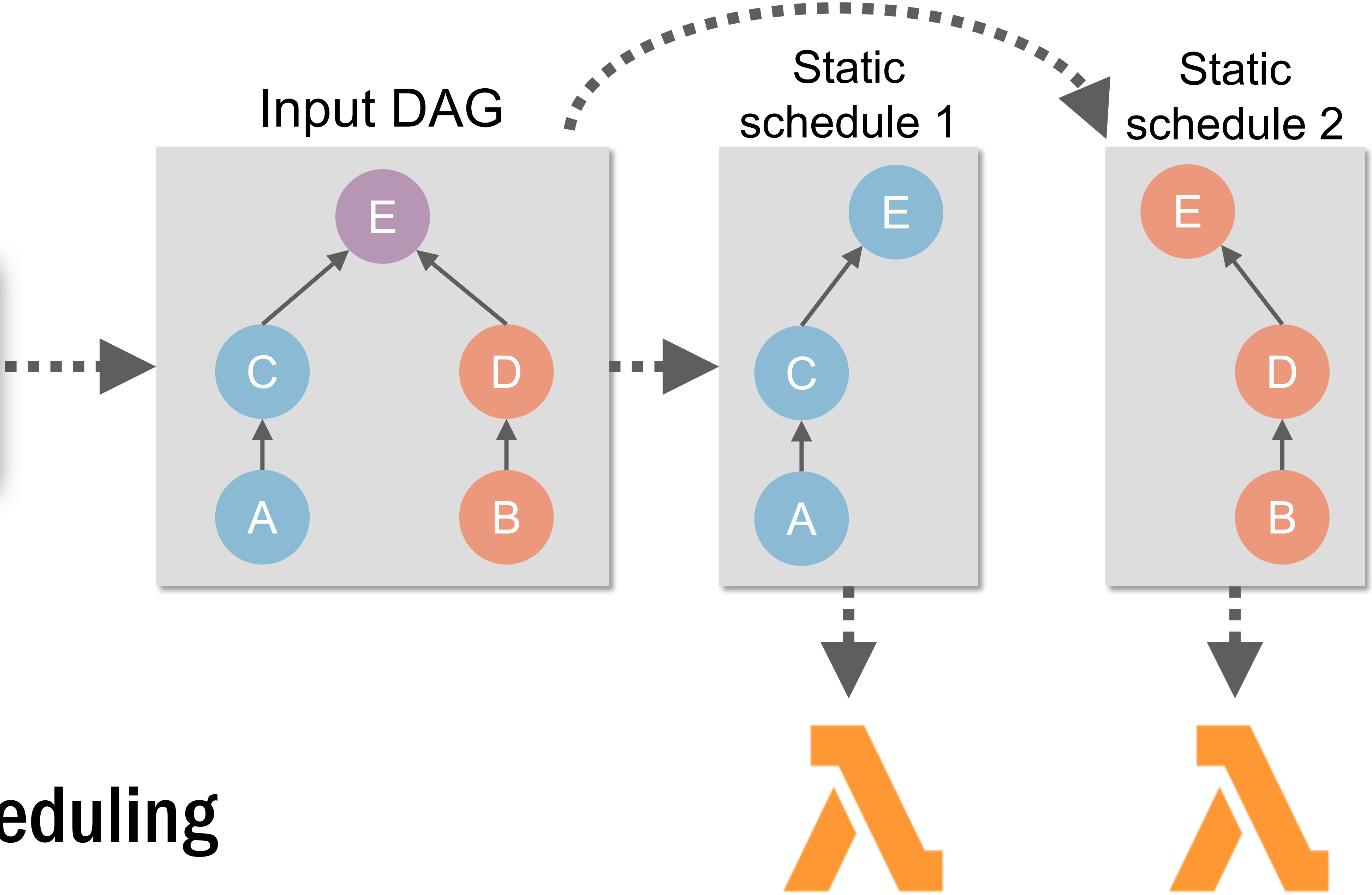
```
func MyFunc(data):
  o = algo(data)
  o.compute()
```



