

---

# Wukong

**Ben Carver, Jingyuan Zheng, Ao Wang, Ali Anwar, Panruo Wu, Yu**

**Apr 25, 2023**



# CONTENTS

<b>1</b>	<b>Installing Wukong</b>	<b>3</b>
1.1	AWS VPC Deployment & Installation . . . . .	3
1.2	AWS Lambda Setup Guide . . . . .	5
<b>2</b>	<b>Wukong Design</b>	<b>11</b>
2.1	Wukong Architecture . . . . .	11
2.2	Core Code Components . . . . .	14
2.3	Design Introduction . . . . .	14
2.4	Scheduling in Wukong . . . . .	15
<b>3</b>	<b>Static Scheduling</b>	<b>17</b>
<b>4</b>	<b>Task Execution &amp; Dynamic Scheduling</b>	<b>19</b>
<b>5</b>	<b>Example Code</b>	<b>21</b>
5.1	Initialization Code . . . . .	21
5.2	Linear Algebra . . . . .	21
5.3	Machine Learning . . . . .	23
<b>6</b>	<b>Introduction</b>	<b>25</b>
6.1	Getting Started . . . . .	25
<b>7</b>	<b>Indices and tables</b>	<b>27</b>







## INSTALLING WUKONG

In this section, you will learn how to install and deploy Wukong.

As of right now, only AWS Lambda is supported. Support for additional serverless frameworks will be provided in the future.

### 1.1 AWS VPC Deployment & Installation

Wukong is designed to run atop AWS Lambda and AWS EC2. In order for Wukong to execute properly, there are several AWS components that must be created. These include a Virtual Private Cloud (VPC), an Internet Gateway, a NAT Gateway, various subnets, and an AWS Fargate cluster.

#### 1.1.1 Automatic VPC Deployment

To simplify the deployment process, we provide a setup script that automatically creates the required AWS infrastructure. This script is located in `Wukong/Static Scheduler/install/aws_setup.py`.

You may run this script in order to create the required AWS components automatically. Alternatively, you may follow the instructions below to create and deploy the necessary AWS infrastructure manually.

#### 1.1.2 Manual VPC Deployment

Users who wish to go through the deployment process manually should follow these instructions. Deploying Wukong manually is more complicated than using the script, but doing so allows users to configure the AWS resources according to their exact needs.

##### Step 1 - Create the Virtual Private Cloud (VPC)

A VPC must be configured as EC2 instances are required to run within a VPC.

Go to the VPC Console within AWS and click on “Your VPCs” (from the menu on the left-hand side of the screen). Next, click the blue “Create VPC” button. See Figure 14 for a snapshot. Provide a name tag and an IPv4 CIDR block. It is not necessary to provide an IPv6 CIDR Block.

For Tenancy, select *Default*. Once you have completed all of the fields, click the blue Create button. You should wait until the VPC has been created before continuing.

## **Step 2 - Create the Security Group**

Next, you should create a security group for the AWS EC2 and AWS Fargate virtual machines.

From the VPC Console, select Security Groups from the menu on the left-hand side of the screen. Then click the blue Create Security Group button.

You will need to provide a security group name and a description. In the VPC field, select the VPC you created above.

## **Step 3 - Configure the Inbound & Outbound Rules for the Security Group**

Next, configure the inbound and outbound rules for the security group. This ensures that the different components will be able to communicate with one another (e.g. AWS Lambda and the KV Store).

## **Step 4 - Allocate an Elastic IP Address**

Next, allocate an Elastic IP Address. From the VPC console, select “Elastic IPs” and click the blue “Allocate new address” button.

## **Step 5 - Create the Internet Gateway**

From the VPC console, select “Internet Gateways” and click the blue “Create internet gateway” button. Supply a name tag and click “Create”.

## **Step 6 - Create Subnets for the VPC**

If you want to allocate more IPv4 CIDR blocks to your VPC, then simply go to the VPC console and select the VPC from the list of VPCs (under the Your VPCs menu option). Click the white Actions button, then select “Edit CIDRs”. At the bottom, you will see a table with a white “Add IPv4 CIDR” button underneath it. Click that button and specify the CIDR block.

You then need to create a subnet for your KV Store and EC2 instances. To create a subnet, first go to the VPC console. Then select “Subnets” from the menu on the left. Finally, click the blue “Create subnet” button.

## **Step 7 - Modify Route Tables**

From the VPC Console, select “Route Tables” and then click the blue “Create route table” button.

## **Step 8 - Create a NAT Gateway**

Next, create a NAT Gateway. A NAT Gateway is placed into a public VPC subnet to enable outbound internet traffic from instances in a private subnet. We placed this NAT Gateway in one of the subnets created for the EC2/KV Store.



