Tetris: Memory-efficient Serverless Inference through Tensor Sharing

Edge System Reading Group @ SEU

田昊冬

Jan 30, 2023.

背景 – 无服务器 (Serverless) 架构

开发者如何部署 Web 应用程序?

- 购买运行服务器所需的物理硬件 😟

成本高昂；需要关心硬件设施的维护；...

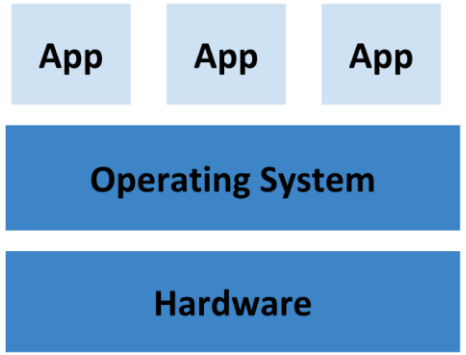
- 租用云服务器或云服务器空间 🤔

成本较低；无需关心硬件设施的维护；

应用程序之间没有隔离，产生冲突或影响性能；

云服务器性能浪费；...

传统部署 (Traditional Deployment)



IaaS (Infrastructure-as-a-Service):
Aliyun ECS, AWS EC2, Microsoft Azure VM, ...

References:
What is Serverless? – Cloudflare
Overview of Kubernetes - Kubernetes

背景 – 无服务器 (Serverless) 架构

开发者如何部署 Web 应用程序?

- 创建虚拟机 (Virtual Machine) 🤔

应用程序隔离；更好地利用服务器资源；...

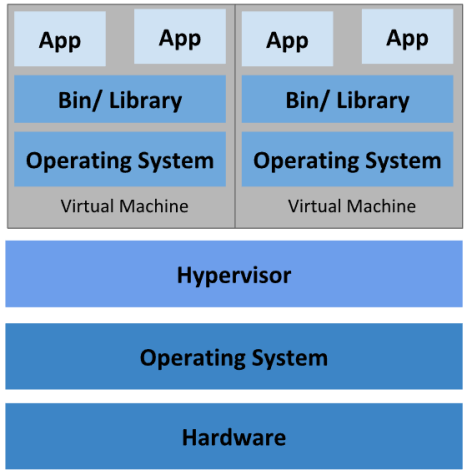
- 创建容器 (Container) 😀

相较于虚拟机，更加简单、高效；

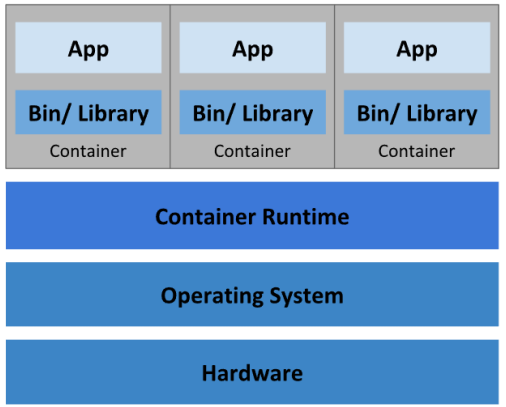
容器可以在不同服务器、不同操作系统间迁移；

Docker 等基础设施提供更多观测指标；...

虚拟化部署 (Virtualized Deployment)



容器化部署 (Container Deployment)



背景 – 无服务器 (Serverless) 架构

无服务器 (Serverless) 架构

- 一种新的 Web 应用程序部署模式
- 开发者无需关注服务器的工作细节，如操作系统、文件系统、负载均衡、动态扩容、运行时环境 (runtime environments) 等
- 开发者专注于实现业务逻辑

References:
What is Serverless? - Red Hat
Getting started with serverless for developers - AWS