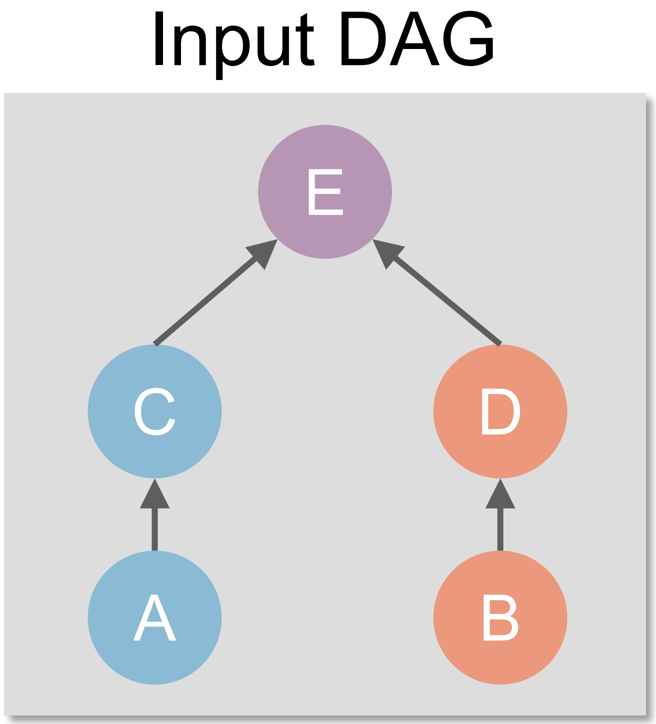
Static scheduling

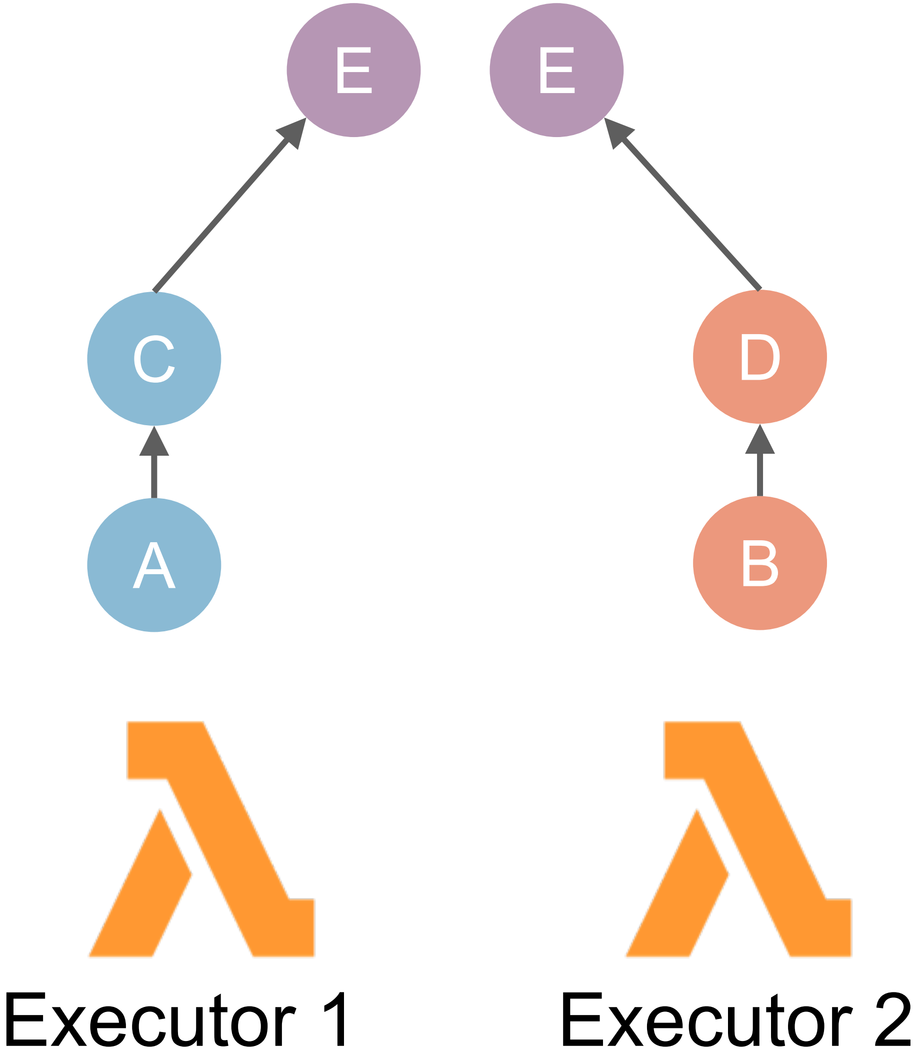


```
func MyFunc(data):
  o = algo(data)
  o.compute()
```



