Ray 是伯克利大学 RISELab 研发的一个简单高效的分布式计算引擎，为开发者提供了简单通用的API来构建分布式程序。

Ray 能够让开发者轻松地构建分布式程序，靠的是通过简单的API来将计算任务分解为以下的计算原语来执行：（以下两段来自实验文档）

Task：一个无状态的计算任务（函数表示）。Ray 允许异步执行任意函数。这些"remote function"（Task）的开销非常低，可以在毫秒内执行，并且可以自动向集群添加节点并调度任务，非常适合扩展计算密集型应用程序和服务。

Actor：一个有状态的计算任务（类表示）。Actor 模型是一个强大的异步编程范例（支持微服务），可以在本地和远程无缝工作。Actor 本质上是一个有状态的 Worker（或 service）。当一个新的 Actor 被实例化时，就创建一个新的 Worker，并将该 Actor 的方法调度到这个特定的 Worker，也可以对 Worker 的状态进行访问和修改。 要获取更详细的关于Ray的基础结构的信息，可以参考原论文：[1712.05889] Ray: A Distributed Framework for Emerging AI Applications (arxiv.org)

简单的来说：Ray就是一个接口，你把任务给他，他可以帮助用户轻松地进行分解，实现分布式计算。

现在我们具体来讲一讲该怎么做：

首先，我将介绍docker和wsl；

什么是WSL

Windows Subsystem for Linux（WSL）是 Windows 10 和 Windows Server 2019 及更高版本中的一个兼容层，允许用户在 Windows 上运行 Linux 二进制可执行文件（ELF 格式）。WSL 使得用户可以在 Windows 系统上直接运行 Linux 发行版，如 Ubuntu、Debian、Kali Linux 等，而无需使用虚拟机或双引导。

简单来说，就是你可以利用它在windows系统上使用一个兼容性很好的类似于子系统的linux系统，具体的安装与下载方式如下，在命令行输入

```
wsl --install
wsl --set-default-version 2#设置wsl为默认版本
wsl#启动，第一次你会需要输入用户名与启动密码
#记得安装python环境
sudo apt update && sudo apt upgrade -y
sudo apt install python3 python3-pip -y
#验证
python3 --version
```

我们的docker是依赖于wsl的

先下载docker--desktop

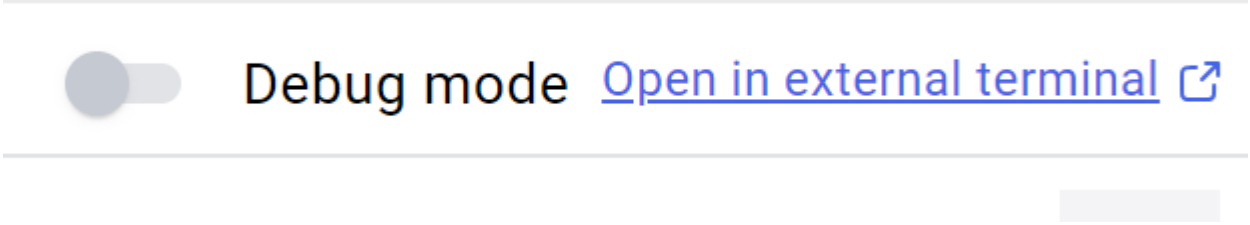
拉取Ray镜像，命令为：`docker pull rayproject/ray`（可用`docker images`或在Docker Desktop的`Images`选项中，查看当前所有镜像，以确认Ray镜像是否成功引入）。

```
docker run --shm-size=4G -t -i -p 8265:8265 -p 3000:3000 -p 9000:9000 -p 6379:6379
rayproject/ray
```

	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
☐	chroma	8a0b38f16252	chroma-core/chroma:latest	8000:8000	0%	1 month ago	▷ ⋮ 🗑
☐	suspicious_banzai	0fdbd62f917e	rayproject/ray	3000:3000 Show all ports (4)	0%	24 minutes ago	■ ⋮ 🗑
☐	docker	-	-	-	0%	26 minutes ago	▷ ⋮ 🗑

- `--shm-size`: 推荐使用4G及以上（配置不足时可以适当减少），此参数可自定义。省略此参数则使用默认空间划分。
- `-i`: 交互式操作。
- `-t`: 终端。
- `-p`: 端口映射，格式为主机端口:容器端口，可多次使用。8265端口为dashboard默认端口，3000端口为Grafana默认端口，9000端口为Prometheus默认端口，6379端口为Ray头结点连接（用于分布式部署）默认端口。

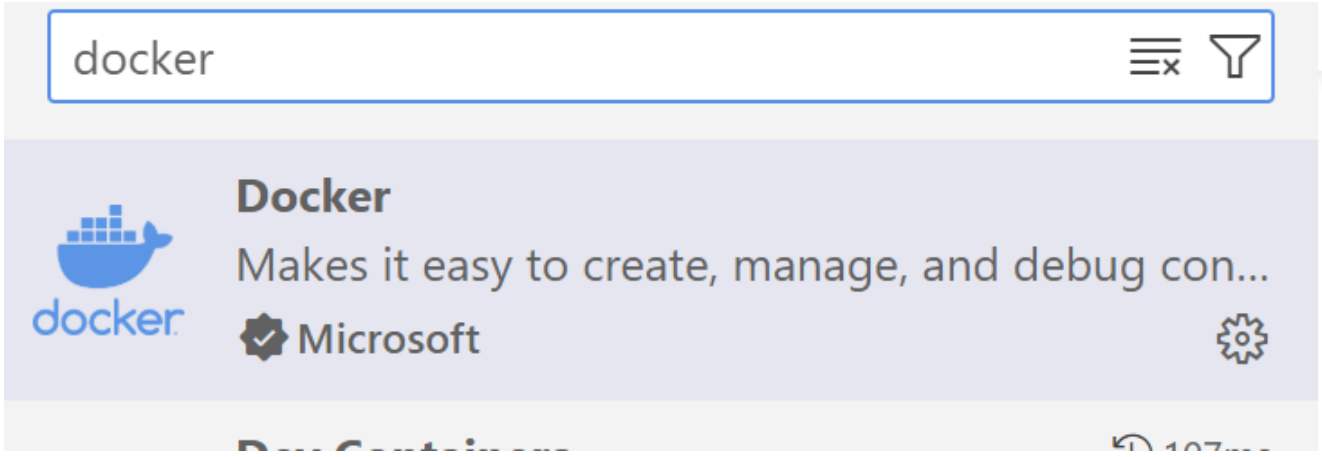
  在其中点击该项目->exe->



打开终端，注意你要自己设置ssh链接github还要自己下载相应的工具（linux环境下）

```
# 安装Ray并支持dashboard和集群启动
pip install -U "ray[default]"
# 安装Ray及其AI运行时的依赖
pip install -U "ray[air]"
pip install pytest
```