背景 – 无服务器 (Serverless) 架构

FaaS - Serverless 架构的一种具体实现

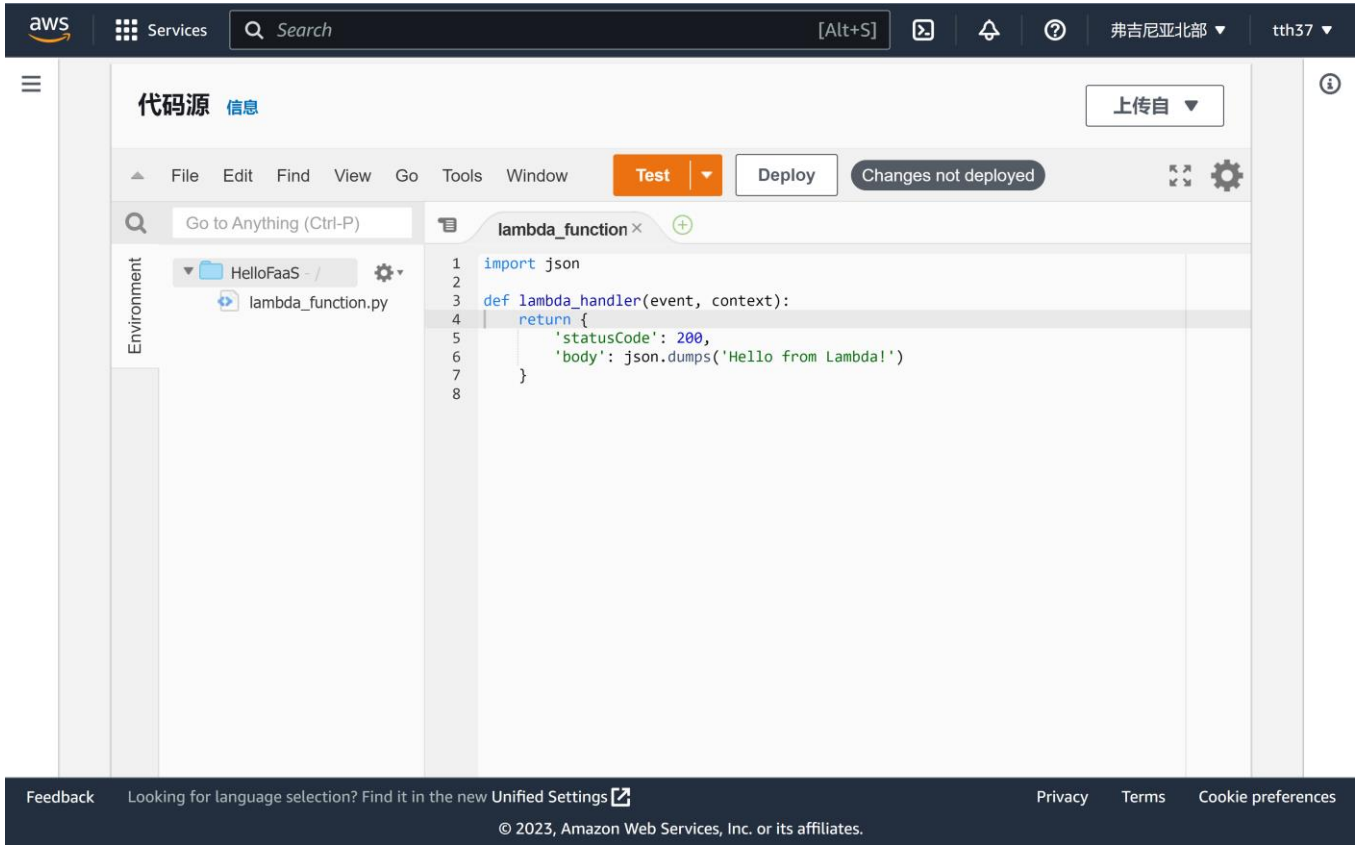
- 开发者编写小型的、执行单一功能的代码块，称为“**函数**”**(functions)**
- 函数通常只会执行很短的时间，是**无状态 (stateless)** 的
- 函数支持**自动扩容 (auto scaling)**，以应对不同并发场景
- FaaS 提供商**按需计费**，不考虑函数的闲置时间

FaaS (Function-as-a-Service):
AWS Lambda, Microsoft Azure Functions

References:
What is AWS Lambda? – AWS
Introduction to Serverless on Kubernetes - edX