## 1.2 AWS Lambda Setup Guide

Wukong uses two AWS Lambda functions to execute workloads. One function is simply used to rapidly scale-up, and we refer to this as a Wukong Invoker Function. The other function is the actual Wukong Serverless Executor, often referred to simply as an Executor.

Just as with the AWS VPC setup, users may choose between using an automated and manual setup. The automated setup for AWS Lambda involves using an AWS Serverless Application Model (SAM) template, which automatically configures and deploys the AWS Lambda functions.

Alternatively, users can opt to manually create and deploy the AWS Lambda functions. This may enable users to configure the Lambda functions more precisely.

### 1.2.1 Automatic AWS Lambda Deployment

The files required to deploy the AWS Lambda functions via AWS SAM template can be found in the `/Wukong/AWS Lambda Task Executor/` directory. Much of the following information can be found in the `SAMREADME.md` file.

NOTE: Ensure you have installed the `boto3` Python module and the AWS CLI on your computer. Likewise, ensure the AWS CLI has been configured so that `boto3` can find and use your AWS credentials.

### 1.2.2 Manual AWS Lambda Deployment

This section will explain the steps required to deploy the AWS Lambda functions manually. We will first create an AWS Lambda function for the Serverless Executor. After that, we will create an AWS Lambda function for the Serverless Invoker.

#### Step 1 - Clone the Source Code

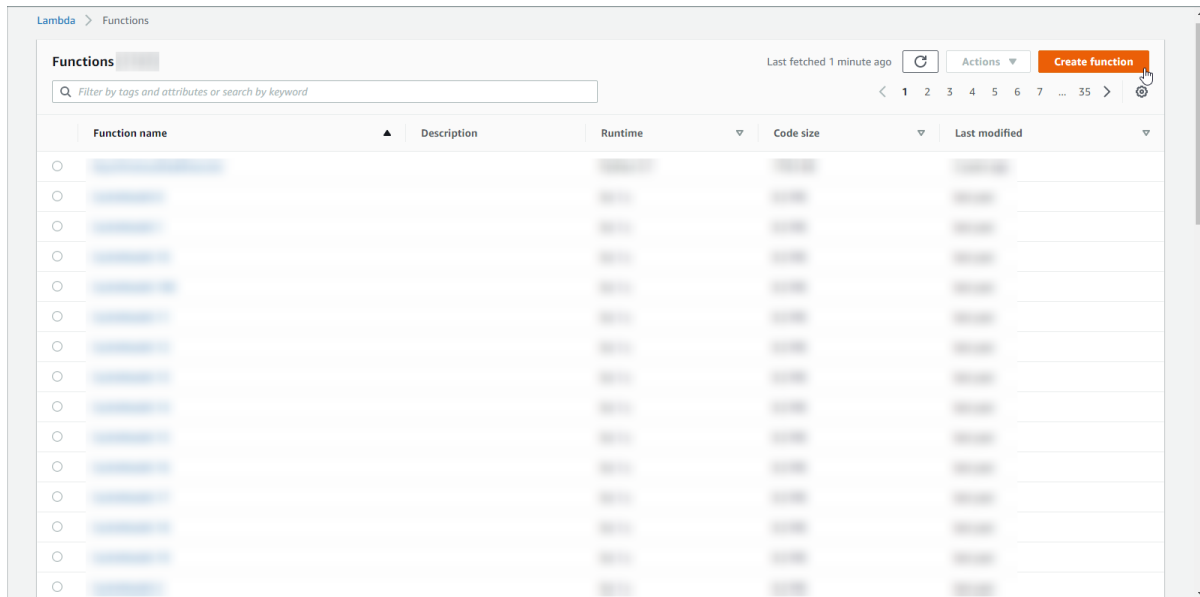
First, you should ensure you have cloned the source code from the GitHub repository.:

```
git clone https://github.com/mason-leap-lab/Wukong/tree/socc2020
```

Next, you will want to create a ZIP of all the source files in `Wukong/AWS Lambda Task Executor/TaskExecutor/`. You may name this zip file whatever you want; for example, `deployment.zip`.

#### Step 2 - Create an AWS Lambda Function

Next, we need to create an AWS Lambda function. This can be done in several ways. In this case, we will be using the AWS Lambda web console. Navigate to the AWS Lambda console and click the “Create function” button (pictured below).



You will be shown a creation screen through which you may specify various information about the function.

For the name, you should use `WukongExecutor`. If you use a different name, then you will have to modify the `executor_function_name` property in the configuration file for the Static Scheduler so that it knows which function to use. This file is called `wukong-config.yaml` and can be found in the `Wukong/Static Scheduler/` directory.

For the Runtime, select Python 3.8.