在vscode中我们可以选择如下的插件，从而实现图形化



随后在边框栏找到container就可以对于文件图形化管理了

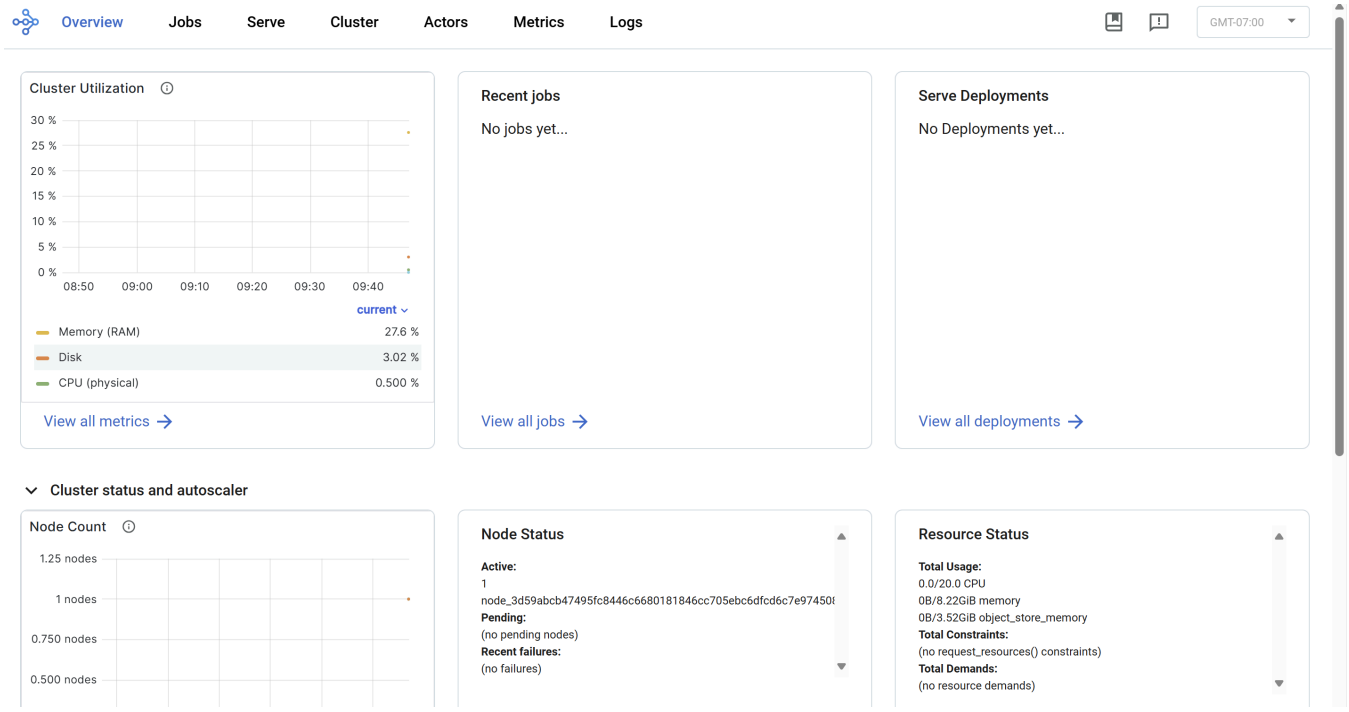
- **Prometheus**：专注于实时数据收集和存储，特别适合监控系统和服务的性能指标。
- **Grafana**：提供强大的数据可视化功能，可以将Prometheus收集的数据以图表形式展示，便于分析和评估。

```
wget https://github.com/prometheus/prometheus/releases/download/v2.37.8/prometheus-2.37.8.linux-amd64.tar.gz
tar -xzvf prometheus-*.tar.gz
wget https://dl.grafana.com/enterprise/release/grafana-enterprise-9.5.2.linux-amd64.tar.gz
tar -xzvf grafana-enterprise-9.5.2.linux-amd64.tar.gz
```

随后

```
ls
BUILD.bazel       bazel            grafana-9.5.2
pyproject.toml  semgrep.yml
CONTRIBUTING.rst  build-docker.sh  grafana-enterprise-9.5.2.linux-amd64.tar.gz
pytest.ini      setup_hooks.sh
LICENSE           ci               java
python          src
README.rst        cpp              prometheus-2.37.8.linux-amd64
release         thirdparty
SECURITY.md       doc              prometheus-2.37.8.linux-amd64.tar.gz         rllib
WORKSPACE         docker           pylintrc
scripts
#这是笔者的输出，你们只要包含对应文件就行
#跳转到对应文件夹
 cd prometheus-2.37.8.linux-amd64
 ./prometheus --config.file=/tmp/ray/session_latest/metrics/prometheus/prometheus.yml
 #另一个的命令，这里注意也要到对应目录，而且二者都是要以开一个额外的终端进行的
 ./bin/grafana-server --config /tmp/ray/session_latest/metrics/grafana/grafana.ini
web
```