背景 – 无服务器 (Serverless) 架构

示例：在 AWS Lambda 部署函数



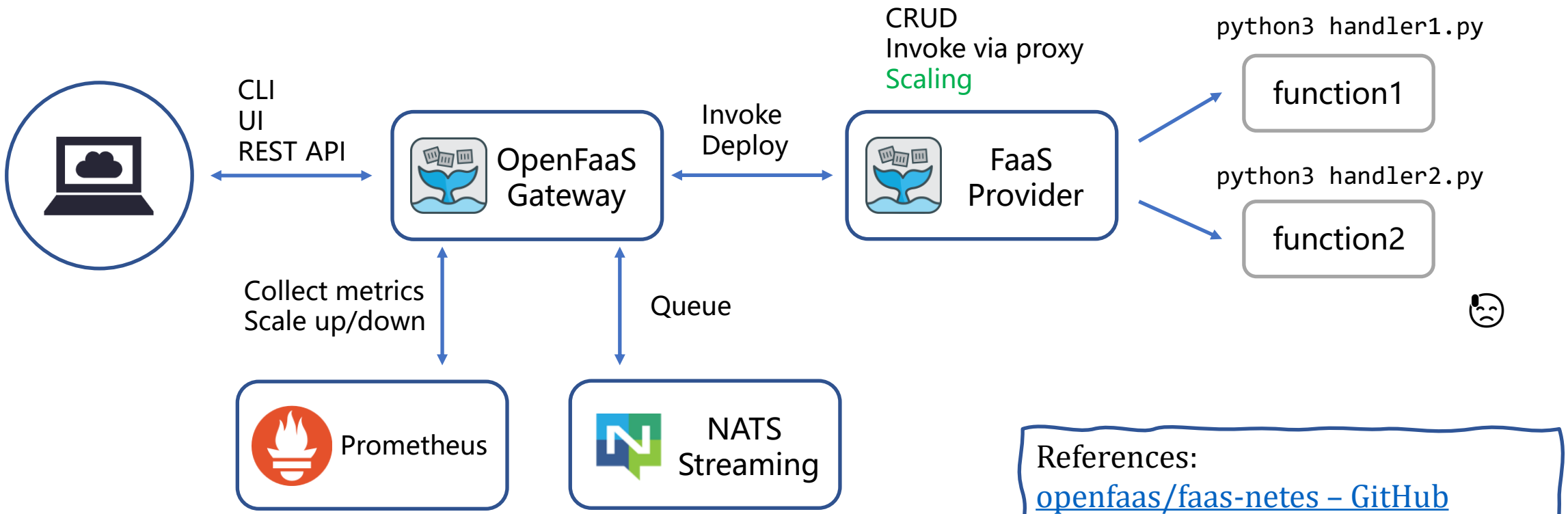
背景 – 无服务器 (Serverless) 架构

在本地（或云服务器）搭建 FaaS 平台

OPENFAAS

背景 – 无服务器 (Serverless) 架构

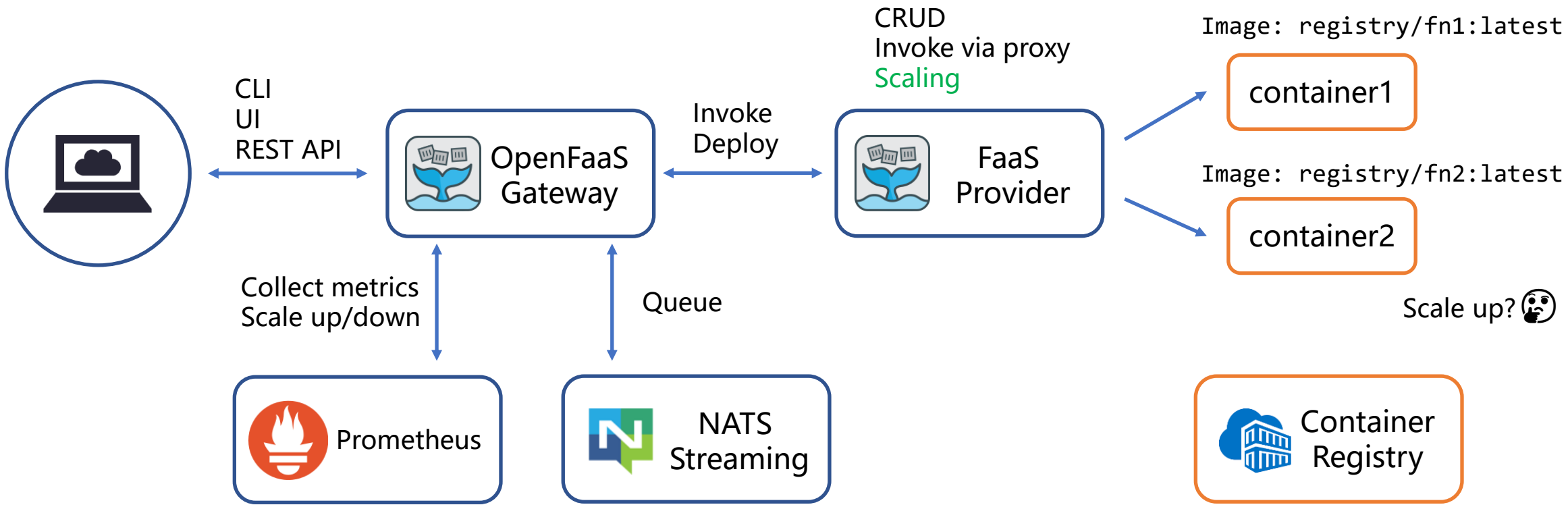
OpenFaaS 架构



References:
[openfaas/faas-netes – GitHub]
[FaaS Provider Interface - OpenFaaS]