This is a screenshot of the 'Create function' wizard in the AWS Lambda console, specifically the 'Basic information' step. The 'Function name' field contains 'WukongExecutor'. Below it, a note says 'Use only letters, numbers, hyphens, or underscores with no spaces.' The 'Runtime' dropdown menu is set to 'Python 3.8'. The 'Permissions' section is expanded, showing 'Change default execution role'. Under 'Execution role', three options are listed: 'Create a new role with basic Lambda permissions', 'Use an existing role' (which is selected with a radio button), and 'Create a new role from AWS policy templates'. Below these, the 'Existing role' dropdown is empty, and there's a 'Refresh' icon. A link to the IAM console is provided for creating a custom role.


## Create an IAM Role


Under Permissions, select **Change default execution role**. Some additional options will be displayed. Click the hyperlink “IAM Console”; this should open the IAM role creation page in a new browser tab.


Create role


1234

Select type of trusted entity


**AWS service**  
 EC2, Lambda and others


**Another AWS account**  
 Belonging to you or 3rd party


**Web identity**  
 Cognito or any OpenID provider


**SAML 2.0 federation**  
 Your corporate directory

Allows AWS services to perform actions on your behalf. [Learn more](#)

Choose a use case

Common use cases

**EC2**  
 Allows EC2 instances to call AWS services on your behalf.

**Lambda**  
 Allows Lambda functions to call AWS services on your behalf.

Or select a service to view its use cases

<a href="#">API Gateway</a>	<a href="#">CloudWatch Events</a>	<a href="#">EKS</a>	<a href="#">IoT Things Graph</a>	<a href="#">Redshift</a>
<a href="#">AWS Backup</a>	<a href="#">CodeBuild</a>	<a href="#">EMR</a>	<a href="#">KMS</a>	<a href="#">Rekognition</a>
<a href="#">AWS Chatbot</a>	<a href="#">CodeDeploy</a>	<a href="#">ElastiCache</a>	<a href="#">Kinesis</a>	<a href="#">RoboMaker</a>
<a href="#">AWS Marketplace</a>	<a href="#">CodeGuru</a>	<a href="#">Elastic Beanstalk</a>	<a href="#">Lake Formation</a>	<a href="#">S3</a>
<a href="#">AWS Support</a>	<a href="#">CodeStar Notifications</a>	<a href="#">Elastic Container Registry</a>	<a href="#">Lambda</a>	<a href="#">SMS</a>
<a href="#">Amplify</a>	<a href="#">Comprehend</a>	<a href="#">Elastic Container Service</a>	<a href="#">Lex</a>	<a href="#">SNS</a>
<a href="#">AppStream 2.0</a>	<a href="#">Config</a>	<a href="#">Elastic Transcoder</a>	<a href="#">License Manager</a>	<a href="#">SWF</a>
<a href="#">AppSync</a>	<a href="#">Connect</a>	<a href="#">ElasticLoadBalancing</a>	<a href="#">MQ</a>	<a href="#">SageMaker</a>
<a href="#">Application Auto Scaling</a>	<a href="#">DMS</a>	<a href="#">Forecast</a>	<a href="#">Machine Learning</a>	<a href="#">Security Hub</a>
<a href="#">Application Discovery Service</a>	<a href="#">Data Lifecycle Manager</a>	<a href="#">GameLift</a>	<a href="#">Macie</a>	<a href="#">Service Catalog</a>
	<a href="#">Data Pipeline</a>	<a href="#">Global Accelerator</a>	<a href="#">Managed Blockchain</a>	<a href="#">Step Functions</a>

\* Required

Cancel

Next: Permissions

From here, you should select the following three policies:

- `arn:aws:iam::aws:policy/AWSLambdaFullAccess`
- `arn:aws:iam::aws:policy/AWSXrayWriteOnlyAccess`
- `arn:aws:iam::aws:policy/AmazonS3FullAccess`

You can use the search functionality to quickly locate these policies in the list. Once you’ve selected these three policies, you can click the blue “Next: Tags” button in the lower-right, and then immediately the “Next: Review” button.

For Role name, you may specify whatever you want – for example, `wukong-role`. Once you’ve typed in a name, click the “Create role”.

Return to the AWS Lambda tab. Click the “reload” button to the right of the “Existing role” drop-down menu. Then find the newly-created IAM role in the list and select the role.

Once selected, click the orange “Create function” button in the lower right. (You may need to scroll down a bit first in order to see the button.)

## Add the Required Lambda Layers

Next, you will need to add four layers to the function. AWS Lambda Layers are basically archives that may contain libraries, custom runtimes, or other required dependencies. Layers are useful as they allow users to include additional libraries in their function without needing to include the libraries in the deployment package.