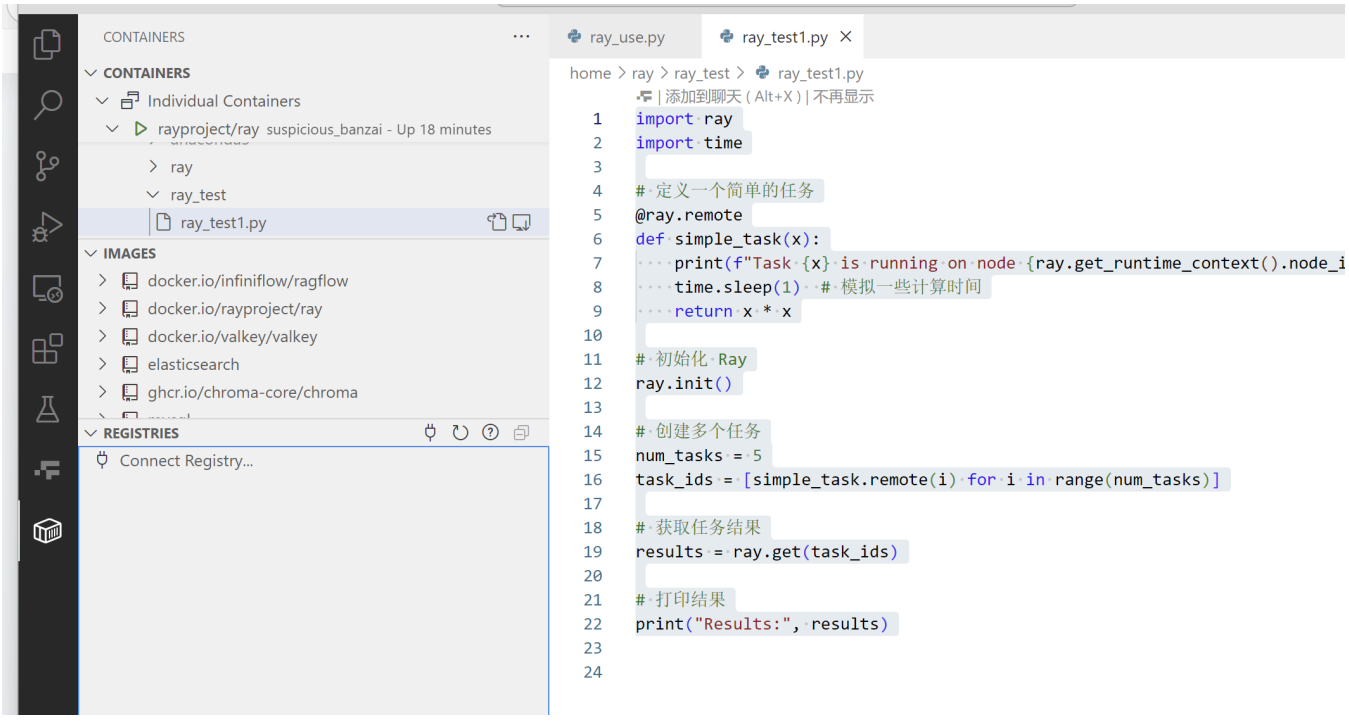
此时打开127.0.0.1:8265应该要可以看到如下的图标

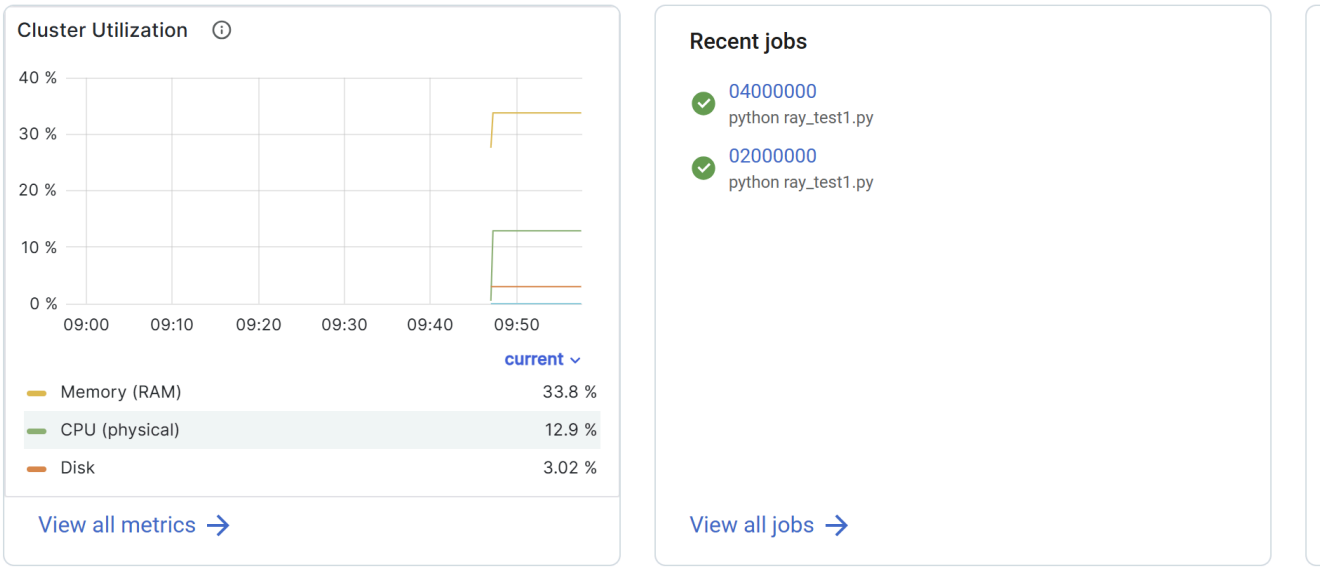


我们使用测试文件

```
import ray
import time

# 定义一个简单的任务
@ray.remote
def simple_task(x):
    print(f"Task {x} is running on node {ray.get_runtime_context().node_id}")
    time.sleep(1)  # 模拟一些计算时间
    return x * x

# 初始化 Ray
ray.init()

# 创建多个任务
num_tasks = 5
task_ids = [simple_task.remote(i) for i in range(num_tasks)]

# 获取任务结果
results = ray.get(task_ids)

# 打印结果
print("Results:", results)
```