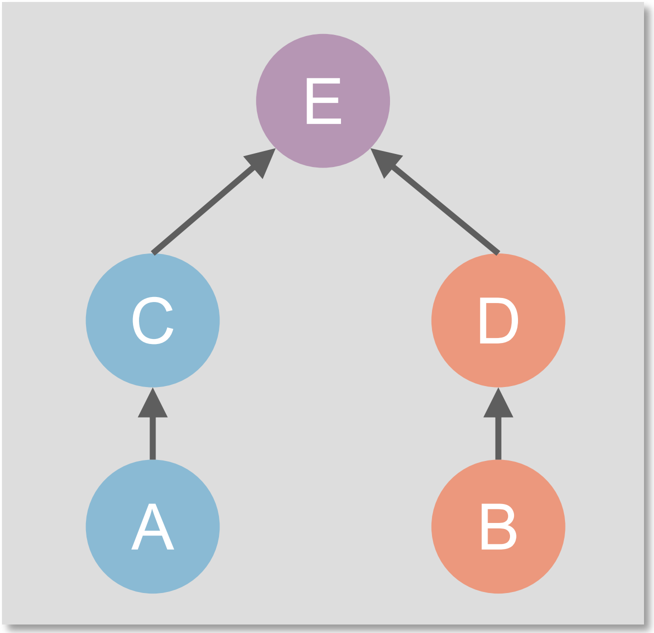
Static scheduling

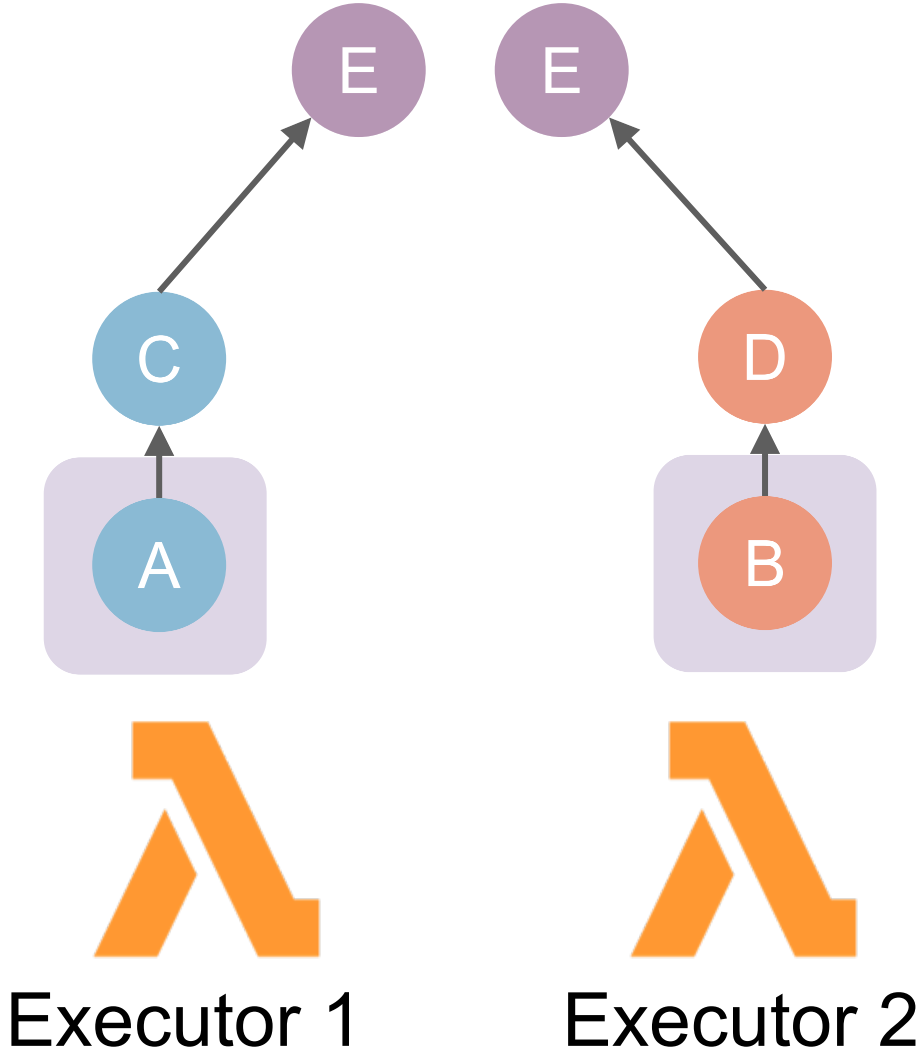




Dynamic scheduling

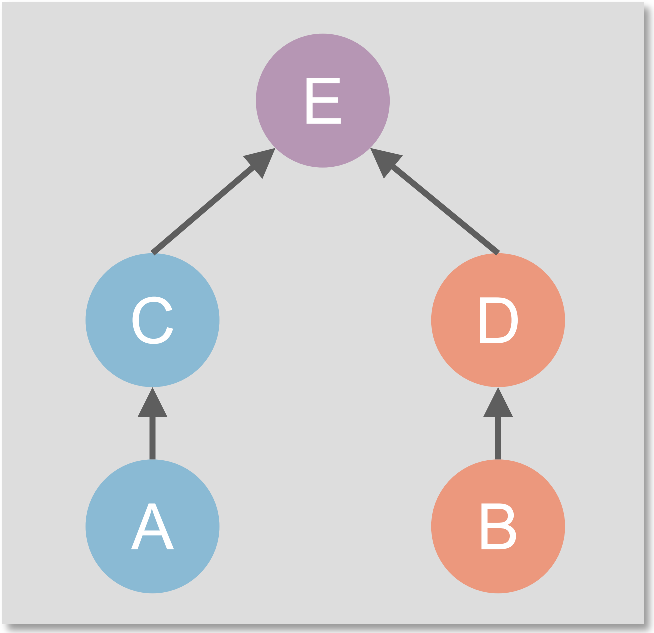
Input DAG

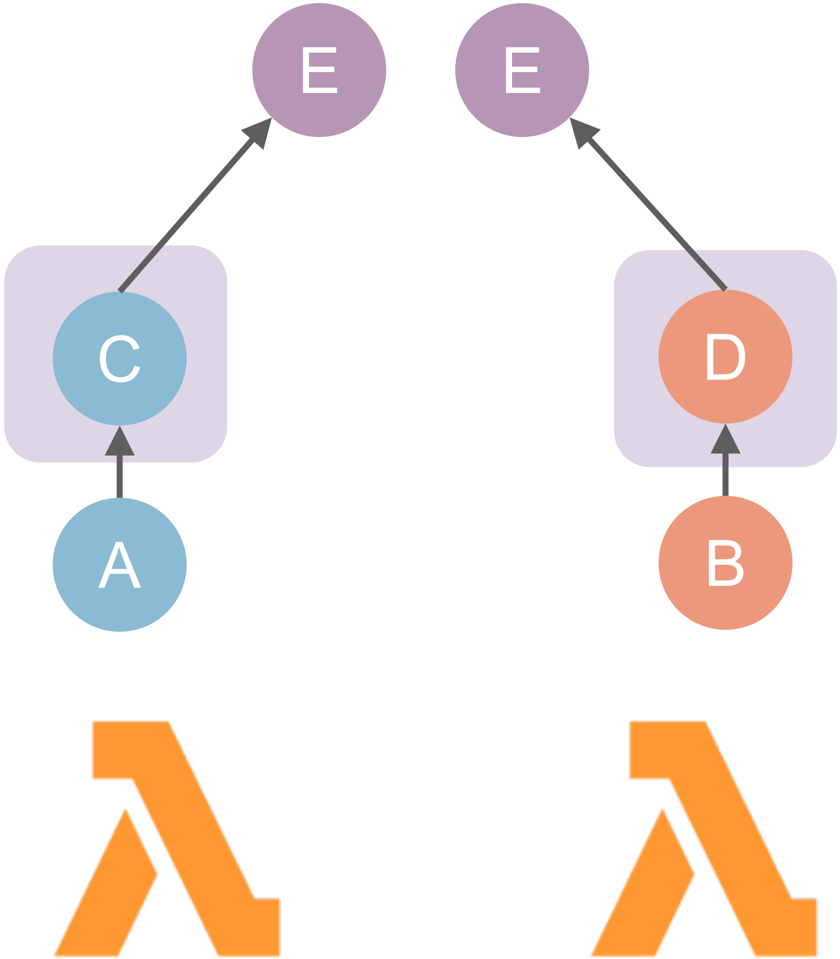




Dynamic scheduling