Scroll down to the “Layers” section and click the “Add a layer” button. Select “Specify an ARN”. Below is a list of layer ARN’s. You should repeat these steps, specifying each of the ARN’s found in the list.

1. `arn:aws:lambda:us-east-1:668099181075:layer:AWSLambda-Python37-SciPy1x:2`
2. `arn:aws:lambda:us-east-1:205616672683:layer:DaskDependencies:2`
3. `arn:aws:lambda:us-east-1:561589293384:layer:DaskLayer2:2`
4. `arn:aws:lambda:us-east-1:561589293384:layer:dask-ml-layer:9`

The first layer contains Numpy and Scipy, two Python modules required by the Wukong Executor. The next layer contains the Python dependencies of Dask along with the AWS X-Ray API, which is used for debugging and metadata. The third layer contains Dask itself, and the last layer contains Dask-ML and its dependencies.

### Add layer

#### Choose a layer [Info](#)

Choose from layers with a compatible runtime or specify the Amazon Resource Name (ARN) of a layer version.

☐ **AWS layers**  
Choose a layer from a list of layers provided by AWS.

☐ **Custom layers**  
Choose a layer from a list of layers created by your AWS account or organization.

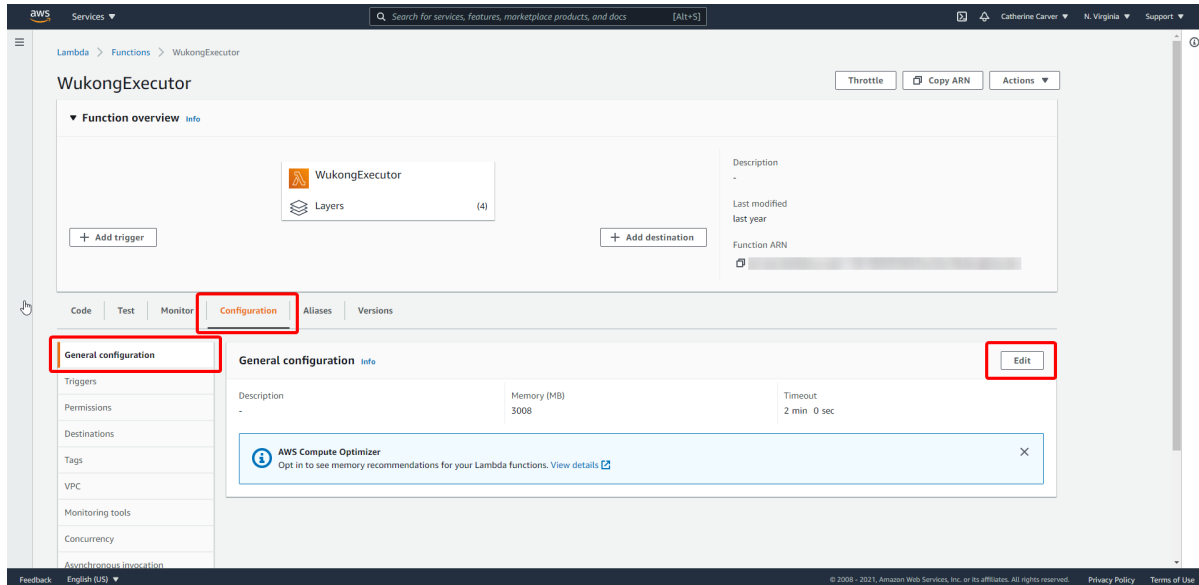
☒ **Specify an ARN**  
Specify a layer by providing the ARN.

**Specify an ARN**  
Specify a layer by providing the Amazon Resource Name (ARN).

[Cancel](#) [Add](#)

## General Configuration

Once you have added the Lambda Layers to the function, you should modify the “General configuration” of the function. This includes the function’s memory (RAM) and Timeout (i.e., how long the function can execute for). To change these values, select the “Configuration” tab. Then select “General configuration” from the list of buttons on the left. Finally, click the “Edit” button.



You will be presented with a “Basic settings” menu through which you may modify the amount of RAM that gets allocated to the function as well as the function’s timeout.

**Attention:** If you are not sure what values to specify for **Memory (MB)** or **Timeout**, we recommend at least 1,024 MB and 30 seconds.

**Warning:** The amount billed for executing an AWS Lambda function is dependent on memory. Increasing the amount of memory allocated to your function may make it more expensive to run.

## Uploading the Deployment Package

The last step is to upload the deployment package, which contains the source code for your AWS Lambda function.

First, select the “Code” tab (from the same section that had the “Configuration” tab). On the right, you will see an “Upload from” button. Click this, and then select “.zip file”.

Use the upload dialog to upload the .ZIP file you created earlier.

Next, scroll down to the “Runtime settings” section. Select the “Edit” button on the right. You just need to modify the `Handler` field.