记得使用`touch`创建文件，在vscode中打开可以直接修改





应该要能看到如图所示的结果（也就是相应的数据会有改变，recent jobs也有改变）

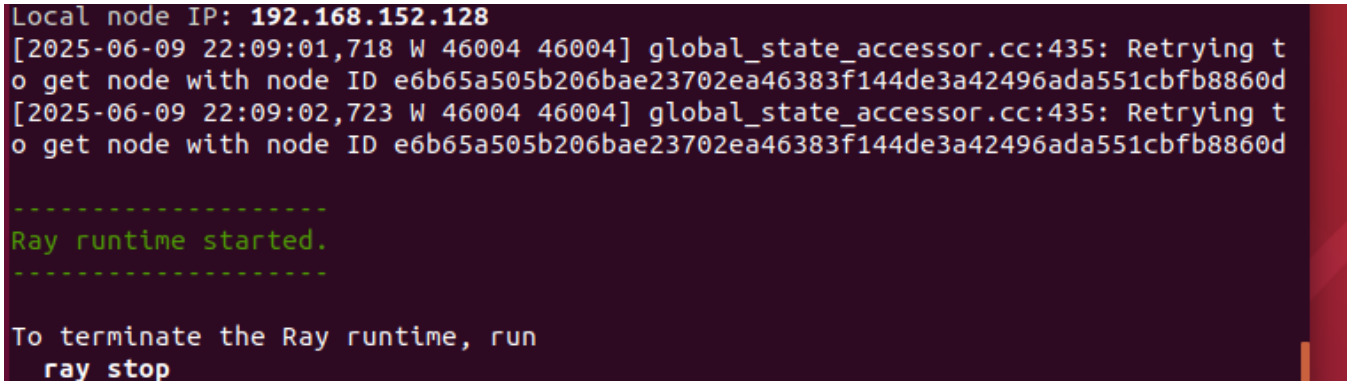
这就是相应的单机部署

接下来我们进行分布式的部署：

记住一定要保证ray对应的python版本一致，如果不一致：自行下载编译，建立虚拟环境保证一致

```
ray start --head --port=6379 --dashboard-host=0.0.0.0#必须重启
在宿主机ipconfig
ipv4地址即为你的IP
ray start --head --port=6379 --dashboard-host=0.0.0.0
ray start --address='主节点IP:6379'#从机
ray status
先启动节点，后启动别的服务
# Prometheus
./prometheus --config.file=/tmp/ray/session_latest/metrics/prometheus/prometheus.yml

# Grafana
./bin/grafana-server --config /tmp/ray/session_latest/metrics/grafana/grafana.ini web
```

```
以太网适配器 VMware Network Adapter VMnet1:

   连接特定的 DNS 后缀 . . . . . . . :
   本地链接 IPv6 地址. . . . . . . . : fe80::d664:14ad:61c7:1f2e%4
   IPv4 地址 . . . . . . . . . . . . : 192.168.217.1
   子网掩码 . . . . . . . . . . . . : 255.255.255.0
   默认网关. . . . . . . . . . . . . :
```



从机会显示相应的动态

```
import ray
import numpy as np
import time

# Define the Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# A Random-parameter Neural Network
class fake_NN:
    def __init__(self):
        self.W1 = np.random.rand(1000, 1000)
        self.B1 = np.random.rand(1000)
        self.W2 = np.random.rand(1000, 1000)
        self.B2 = np.random.rand(1000)
        self.W3 = np.random.rand(1000, 1000)
        self.B3 = np.random.rand(1000)

    def forward(self, input):
        x = sigmoid(input @ self.W1 + self.B1)
        x = sigmoid(x @ self.W2 + self.B2)
        x = sigmoid(x @ self.W3 + self.B3)
        return x

    def forwards(self, inputs):
        result = []
        for input in inputs:
            result.append(self.forward(input))
        return result

# Ray Distributed Actor
@ray.remote
class Actor:
    def __init__(self):
        self.model = fake_NN()

    def predict(self, inputs):
```

```
        return self.model.forwards(inputs)

# Initialize Ray
ray.init(address='auto', dashboard_host="0.0.0.0")

# Task parameters
task_num = 100000
batch_size = 10

# Initialize actors
actor_num = 10
actors = [Actor.remote() for _ in range(actor_num)]

# Start timer
start_timer = time.time()

# Distribute tasks
tasks = []
for i in range(task_num // batch_size):
    inputs = [np.random.rand(1000) for _ in range(batch_size)]
    tasks.append(actors[i % actor_num].predict.remote(inputs))

# Get results
results = ray.get(tasks)

# Print time used
print(f"Time used: {time.time() - start_timer:.2f} seconds")

# Shutdown Ray
ray.shutdown()#这是一个神经网络，用的sigmod函数，多参数从而保证必然是计算密集型
```

我将介绍我们使用的指标：

1.前后差的时间：这综合体现了计算性能

2.cpu利用率：对于计算密集型任务来说，他越高说明理想情况下（忽略网络I/O）效率越高

3.Object Store Memory（对象存储内存）

- **适量的对象存储内存使用**：适量的对象存储内存使用是正常的，表明系统正在高效地管理对象。
- **过高的对象存储内存使用**：过高的对象存储内存使用可能会导致内存不足，从而触发对象溢出到磁盘，降低性能。
- **过低的对象存储内存使用**：过低的对象存储内存使用可能表明系统没有充分利用对象存储资源，存在资源浪费。