Input DAG





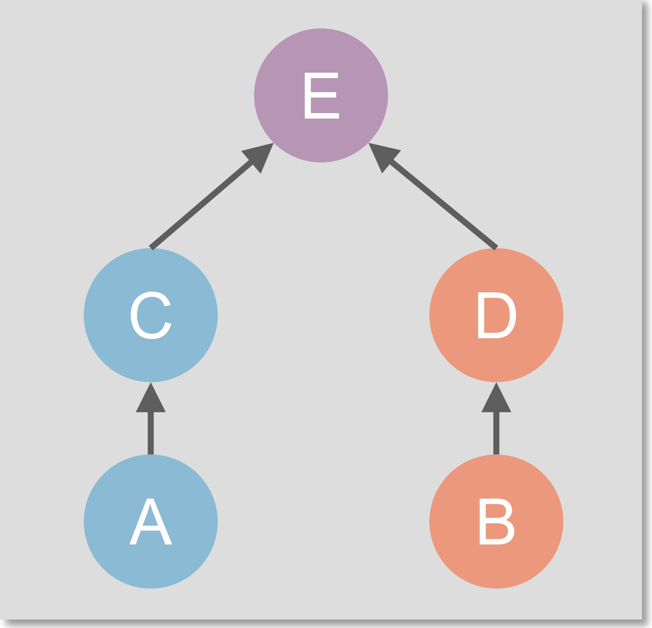
Executor 1

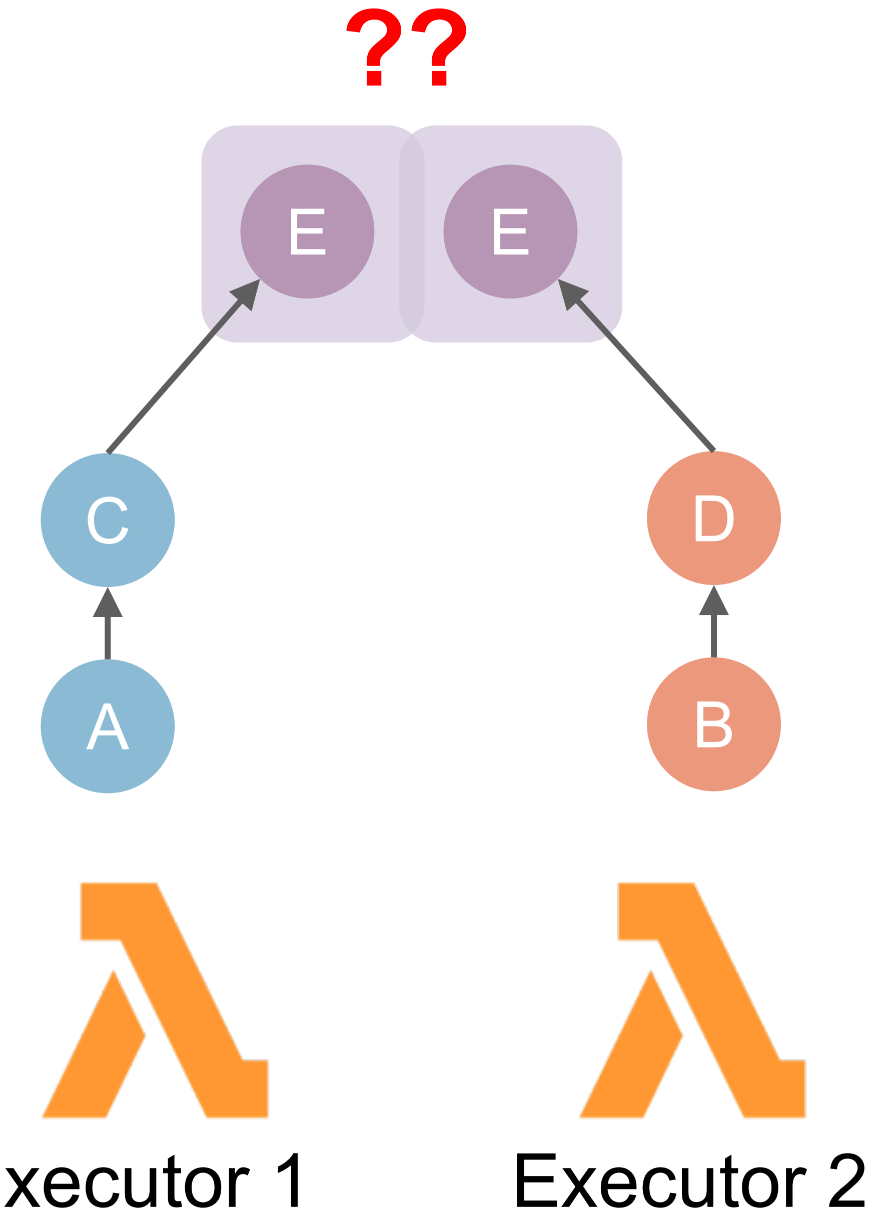
Executor 2

Dynamic scheduling

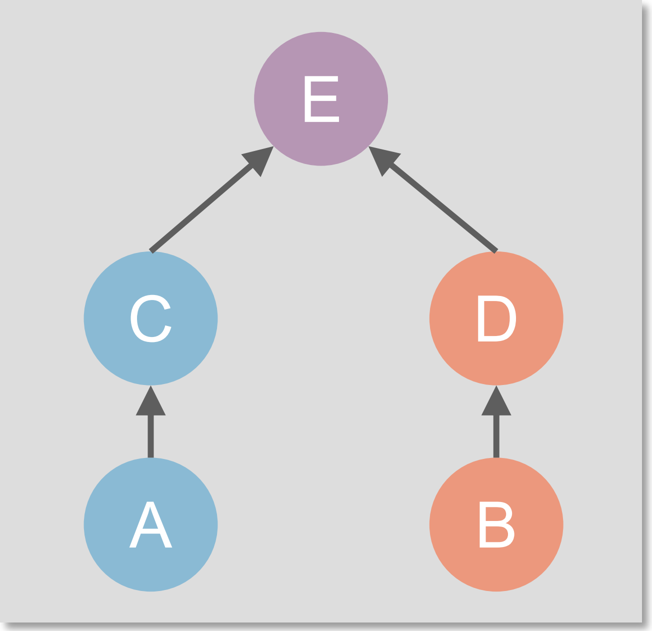
Dynamic scheduling

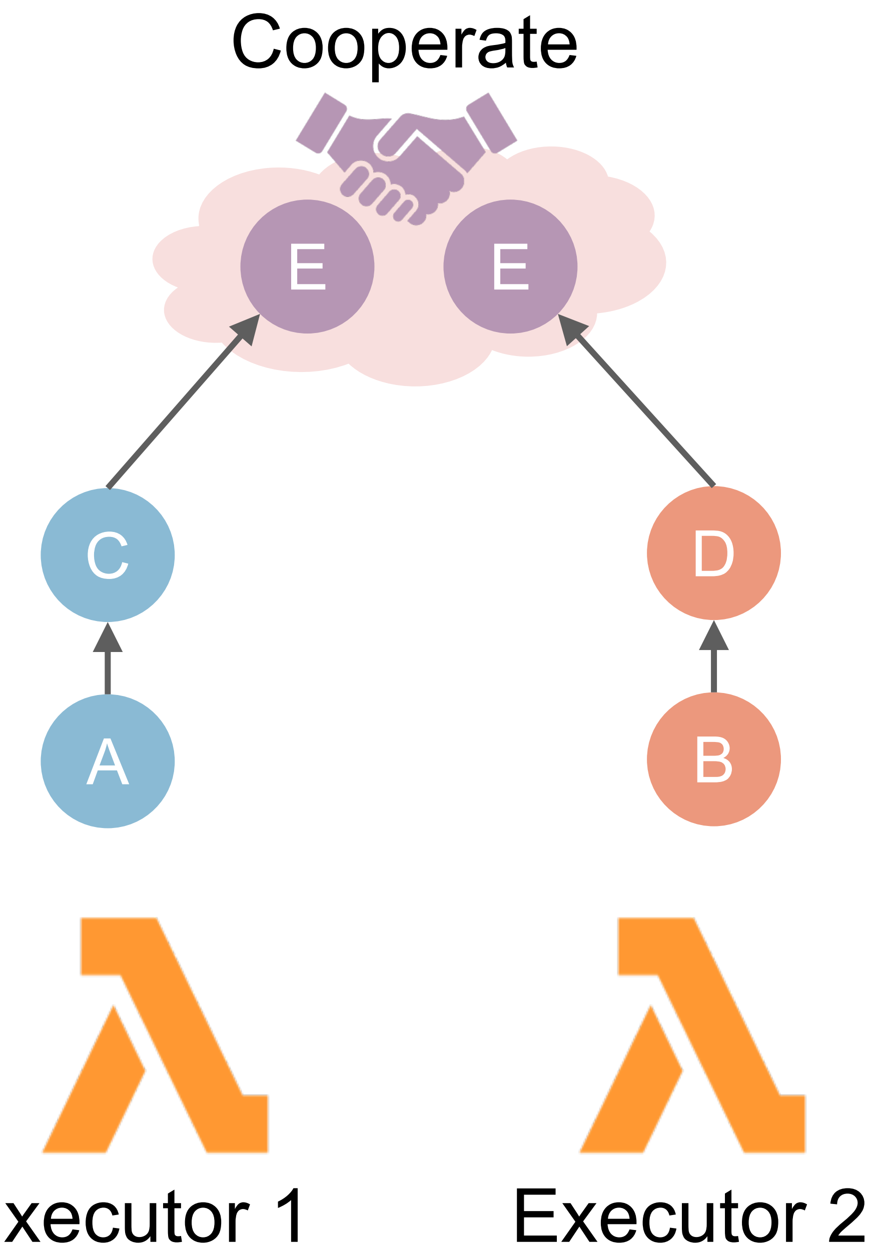