Replace whatever is currently there with `function.lambda_handler` and click “Save”.

Congratulations! You have successfully deployed the Wukong Serverless Executor.

A majority of the required AWS infrastructure can be created using the provided `aws_setup.py` script in `Wukong/Static Scheduler/install/` directory. Please be sure to read through the `wukong_setup_config.yaml` configuration file located in

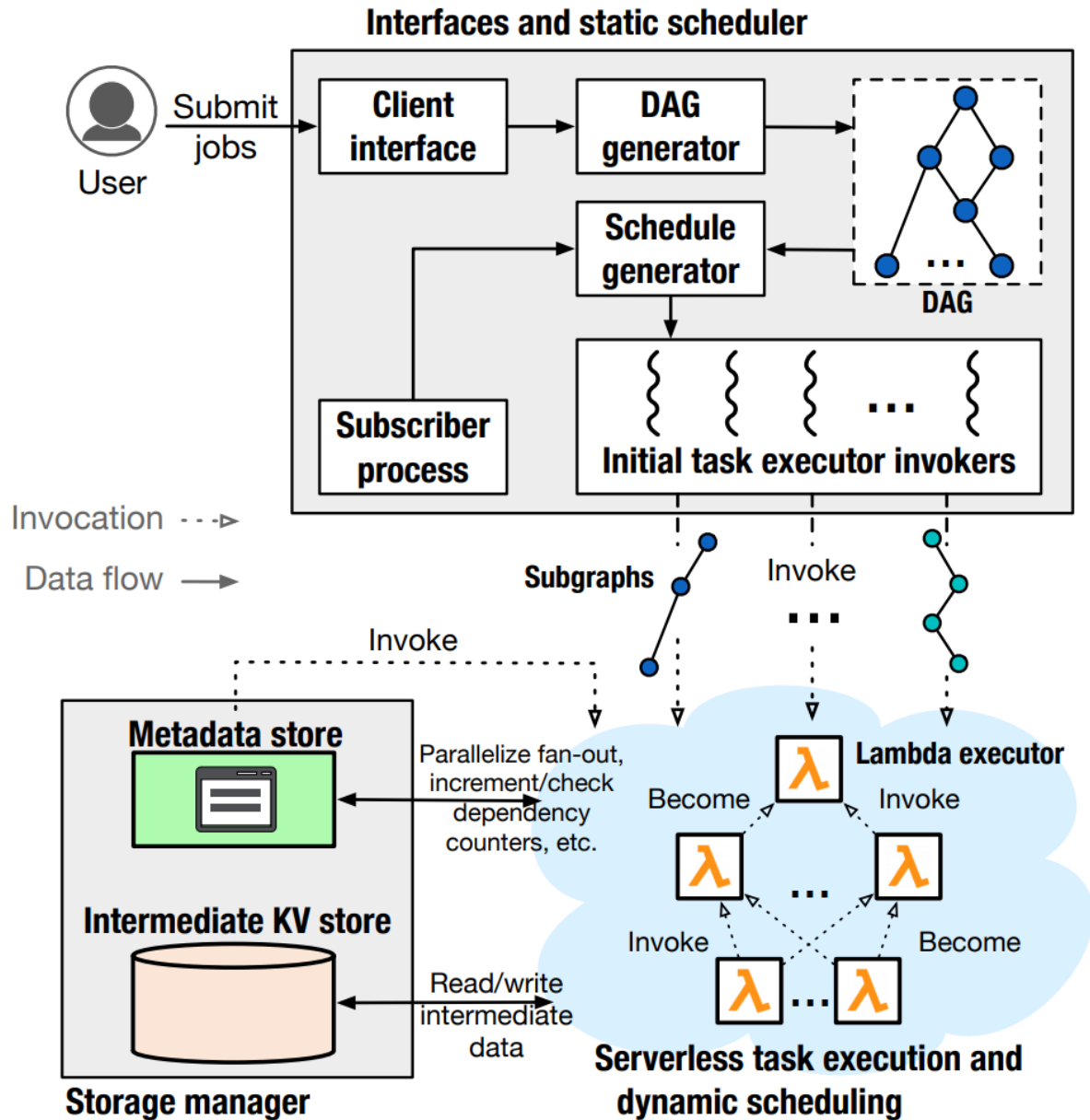
the same directory prior to running the script. In particular, your public IP address should be added to the configuration file if you'd like SSH to be enabled from your machine to VMs created in the Wukong VPC.

**NOTE:** Ensure you have installed the `boto3` Python module and the AWS CLI on your computer. Likewise, ensure the AWS CLI has been configured so that `boto3` can find and use your AWS credentials.