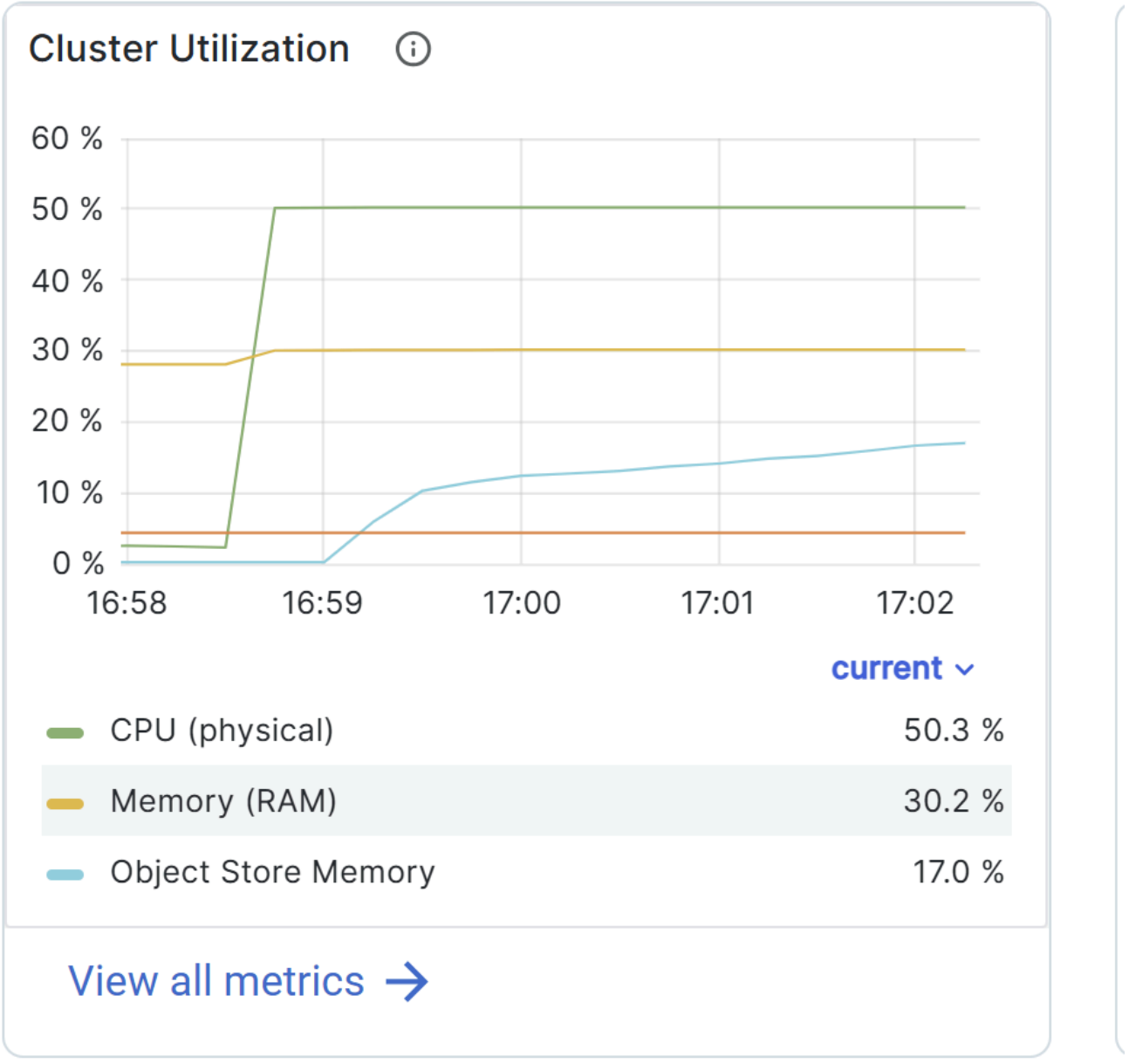
4.mem利用率：

- **适量的内存使用**：适量的内存使用是正常的，表明系统正在高效运行。

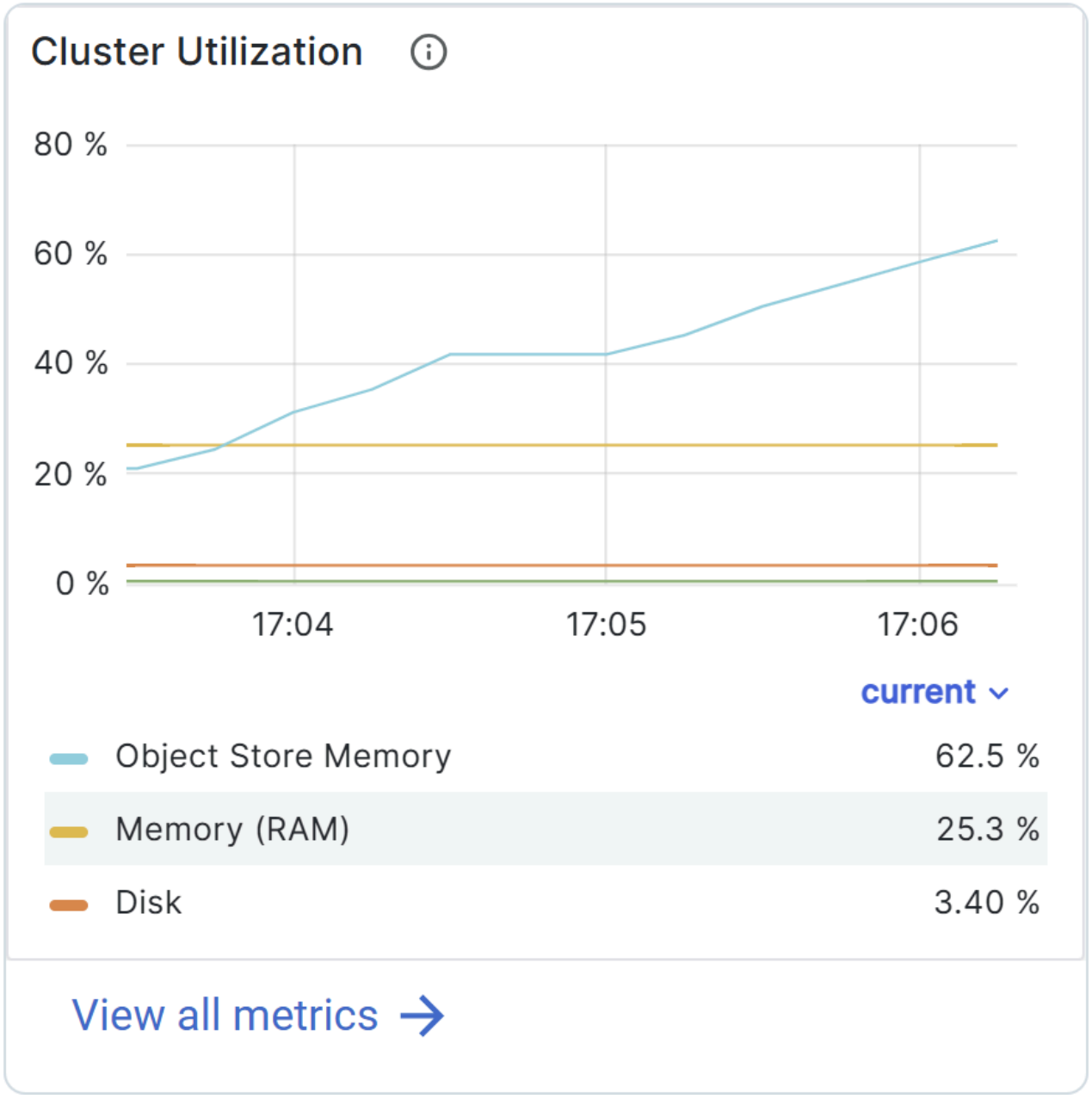
- **过高的内存使用**：如果内存使用率过高，可能会导致系统频繁进行磁盘交换（swapping），从而显著降低性能。
- **过低的内存使用**：如果内存使用率过低，可能表明系统没有充分利用可用资源，存在资源浪费