Input DAG





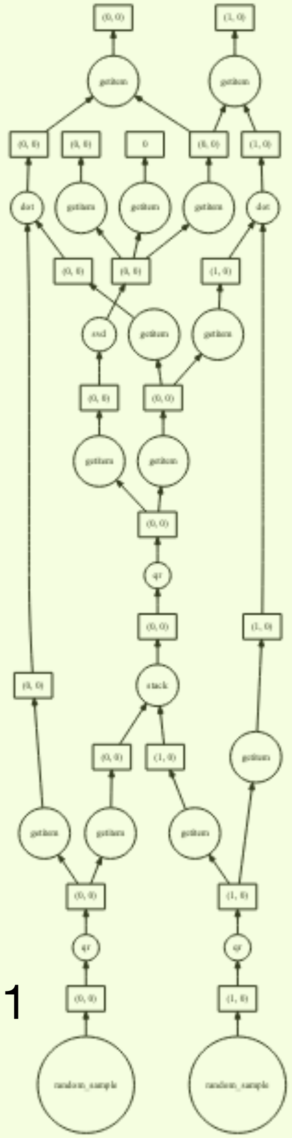
Input DAG



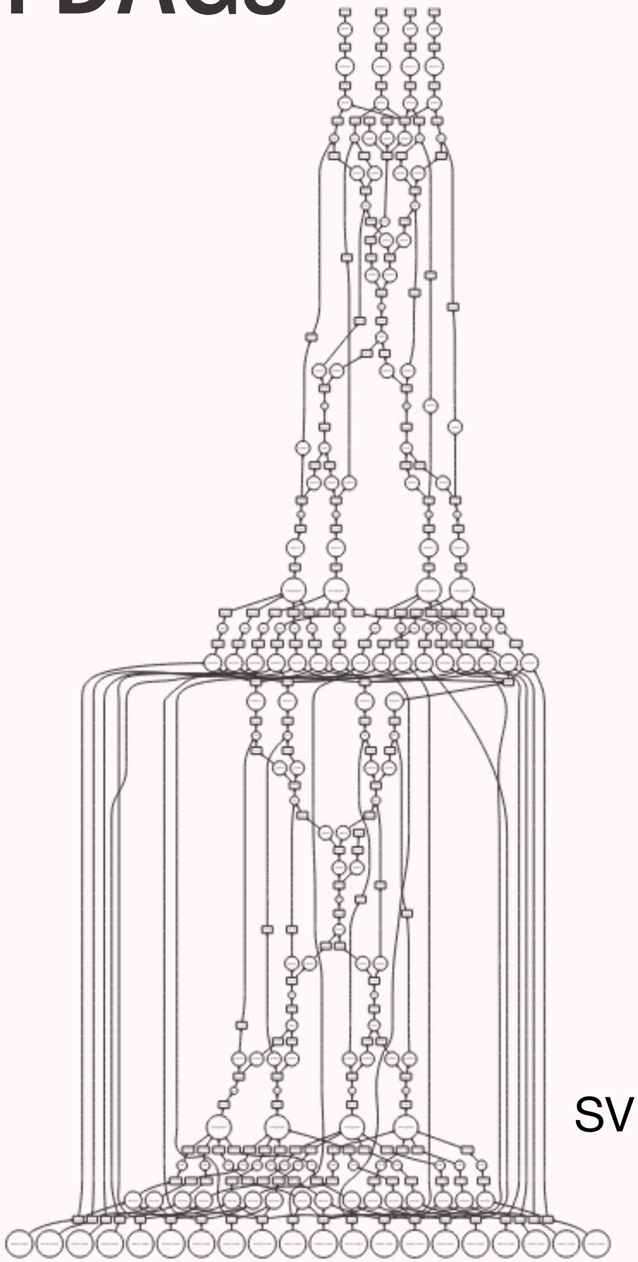


Dynamic scheduling

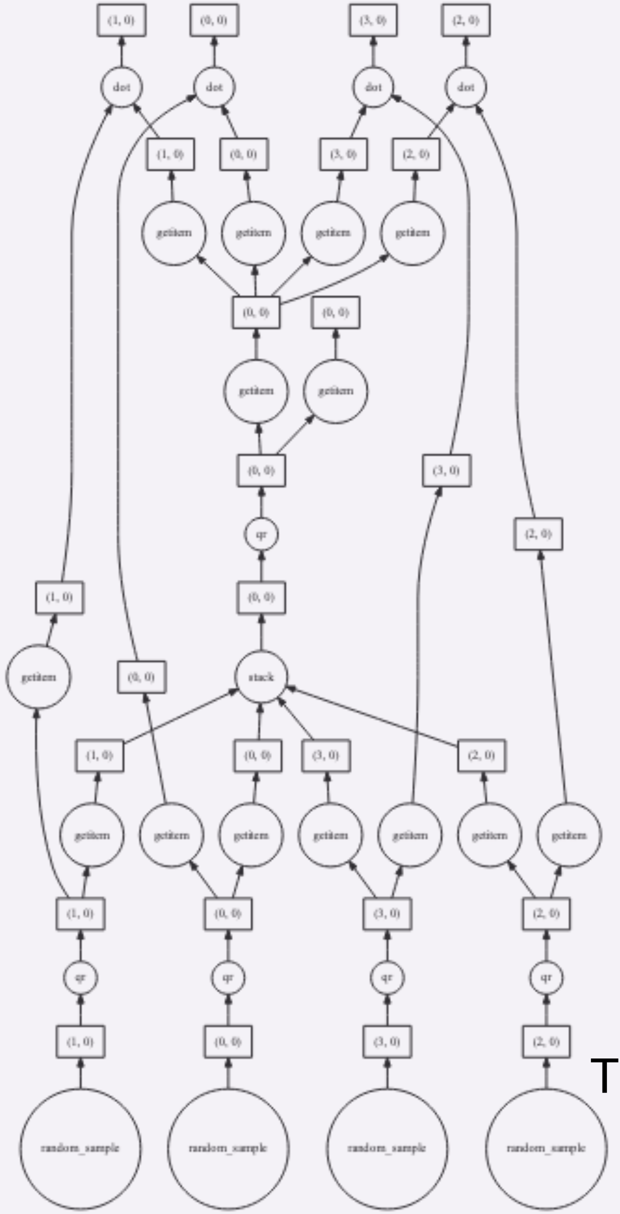
Application DAGs



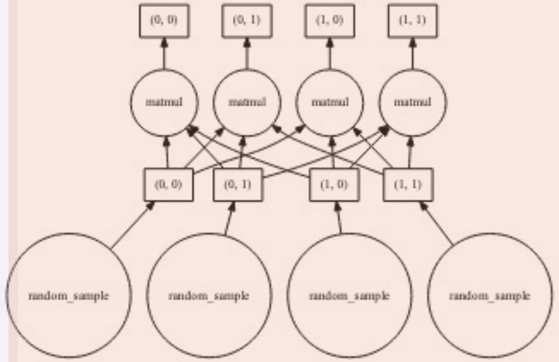
SVD1