## WUKONG DESIGN

### 2.1 Wukong Architecture

Wukong is composed of three primary components: 1. Static Scheduler 2. Serverless Task Executors 3. Storage Manager



The following sections will describe these components in greater detail.

### 2.1.1 Static Scheduler

The **Static Scheduler** in Wukong serves as a front-end client interface to the framework. Users can submit jobs via the Client Interface. These jobs will be passed to the DAG Generator, which will convert the user-written Python code into an executable DAG.

The generated DAG will then be handed off to Wukong’s Schedule Generator. The schedule generator performs a series of depth-first searches on the DAG to partition it into a collection of sub-graphs, referred to as **static schedules**. Finally, the static schedules corresponding to so-called “leaf tasks” (i.e., tasks with no data dependencies) are assigned to Serverless Executors by the Initial Task Executor Invokers. Specifically, the static schedules are serialized and included in invocation payloads for the Serverless Executors.



## 2.1.2 Serverless Task Executors

The Serverless Task Executors (often referred to simply as “Executors”) are the workers of Wukong. Each Executor is simply an on-going invocation of the AWS Lambda function. When an Executor begins running, it retrieves its assigned static schedule from the invocation payload or from intermediate storage. (Invocation payloads have a size limit of 256kB, meaning some data may need to be stored in intermediate storage rather than included directly within the invocation payload.) When an Executor runs out of work, it simply terminates, rather than waiting for more work from the Scheduler or fetching more work from an external queue.

### Dependency Tracking

Executors communicate with one another through intermediate storage. Each task in the DAG has an associated “dependency counter” maintained within the Metadata Store (MDS), a component of the Storage Manager. Each time an Executor completes a task, the Executor increments the dependency counter of each of the completed task’s dependents. Executors can check whether a task is ready to execute by examining the value of the task’s dependency counter. If the value of the counter is equal to the number of dependencies of the task, then the task is ready to execute. To better illustrate this process, consider the following example.

### Dependency Tracking Example

The diagram below shows a simple DAG containing three tasks: Task A, Task B, and Task C. This DAG will be executed by two Executors: Executor #1 and Executor #2.

Looking at the structure of the DAG, we can see that Task C is dependent on Task A and Task B. As a result, Task C will not be able to execute until both Task A and Task B have been completed.

Assume that Executor #1 completes Task A first (i.e., before Executor #2 completes Task B). Executor #1 will next examine the dependents – also known as the *downstream tasks* – of Task A. In this case, there is just one downstream task, Task C. Executor #1 will increment the dependency counter of Task C by 1, thereby indicating that one dependency of Task C has resolved (i.e., has been executed).

After incrementing Task C’s dependency counter, Executor #1 will check to see if Task C is ready to execute. It will compare the current value of Task C’s dependency counter against the number of data dependencies of Task C. In this case, Task C has 2 dependencies and its dependency counter has the value 1. Because these values are not equal, Executor #1 will determine that Task C cannot be executed yet, and Executor #1 will terminate. The result of this is shown in the diagram below.

Next, assume that Executor #2 completes Task B. Executor #2 will next check for any downstream tasks of Task B and discover Task C. Now Executor #2 will increment the dependency counter of Task C. Prior to this increment operation, the value of Task C’s dependency counter is 1. After Executor #2 increments Task C’s dependency counter, the value of the counter is 2. Executor #2 compares the value of Task C’s dependency counter against the number of dependencies of Task C.

At this point, Executor #2 will find that the two values are equal (they are both 2). Executor #2 will conclude that Task C is ready to execute. In order to execute Task C, Executor #2 will first retrieve the output of Task A from intermediate storage. Once obtained, Executor #2 will have satisfied all data dependencies of Task C and will proceed to execute Task C.

Nam erat dolor, porta sit amet ultricies vel, scelerisque at sapien. Quisque eleifend magna at pharetra suscipit. Proin eu pretium nisi. Praesent ante velit, hendrerit vitae sagittis sit amet, ultricies ac dolor. Vivamus pharetra vitae nisl et ornare. Pellentesque tincidunt eleifend accumsan. Sed augue nisl, sagittis ut scelerisque eu, imperdiet quis nisi. Praesent auctor consectetur risus, in lacinia elit consequat ac.

### 2.1.3 Storage Manager

The Storage Manager abstractly defines the intermediate storage of Wukong. It is composed of a Metadata Store (MDS) and an Intermediate KV Store (KVS). The MDS is simply a single instance of Redis running on an AWS EC2 virtual machine. The KVS is an AWS Fargate cluster in which each AWS Fargate node is running a Redis server.