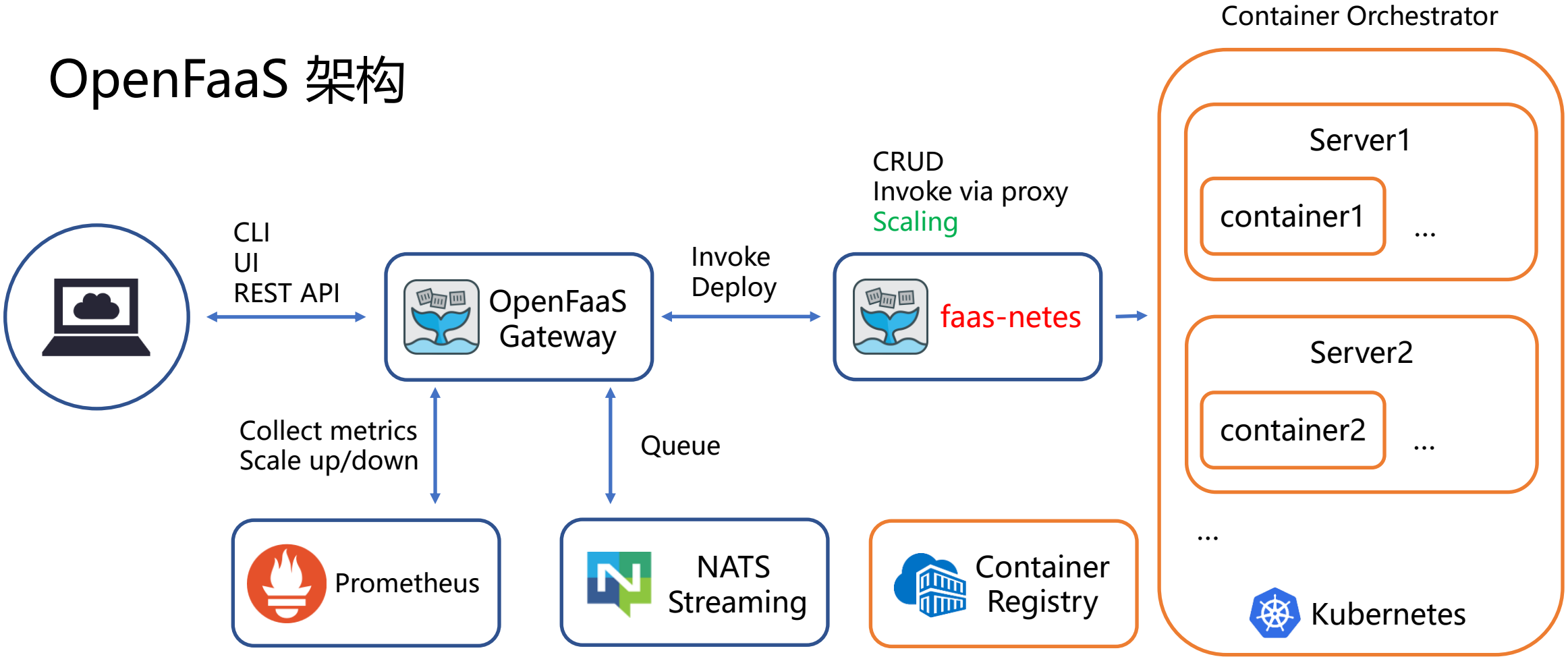
背景 – 无服务器 (Serverless) 架构

OpenFaaS 架构



背景 – 无服务器 (Serverless) 架构

OpenFaaS 架构

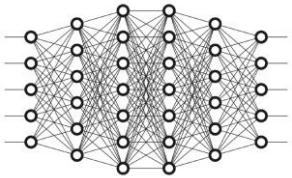


背景 – FaaS 平台上的 DL 推理

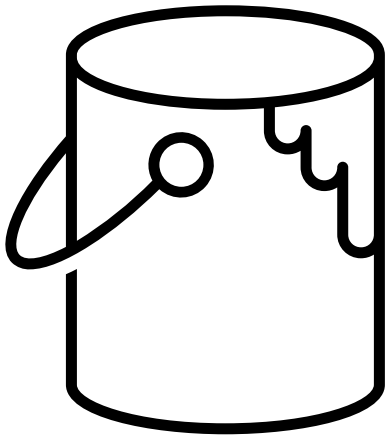
使用 DL 框架直接制作 Docker Image

DL Framework

handler.py



DL Model



References:
[Deploying PyTorch Model as a Serverless Service - serverless]

背景 – FaaS 平台上的 DL 推理

使用 TensorFlow Serving 框架

- 提供 REST API 以及 gRPC
- 提供 Batching Configuration

```
max_batch_size, num_batch_threads, …
```

- 可以封装成 Docker Image

References:
Serving Models - TensorFlow

背景 – FaaS 平台上的 DL 推理

观察：内存使用效率 (memory efficiency) 低下！

- 创建新的容器时，**加载模型参数**所需的时间占比非常高
- 推理过程中，**模型参数**占用很多内存空间
- 过多的容器会导致额外的**运行时环境**内存开销

提高并行度，增加批量大小…

- 不同模型之间使用相同的 tensor，造成 **tensor 冗余**

Multi-versioned functions, pretraining & transfer learning

系统设计 – Tensor Store

概述：同一物理机器上的 container 共享 tensor

- 所有 container 在运行时使用的 tensor 均统一存储在 Tensor Store 中
- Tensor Store 存放在虚拟存储 (Virtual Memory) 中