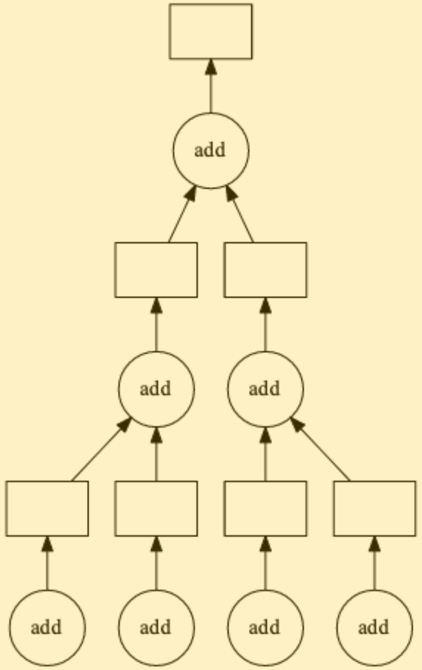
SVD2



TSQR

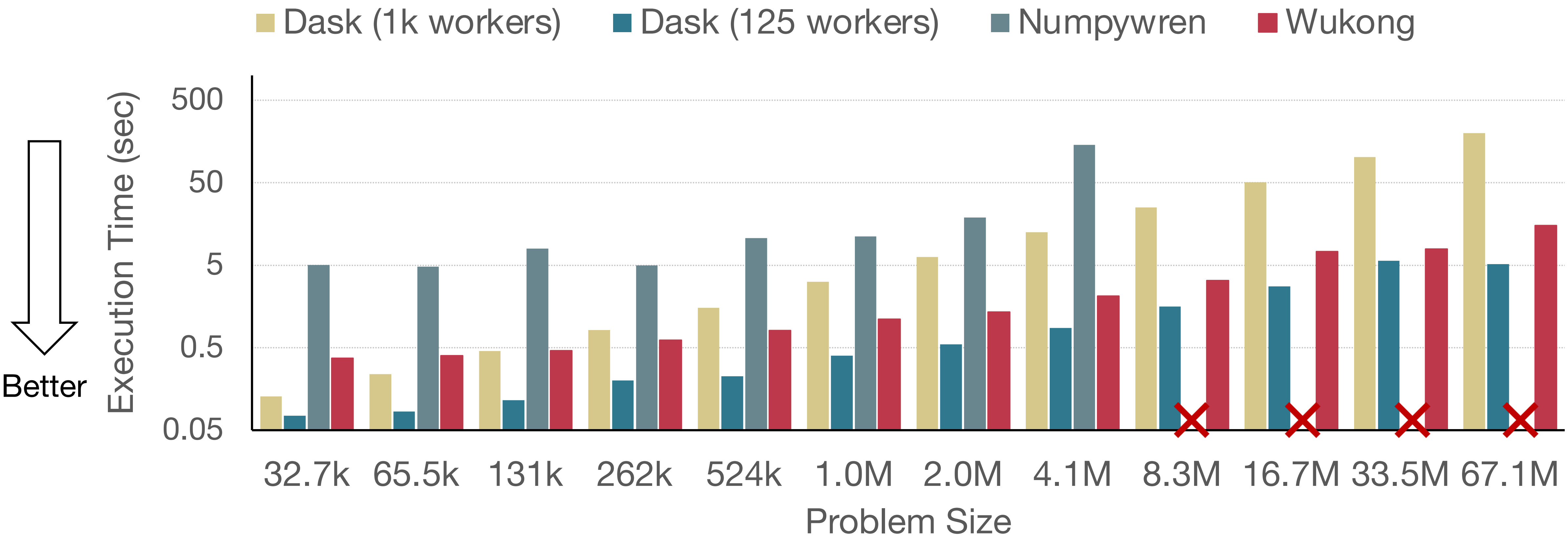


GEMM



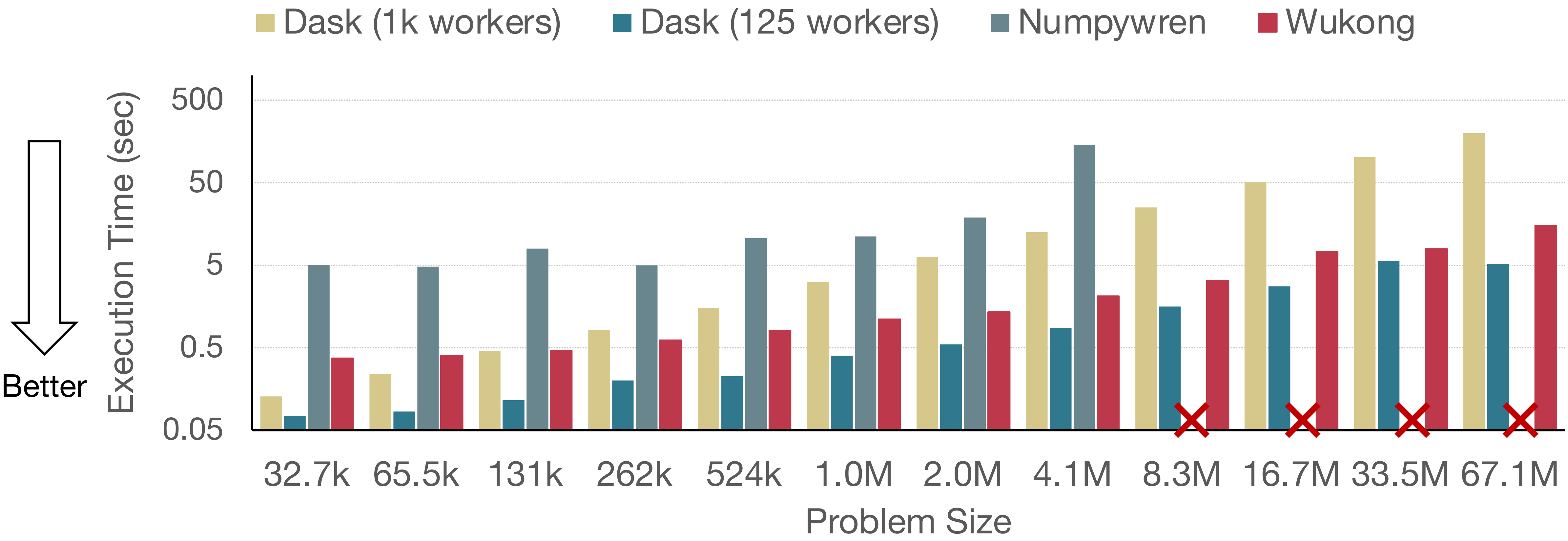
Tree reduction

Application performance: TSQR



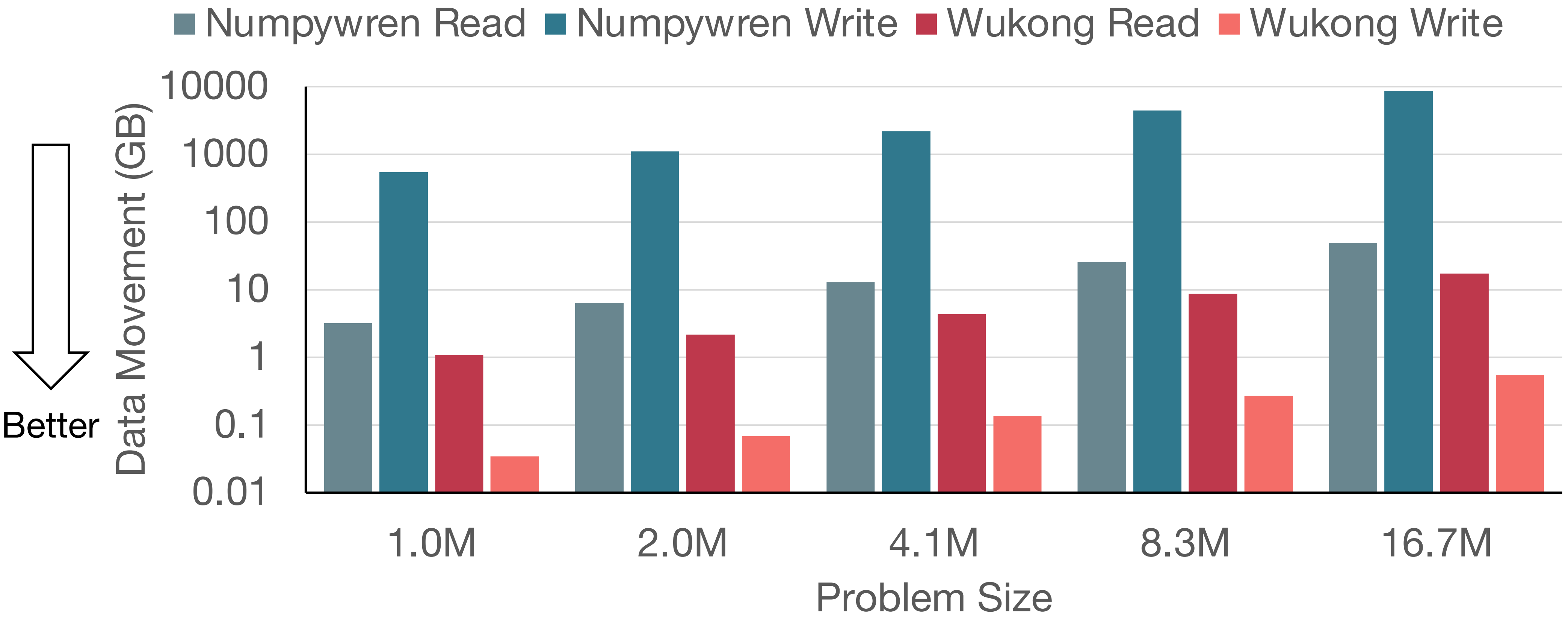