5.disk利用率：

- **适量的磁盘使用**：适量的磁盘使用是正常的，尤其是在对象存储溢出时。
- **过高的磁盘使用**：过高的磁盘使用可能会导致磁盘 I/O 成为性能瓶颈，特别是在对象存储频繁溢出到磁盘时。
- **过低的磁盘使用**：过低的磁盘使用通常不是问题，但可能表明系统没有充分利用磁盘资源



而时间花了3分44s



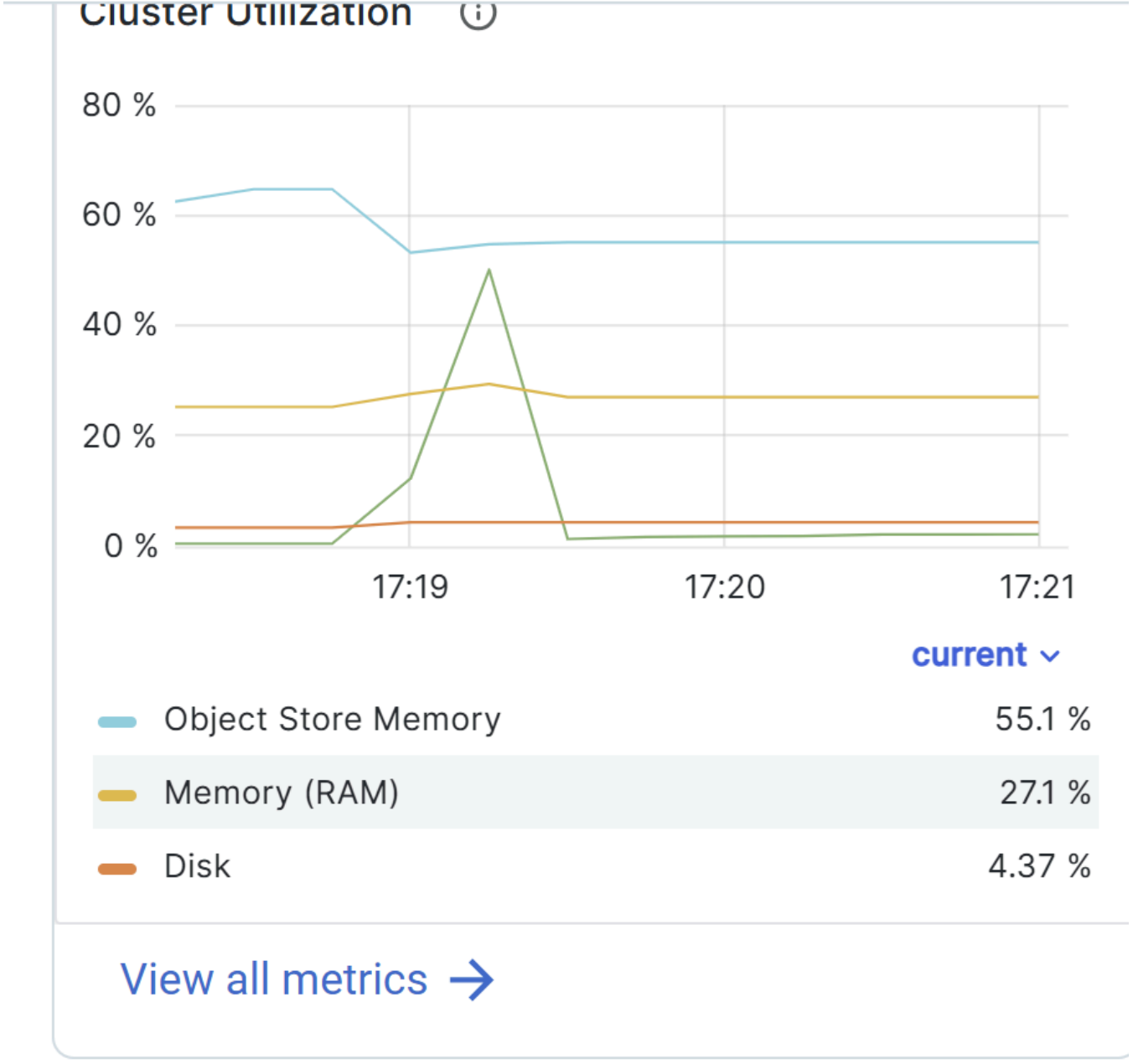
相对来说单机的时间其实要更短，59s 335ms，这跟我们的cpu核差距有关，一个20个核一个2个核，但是网络的I/O速率太慢，我们可以看到CPU的利用率在多节点大大提升，同时MEM的I/O提高了30%左右

Object Store Memory几乎不变，而内存利用率在适量的增加了5%左右，disk则显著减少，说明减少了过高的磁盘使用可能会导致磁盘 I/O 成为性能瓶颈，特别是在对象存储频繁溢出到磁盘时；最后，cpu利用率提升了一个数量级，可以见得，如果我们的任务在数据减少以后，可以减少网络数据的传递时，cpu的利用率的提高与disk交换的变少将会提高我们的效率