- Tensor Store 以 tensor 的 hash 值作为索引
- DL 框架在加载 tensor 时，如果已经在 Tensor Store 中存在，直接映射即可，无需划分额外的内存空间

系统设计 – Tensor Store

实现：/dev/shm 与 flock

- 将 *tmpfs* 挂载到 /dev/shm 目录下，并在 /dev/shm/serving_memorys 中以文件的形式存放 tensor
- 在 /path/to/serving_locks 中以文件的形式存放文件锁 (flock)，确保共享 tensor 的并发安全

tmpfs:
类 Unix 系统上，将资料存储在快速、易失性存储器中，并且具有完整文件系统的虚拟存储空间。

系统设计 – Tensor Store

核心代码：应用 RAII 机制的 FileLock 类

/tensorflow/tensorflow/core/framework/cpu_allocator_impl.cc

```
void FileLock::lock(int fd) {
    struct flock lock;

    lock.l_type = F_WRLCK; // A write lock is requested.
    lock.l_whence = SEEK_SET; // The relative offset is measured from the start of the file.
    lock.l_start = 0; // Defines the relative offset in bytes, measured from the starting point in the l_whence field.
    lock.l_len = 0; // (write lock entire file here) Specifies the number of consecutive bytes to be locked.

    Fcntl(fd, F_SETLKW, &lock); // block until lock acquired.
}
```