#### The Metadata Store (MDS)

The MDS is simply a Redis server running on an AWS EC2 virtual machine. Typically, the MDS will be running on a separate EC2 VM from the Static Scheduler. There is also a KVS Proxy running on the MDS virtual machine. The KVS Proxy aids in invoking downstream tasks and storing and transferring intermediate data. Typically, the KVS Proxy is only utilized in cases where many downstream tasks need to be invoked all at once. The higher network bandwidth of the KVS Proxy is beneficial in transferring intermediate data to newly-invoked Executors.

The Static Scheduler stores static schedules and dependency counters in the MDS at the very beginning of a workload. Executors will fetch static schedules and increment/retrieve dependency counters during a workload's execution. Additionally, the final result(s) of a workload will be stored within the MDS rather than the KVS.

#### The Intermediate Key-Value Store (KVS)

The KVS consists of an AWS Fargate cluster. The Wukong static scheduler will scale-up the AWS Fargate cluster according to the user's specification. Each virtual machine within the AWS Fargate cluster will be configured with 4 CPU cores and 32GB of RAM (though users could modify these values if they desire). Each of the AWS Fargate nodes houses an active Redis server. Intermediate data is stored in these servers during a workload's execution. Data is consistently hashed across the entire AWS Fargate cluster in order to ensure a relatively balanced distribution of data.

## 2.2 Core Code Components

### 2.3 Design Introduction

Wukong is a serverless-oriented, decentralized, locality-aware, and cost-effective parallel computing framework. The key insight of Wukong is that partitioning the work of a centralized scheduler (i.e., tracking task completions, identifying and dispatching ready tasks, etc.) across a large number of Lambda executors, can greatly improve performance by permitting tasks to be scheduled in parallel, reducing resource contention during scheduling, and making task scheduling data locality-aware, with automatic resource elasticity and improved cost effectiveness.

Scheduling is decentralized by partitioning a DAG into multiple, possibly overlapping, subgraphs. Each subgraph is assigned to a task Executor (implemented as an AWS Lambda function runtime) that is responsible for scheduling and executing tasks in its assigned subgraph. This decentralization brings multiple key benefits.

The design consists of three major components: a static schedule generator which runs on Amazon EC2, a pool of Lambda Executors, and a storage cluster.

### 2.3.1 Enhanced Data Locality & Reduced Resource Contention

Decentralization improves the data locality of scheduling. Unlike PyWren and numpywren, which require executors to perform network I/Os to obtain each task they execute (since numpywren's task executor is completely stateless), Wukong preserves task dependency information on the Lambda side. This allows Lambda executors to cache intermediate data and schedule the downstream tasks in their subgraph locally, i.e., without constant remote interaction with a centralized scheduler.

### 2.3.2 Harnessing Scale & Local Optimization Opportunities

Decentralizing scheduling allows an Executor to make local data-aware scheduling decisions about the level of task granularity (or parallelism) appropriate for its subgraph. Agile executors can scale out compute resources in the face of burst-parallel workloads by partitioning their subgraphs into smaller graphs that are assigned to other executors for an even higher level of parallel task scheduling and execution. Alternately, an executor can execute tasks locally, when the cost of data communication between the tasks outweighs the benefit of parallel execution.

### 2.3.3 Automatic Resource Elasticity & Improved Cost Effectiveness

Decentralization does not require users to explicitly tune the number of active Lambdas running as workers and thus is easier to use, more cost effective, and more resource efficient.

## 2.4 Scheduling in Wukong

Scheduling in Wukong is decentralized and uses a combination of **static** and **dynamic scheduling**. A **static schedule** is a subgraph of the DAG. Each static schedule is assigned to a separate Executor.

An Executor uses **dynamic scheduling** to enforce the data dependencies of the tasks in its static schedule. An Executor can invoke additional Executors to increase task parallelism, or cluster tasks to eliminate any communication delay between them. Executors store intermediate task results in an elastic in-memory key-value storage (KVS) cluster (hosted using AWS Fargate) and job-related metadata (e.g., counters) in a separate KVS that we call metadata store (MDS).



## STATIC SCHEDULING

Wukong users submit a Python computing job to the DAG generator, which uses the Dask library to convert the job into a DAG. The Static-Schedule Generator generates *static schedules* from the DAG.



## **TASK EXECUTION & DYNAMIC SCHEDULING**

Wukong workflow execution starts when the static scheduler's Initial-Executor Invokers assign each static schedule produced by the Static-Schedule Generator to a separate Executor. Recall that each static schedule begins with one of the leaf node tasks in the DAG. The InitialExecutor invokes these "leaf node" Executors in parallel. After executing its leaf node task, each Executor then executes the tasks along a single path through its schedule.

An Executor may execute a sequence of tasks before it reaches a fan-out operation with more than one out edge or it reaches a fan-in operation. For such a sequence of tasks, there is no communication required for making the output of the earlier tasks available to later tasks for input. All intermediate task outputs are cached in the Executor's local memory with inherently enhanced data locality. Executors also look ahead to see what data their tasks may need, which allows them to discard data in their local memory that is no longer needed.