Wukong and numpywren ran on AWS Lambda w/ 3GB memory
Dask distributed ran on 125 c5.4xlarge EC2 VMs w/ 2,000 vCPU cores

Application performance: TSQR

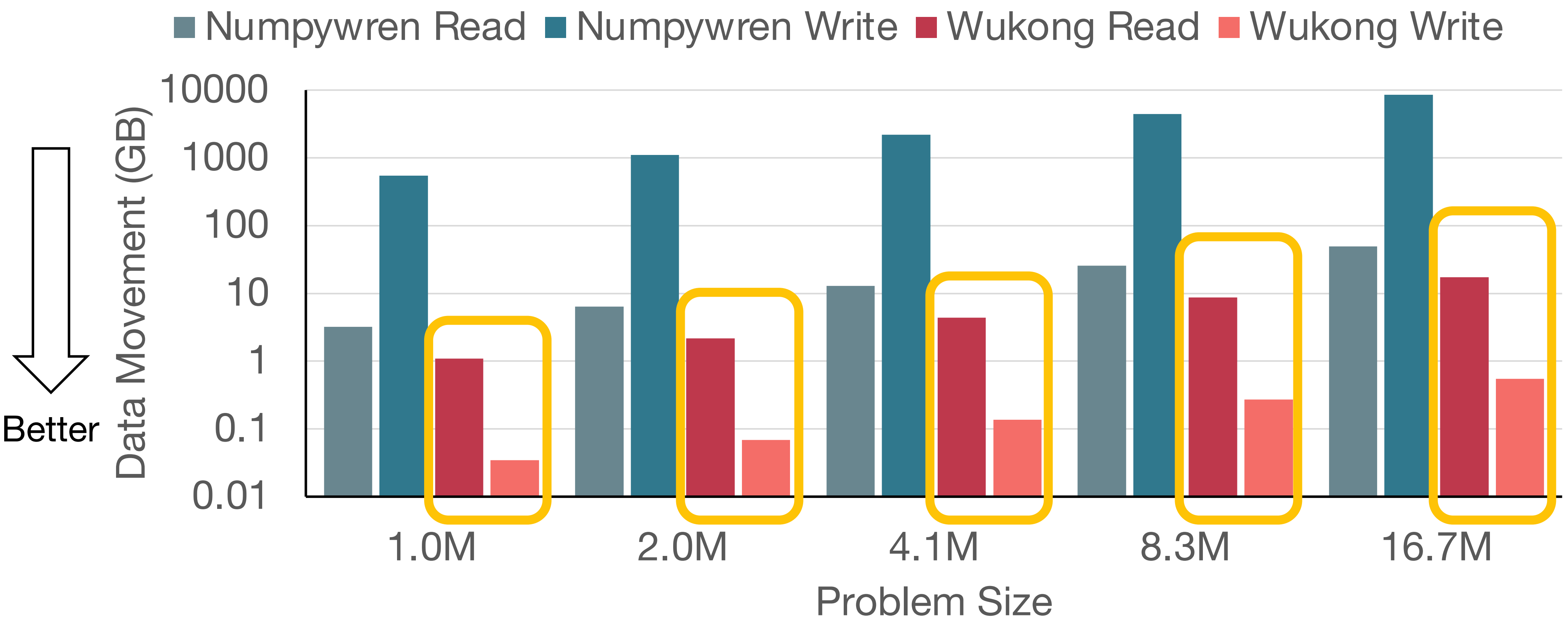


Wukong outperforms numpywren considerably for all problem sizes.