系统设计 – Tensor Store

核心代码：使用 mmap 将 /dev/shm 中的 tensor 映射到内存

/tensorflow/tensorflow/core/framework/cpu_allocator_impl.cc

```
void* mmap_shm_open(std::string mmap_file) {
    int fd = open(mmap_file.c_str(), O_RDWR);
    if (fd == -1) { err_sys("open error for mmap file"); }
    struct stat statbuf;
    if (fstat(fd, &statbuf) != 0) { err_sys("fstat error for mmap file"); }
    void *ptr = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (ptr == MAP_FAILED) { err_sys("mmap error for mmap file"); }
    if (close(fd) != 0) { err_sys("close failed for mmap file"); }
    return ptr;
}
```

系统设计 – Tensor Store

核心代码：使用 mmap_allocator 重写 RestoreOp

/tensorflow/tensorflow/core/kernel/save_restore_tensor.cc

```
bool mem_not_exist = true;
TF_RETURN_IF_ERROR(
    context->allocate_output_mmap(idx, restored_full_shape, &restored_tensor, mmap_id, mem_not_exist));
if(mem_not_exist) {
    TF_RETURN_IF_ERROR(reader->Lookup(tensor_name, restored_tensor));
}

// TF_RETURN_IF_ERROR(
//     context->allocate_output(idx, restored_full_shape, &restored_tensor));
// TF_RETURN_IF_ERROR(reader->Lookup(tensor_name, restored_tensor));
```

系统设计 – Tensor Store

核心代码：随机化加载 tensor 的顺序以避免文件锁冲突

/tensorflow/tensorflow/core/kernel/save_restore_tensor.cc

```
std::vector<size_t> sorted_name_idx(tensor_names_flat.size());
std::iota(sorted_name_idx.begin(), sorted_name_idx.end(), 0);
std::srand(unsigned(std::time(0)));
std::random_shuffle(sorted_name_idx.begin(), sorted_name_idx.end(), myrandom);
```

系统设计 – Auto Scaling

概述：自定义函数实例的配置信息

- 分配的 CPU 核心数量
- 分配的内存空间大小
- 最大批量大小
- 线程数量

Batch scheduling parameters provided by Tensorflow Serving:
max_batch_size, batch_timeout_micros, num_batch_threads, max_enqueued_batches

系统设计 – Auto Scaling

概述：不同配置下函数示例时延分析器 (profiler)

- $latency(c, m, b, p)$
- $c \in C, m \in M, b \in B = \{1,2,4\}, p \in P = \{1,2,3,4\}$
- 仅考虑小批量和较低的并行度，以减小分析器和决策算法负担

References:
Performance Tuning - Tensorflow Serving

系统设计 – Auto Scaling

概述：决策启动新的函数实例的配置信息

- 已知每秒请求数量 RPS，服务时延目标 SLO，分析数据 $O = \{< c, m, b, p, l >\}$
- 最小化新的函数实例的内存占用总量
- 满足时延目标限制

- 令函数实例的吞吐量 $throughput = bp/l$，满足 $\sum throughput \geq RPS$
- 令单位内存所能提供的吞吐量 $tpm = bp/lm$，优先考虑 tpm 较小的实例
- 如果考虑的实例大致满足时延限制，则启动该实例

系统设计 – Auto Scaling

概述：选取 tensor 更相似的服务器优先部署

- 服务器**硬件资源**仍然满足函数实例的配置要求
- 在此基础上，优先部署在 Tensor Store 与本身模型更相似的服务器上

系统设计 – Auto Scaling

实现：DTS 贪心算法

Algorithm 1: DTS Algorithm

Input:

Requests R; profile $O = \{< c, m, b, p, l >\}$; t_{slo};