## EXAMPLE CODE

## 5.1 Initialization Code

Everytime you run a job on Wukong, you'll need to create an instance of the `LocalCluster` class as well as an instance of the `Client` class.

```
1 import dask.array as da
2 from wukong import LocalCluster, Client
3 local_cluster = LocalCluster(
4     host="10.0.88.131:8786",
5     proxy_address = "10.0.88.131",
6     proxy_port = 8989,
7     num_lambda_invokers = 4,
8     chunk_large_tasks = False,
9     n_workers = 0,
10    use_local_proxy = True,
11    local_proxy_path = "/home/ec2-user/Wukong/KV Store Proxy/proxy.py",
12    redis_endpoints = [("127.0.0.1", 6379)],
13    use_fargate = False)
14 client = Client(local_cluster)
```

In all of the following examples, the code given assumes you've created a local cluster and client object first.

## 5.2 Linear Algebra

Wukong supports many popular linear algebra operations such as Singular Value Decomposition (SVD) and TSQR (Tall-and-Skinny QR Reduction).

### 5.2.1 Singular Value Decomposition (SVD)

#### Tall-and-Skinny Matrix

Here, we are computing the SVD of a 200,000 x 100 matrix. In this case, we partition the original matrix into chunks of size 10,000 x 100.

```
1 X = da.random.random((200000, 100), chunks=(10000, 100)).persist()
2 u, s, v = da.linalg.svd(X)
3 v.compute(scheduler = client.get)
```

## Square Matrix

We can also compute the SVD of a square matrix – in this case, the input matrix is size 10,000 x 10,000. We partition this input matrix into chunks of size 2,000 x 2,000 in this example.

```
1 X = da.random.random((10000, 10000), chunks=(2000, 2000)).persist()
2 u, s, v = da.linalg.svd_compressed(X, k=5)
3 v.compute(scheduler = client.get)
```

## 5.2.2 QR Reduction

```
1 X = da.random.random((128, 128), chunks = (16, 16))
2 q, r = da.linalg.qr(X)
3 r.compute(scheduler = client.get)
```

## 5.2.3 Tall-and-Skinny QR Reduction (TSQR)

We can also compute the tall-and-skinny QR reduction of matrices using Wukong.

```
1 X = da.random.random((262_144, 128), chunks = (8192, 128))
2 q, r = da.linalg.tsqr(X)
3 r.compute(scheduler = client.get)
```

## 5.2.4 Cholesky Decomposition

```
1 def get_sym(input_size):
2     A = da.ones((input_size,input_size), chunks = chunks)
3     lA = da.tril(A)
4     return lA.dot(lA.T)
5
6 input_matrix = get_sym(100)
7 X = da.asarray(input_matrix, chunks = (25,25))
8
9 # Pass 'True' for the 'lower' parameter if you wish to compute the lower cholesky_
10 ↪decomposition.
11 chol = da.linalg.cholesky(X, lower = False)
12 chol.compute(scheduler = client.get)
```

## 5.2.5 General Matrix Multiplication (GEMM)

```
1 x = da.random.random((10000, 10000), chunks = (2000, 2000))
2 y = da.random.random((10000, 10000), chunks = (2000, 2000))
3
4 z = da.matmul(x, y)
5 z.compute(scheduler = client.get)
```

## 5.3 Machine Learning

Wukong also supports many machine learning workloads through the use of Dask-ML.

### 5.3.1 Support Vector Classification (SVC)

```
1 import pandas as pd
2 import seaborn as sns
3 from collections import defaultdict
4 import sklearn.datasets
5 from sklearn.svm import SVC
6
7 import dask_ml.datasets
8 from dask_ml.wrappers import ParallelPostFit
9
10 X, y = sklearn.datasets.make_classification(n_samples=1000)
11 clf = ParallelPostFit(SVC(gamma='scale'))
12 clf.fit(X, y)
13
14 results = defaultdict(list)
15
16 X, y = dask_ml.datasets.make_classification(n_samples = 100000,
17                                           random_state = 100000,
18                                           chunks = 100000 // 20)
```



## INTRODUCTION

Wukong is a serverless-oriented, locality-aware DAG scheduler built atop AWS Lambda and AWS EC2. Wukong utilizes the programming model of Dask & Dask Distributed to convert user-written Python code into an executable directed acyclic graph (DAG). These DAGs are then executed on AWS Lambda in order to provide a fast, cost-effective, and easy-to-use DAG execution engine.

### 6.1 Getting Started

Wukong's serverless design enables the framework to be both easy to deploy and easy to use. Visit [Installing Wukong](#) to learn how to deploy Wukong.



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`