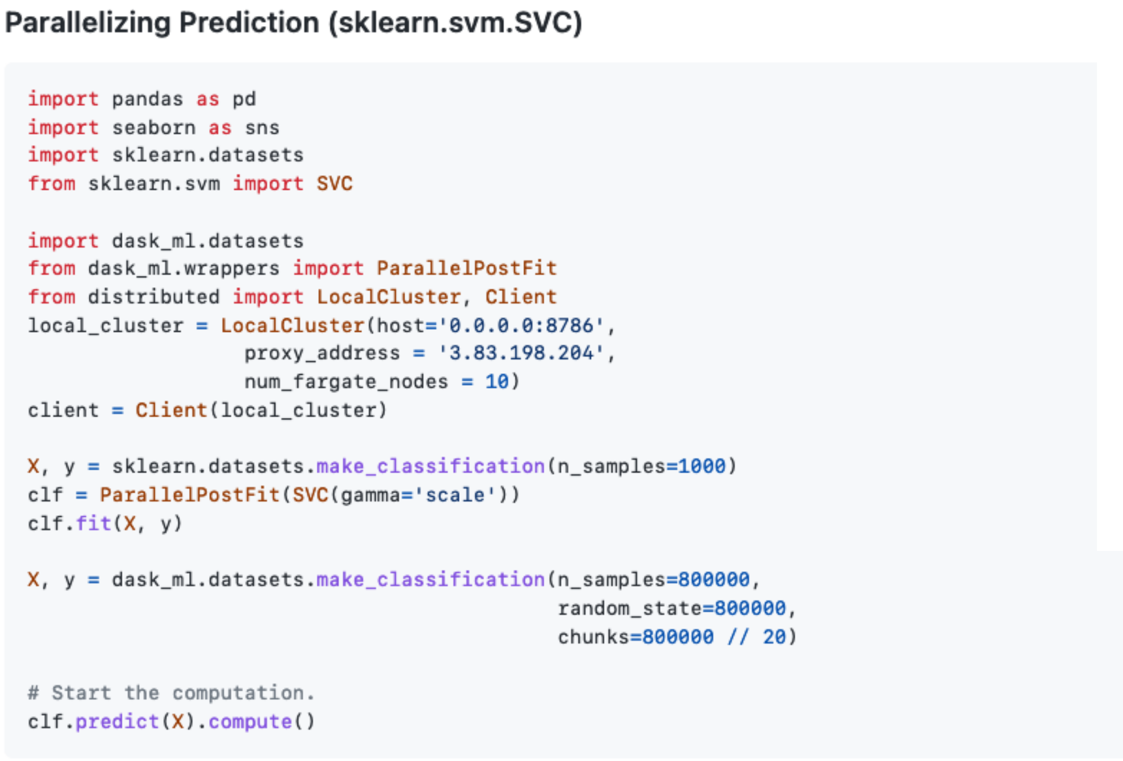
Data movement cost: TSQR



Data movement cost: TSQR

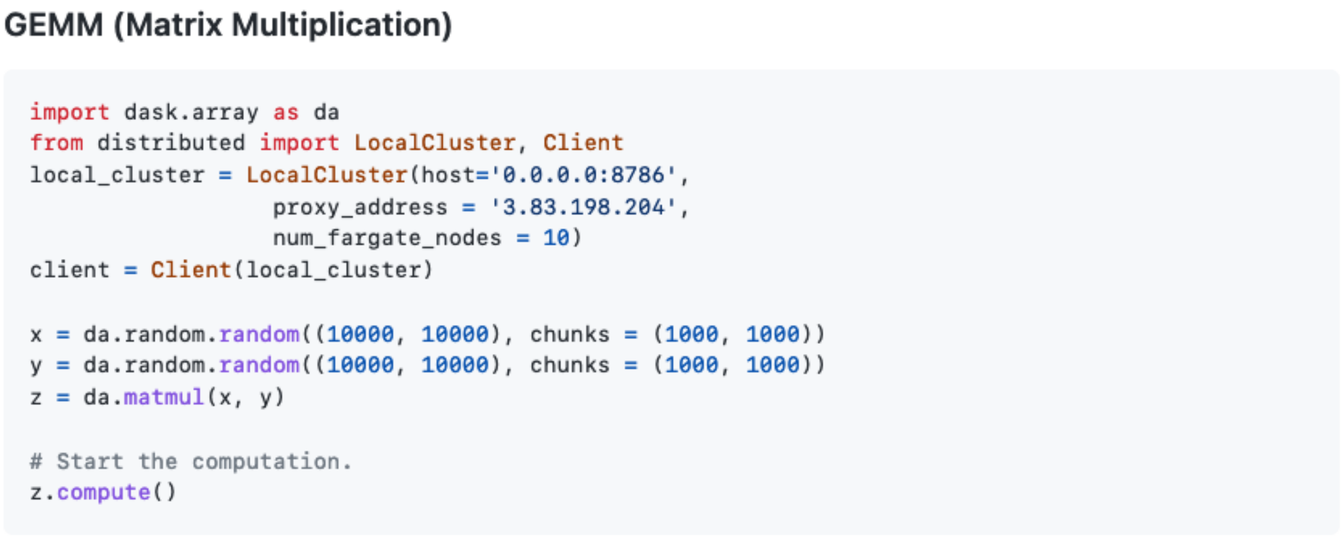


Wukong reads and writes considerably less data than numpywren.



Parallelizing Prediction (sklearn.svm.SVC)

```
import pandas as pd
import seaborn as sns
import sklearn.datasets
from sklearn.svm import SVC

import dask_ml.datasets
from dask_ml.wrappers import ParallelPostFit
from distributed import LocalCluster, Client
local_cluster = LocalCluster(host='0.0.0.0:8786',
                proxy_address = '3.83.198.204',
                num_fargate_nodes = 10)
client = Client(local_cluster)

X, y = sklearn.datasets.make_classification(n_samples=1000)
clf = ParallelPostFit(SVC(gamma='scale'))
clf.fit(X, y)

X, y = dask_ml.datasets.make_classification(n_samples=800000,
                                            random_state=800000,
                                            chunks=800000 // 20)

# Start the computation.
clf.predict(X).compute()
```



GEMM (Matrix Multiplication)

```
import dask.array as da
from distributed import LocalCluster, Client
local_cluster = LocalCluster(host='0.0.0.0:8786',
                proxy_address = '3.83.198.204',
                num_fargate_nodes = 10)
client = Client(local_cluster)

x = da.random.random((10000, 10000), chunks = (1000, 1000))
y = da.random.random((10000, 10000), chunks = (1000, 1000))
z = da.matmul(x, y)

# Start the computation.
z.compute()
```

https://github.com/ds2-lab/Wukong