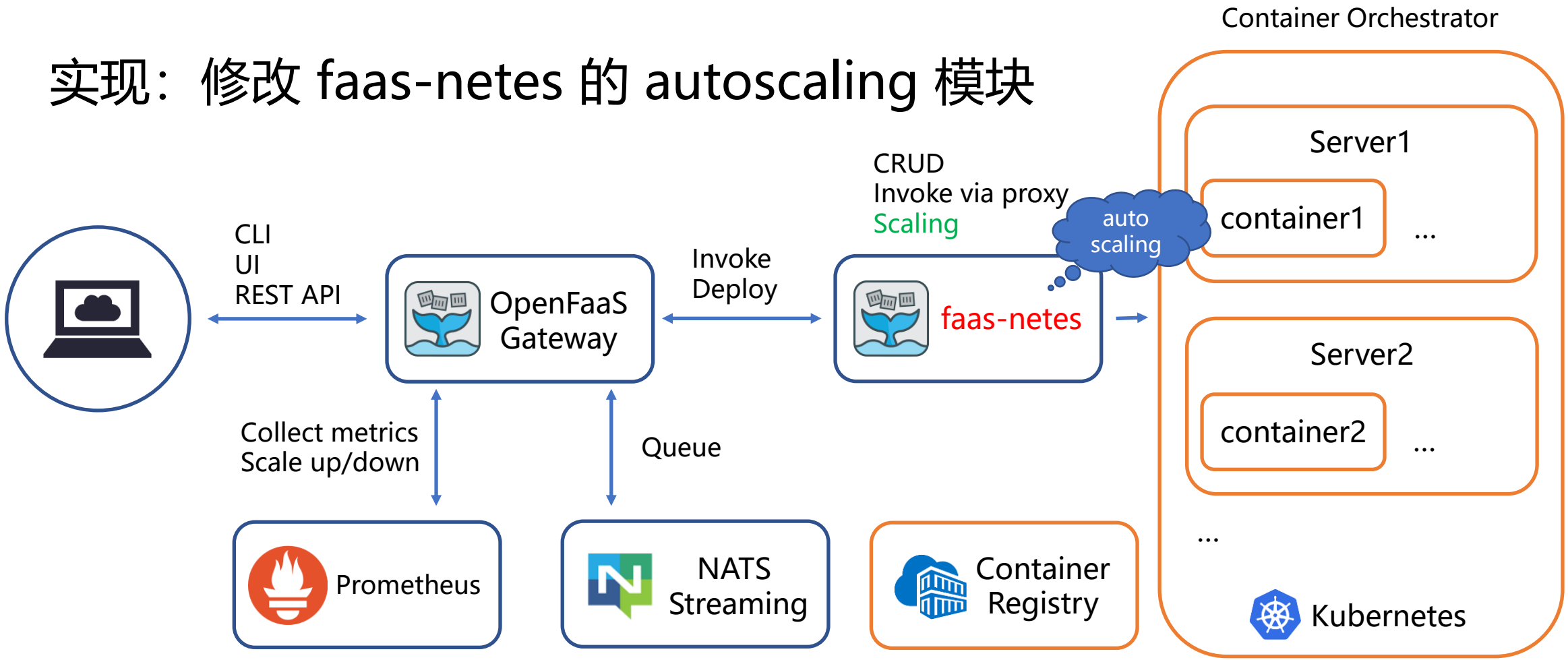
Output:

S: the set of selected instance configurations;

```
1  S = ∅;
2  sort O in descending order of (b_i p_i)/(l_i m_i), ∀i ∈ [1..n];
3  while R > 0 do
4      for each configuration o_i ∈ O do
5          if b_i = 1 ∧ t_exec(c_i, b_i, p_i) > t_slo then
6              continue;
7          if b_i > 1 ∧ t_exec(c_i, b_i, p_i) > t_slo/2 then
8              continue;
9          R ← R − (b_i p_i)/l_i;
10         S ← S ∪ {o_i};
11         break;
```