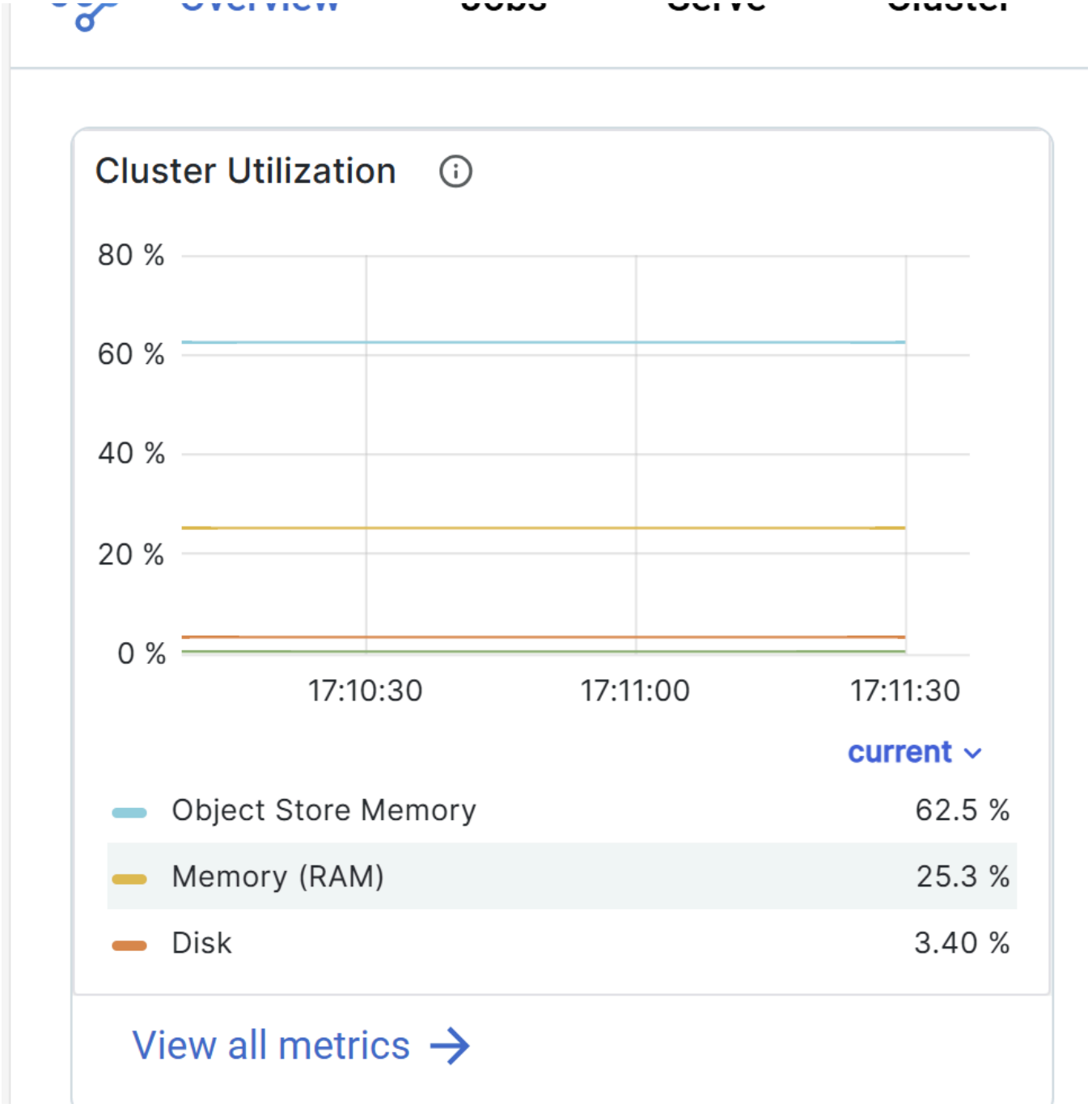
```
import ray
import numpy as np
import time

# Define the Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

```
# A Random-parameter Neural Network
class fake_NN:
    def __init__(self):
        self.W1 = np.random.rand(1000, 150)
        self.B1 = np.random.rand(150)
        self.W2 = np.random.rand(150, 150)
        self.B2 = np.random.rand(150)
        self.W3 = np.random.rand(150, 100)
        self.B3 = np.random.rand(100)

    def forward(self, input):
        x = sigmoid(input @ self.W1 + self.B1)
        x = sigmoid(x @ self.W2 + self.B2)
        x = sigmoid(x @ self.W3 + self.B3)
        return x

    def forwards(self, inputs):
        result = []
        for input in inputs:
            result.append(self.forward(input))
        return result

# Ray Distributed Actor
@ray.remote
class Actor:
    def __init__(self):
        self.model = fake_NN()

    def predict(self, inputs):
        return self.model.forwards(inputs)

# Initialize Ray
ray.init(address='auto', dashboard_host="0.0.0.0")

# Task parameters
task_num = 100000
batch_size = 10

# Initialize actors
actor_num = 10
actors = [Actor.remote() for _ in range(actor_num)]

# Start timer
start_timer = time.time()

# Distribute tasks
tasks = []
for i in range(task_num // batch_size):
    inputs = [np.random.rand(1000) for _ in range(batch_size)]
```

```
    tasks.append(actors[i % actor_num].predict.remote(inputs))