系统设计 – Auto Scaling

实现：修改 faas-netes 的 autoscaling 模块



系统设计 – Auto Scaling

核心代码：实现 DTS 贪心算法

/openfaas/faas-netes/gpu/controller/scheduler.go

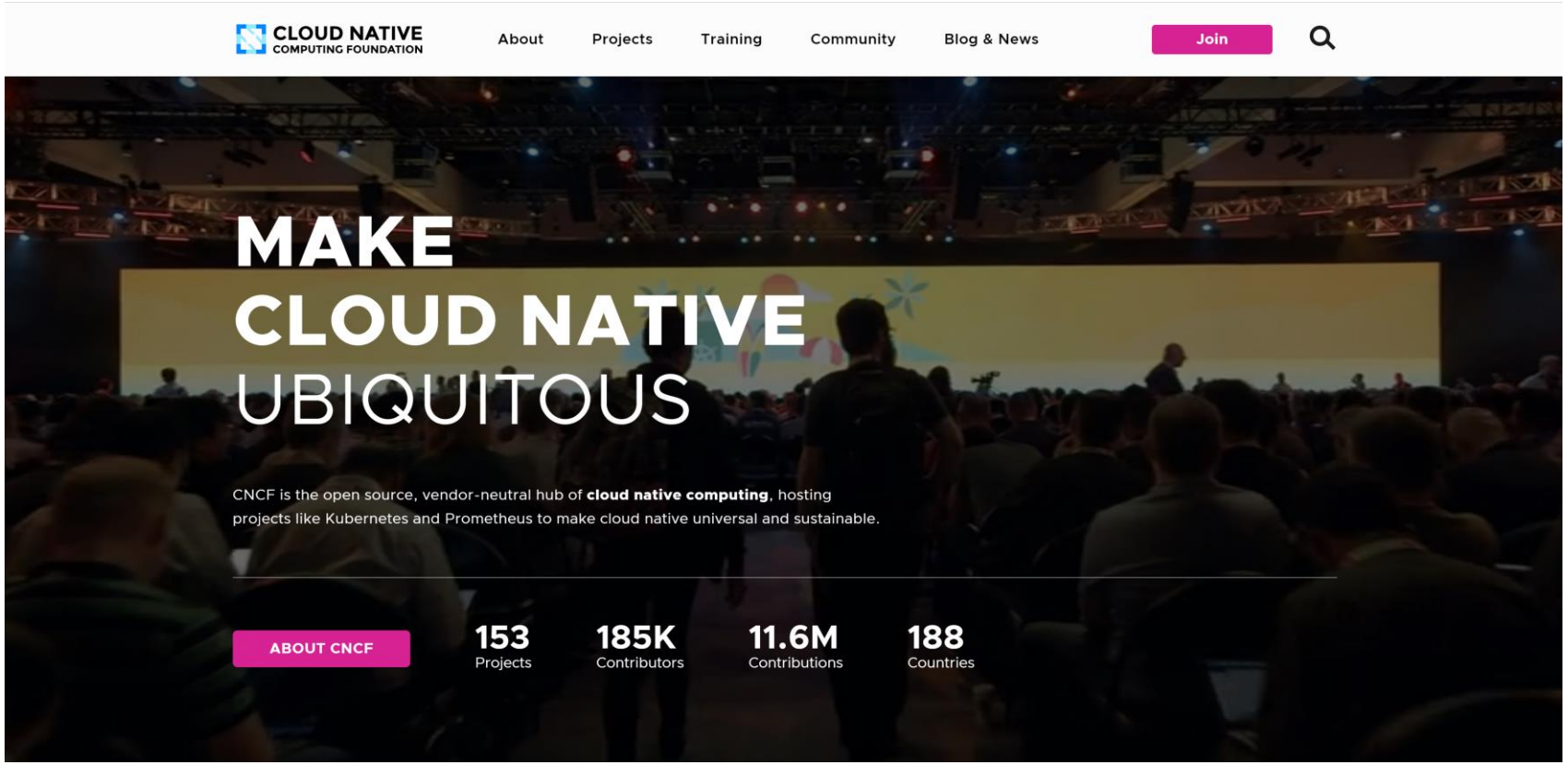
```
func CB_ScaleUp(funcName string, namespace string, latencySLO float64, reqArrivalRate int32, ...
    IC_Sort(&bcs)
    for {
        if residualReq <= 0 {
            break
        }
        if residualReq > 0 {
            resourcesConfigs, errInfer := inferResourceConfigs(funcName, &bcs, latencySLO, residualReq)
            ...
```

系统设计 – Memory Reclaiming

概述：当 tensor 的引用数量为 0 时，回收内存空间

- 可选的缓存策略：*Keep-alive window*, *Least Recent Used (LRU)*

讨论 – 云原生 (Cloud Native) 计算



总结

- Tensor Store 在针对 DL 推理的应用场景下表现出色
- Auto Scaling 的实现比较粗糙
- 对于具有不同吞吐率的函数实例，是否有针对性的负载均衡？

- Tetris 开源代码：https://github.com/JelixLi/Tetris
- 在本地部署 OpenFaaS 平台：https://zhuanlan.zhihu.com/p/601688767