# Get results
results = ray.get(tasks)

# Print time used
print(f"Time used: {time.time() - start_timer:.2f} seconds")

# Shutdown Ray
ray.shutdown()
```

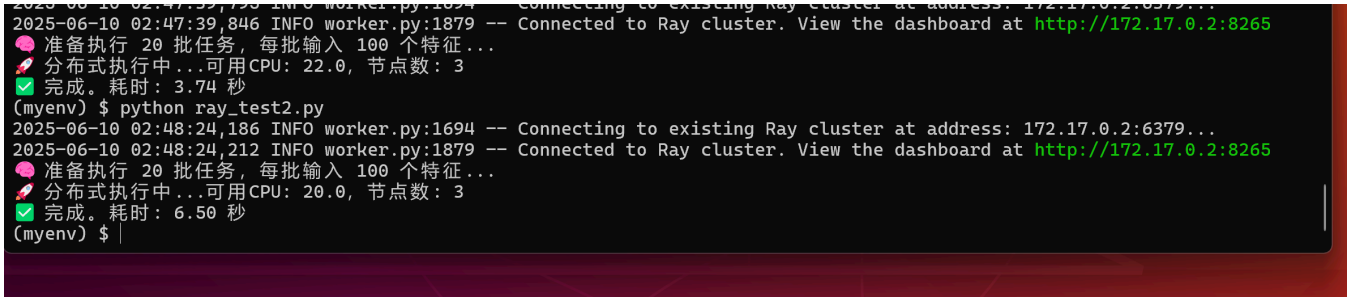


Cluster status and autoscaler



此时使用时间分别为27s 725ms，6s 962ms

前后实验中CPU利用率提高了数倍，同时disk存取降低了将近20%，而mem利用率提高，I/O效率综合也提高了近20%，但是由于我们的CPU核的不对等和节点数太少，加之网络速率I/O低，导致了总时间的提高，但是先后时间的比例（数据少时比例较低）已经说明了：当我们的网络I/O降低时，将提高我们的效率，同时当分布式具有均衡性时，我们的性能也会提高（不均衡的能力导致了堵塞），另一方面，我们可以看到，mem的提高率5%左右，disk下降5%左右，这说明我们的系统更多的利用了内存I\O，而内存I\O快与DISKI|O，说明了我们的分布式部署能一定程度上提高单机I\O的效率。

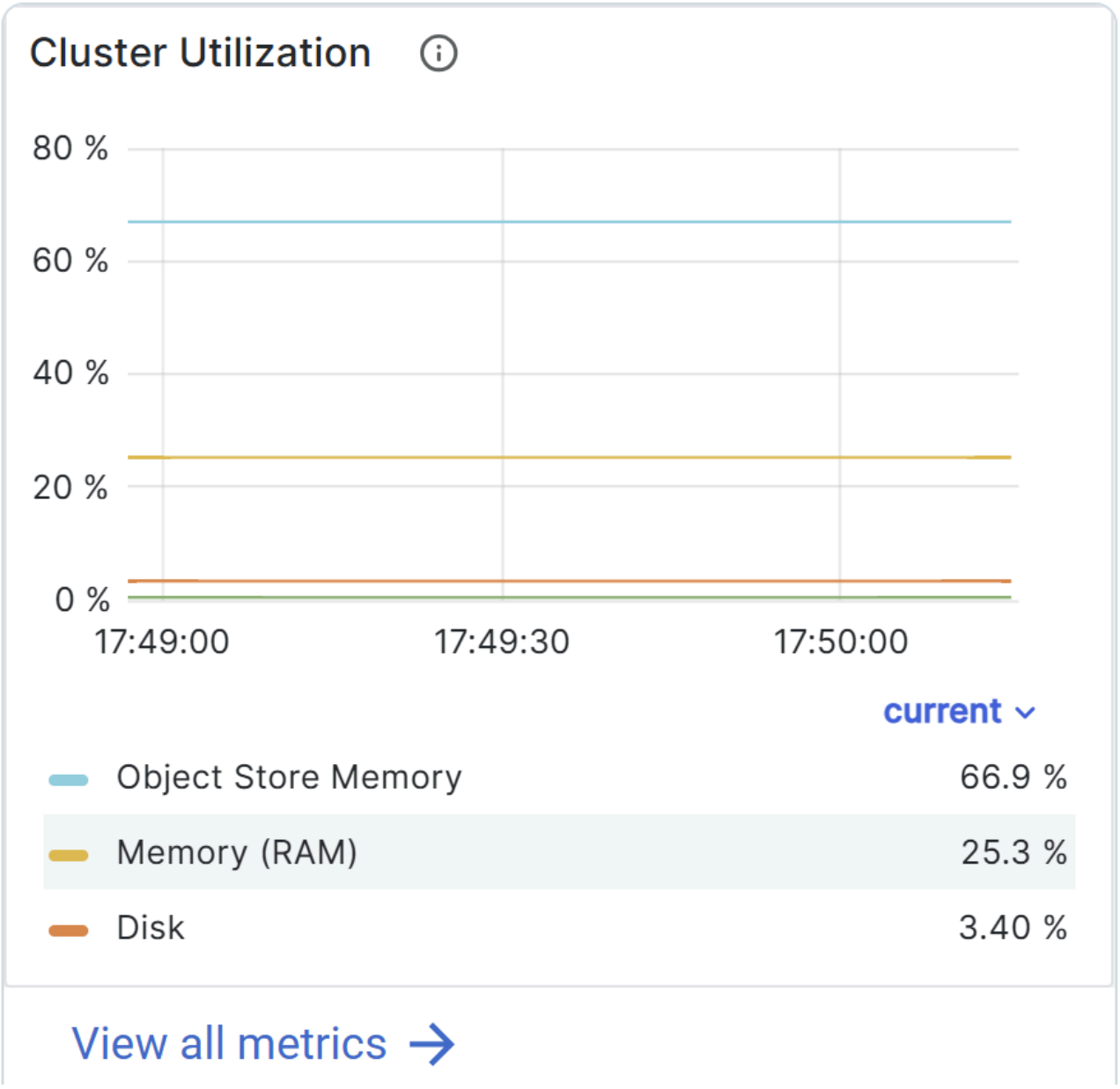


当我们把任务量减小到2个cpu可以完成时（没有了sigmod函数）

```
import time
import numpy as np
import ray

# ---- Ray远程函数，用于 Dashboard 观察 ---- #
@ray.remote
def forward_remote(x, weights):
    for w in weights:
        x = np.dot(x, w)
        x = np.maximum(x, 0)  # ReLU
    return np.sum(x)  # 做个聚合避免优化掉

# ---- 初始化网络参数 ---- #
def generate_weights(layers):
    return [np.random.randn(layers[i], layers[i + 1]) for i in range(len(layers) -
1)]

# ---- 大数据生成函数 ---- #
def generate_inputs(num_batches, input_size):
    return [np.random.randn(input_size) for _ in range(num_batches)]

# ---- 执行分布式前向传播任务 ---- #
def run_distributed(num_batches=20, input_size=100, layers=[1000, 1000, 1000, 1000]):
    print(f"🧠 准备执行 {num_batches} 批任务，每批输入 {input_size} 个特征...")

    weights = generate_weights([input_size] + layers)
    inputs = generate_inputs(num_batches, input_size)

    print(f"🚀 分布式执行中...可用CPU: {ray.available_resources().get('CPU', '未
知')}，节点数: {len(ray.nodes())}")

    start = time.time()
    futures = [forward_remote.remote(x, weights) for x in inputs]
    ray.get(futures)
    end = time.time()

    print(f"✅ 完成。耗时: {end - start:.2f} 秒")

# ---- 主函数 ---- #
```

```
if __name__ == "__main__":
    ray.init(address="auto")  # 连接已有集群（主从结构）
    run_distributed()
```

可以看到时间减少为大约是原来的一半，（这就是分布式计算带来的优势）这也验证了我之前说的需要减少网络I/O和计算资源均匀化



最后：我们认为，RAY的部署将会提高计算型密集的任务，但是要求系统资源均匀化，同时对于数据大小有要求，过大的网络I/O将会大大提高运算时间