

# SAND: Towards High- Performance Serverless Computing

Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein,  
Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt,  
Nokia Bell Labs



# Serverless Computing

---

无服务器计算已经成为一种新的云计算模型，其中应用程序由可以单独管理和执行的各个功能组成。

Serverless是一种基于互联网的技术架构理念，应用逻辑并非全部在服务端实现，而是采用FaaS（Function as a Service）架构，通过功能组合来实现应用程序逻辑。

FaaS架构将动态管理云资源的职责转移到提供者，允许开发人员只关注其应用程序逻辑。它还为云提供商创造了提高其基础架构资源效率的机会。

现有的商业无服务产品：

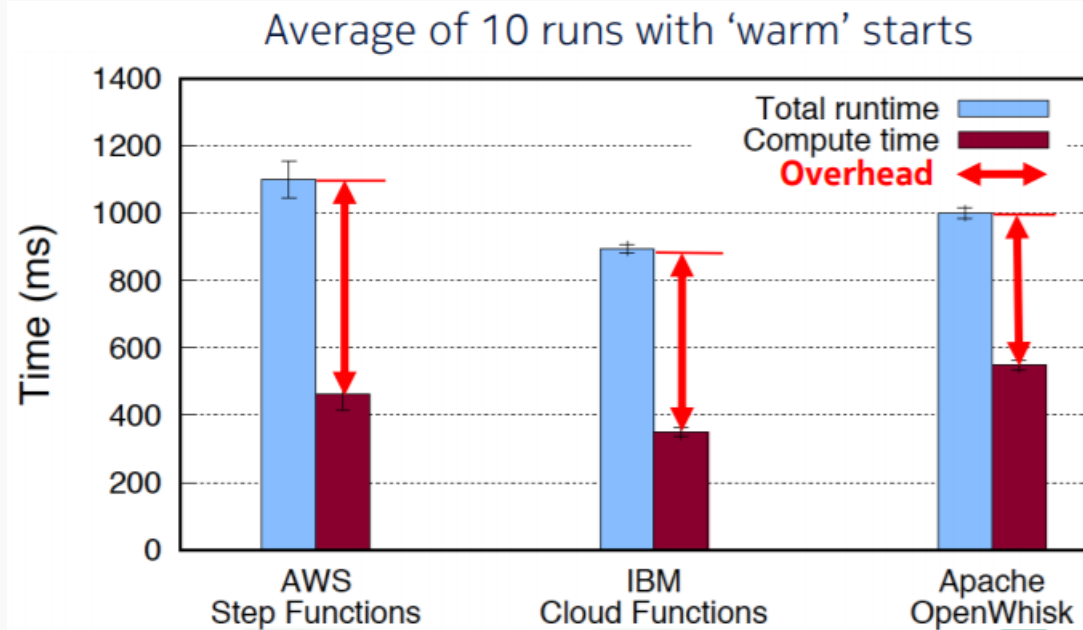
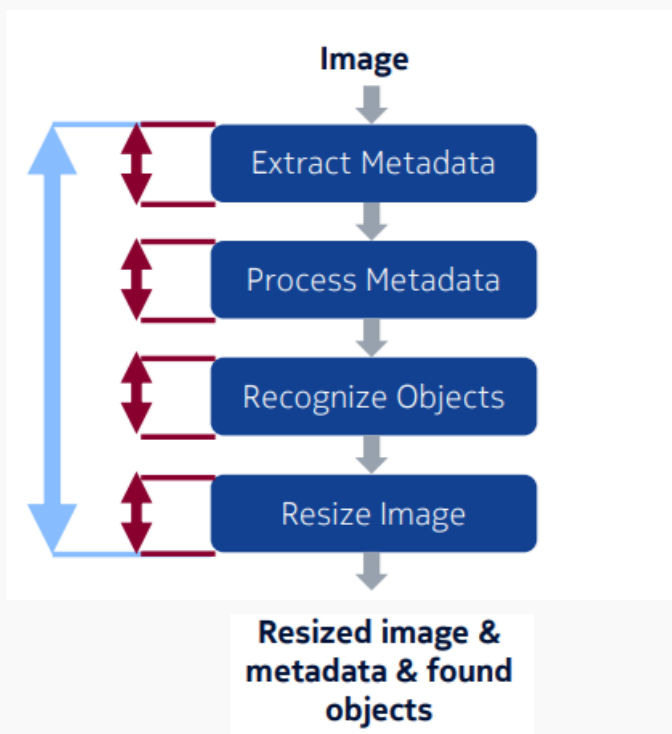
Amazon Lambda ， IBM Cloud Functions ， Microsoft Azure Functions 和 Google Cloud Functions



# Serverless Computing

虽然现有的无服务器平台适用于简单的应用程序，但由于它们的开销，它们不适合更复杂的服务，特别是当应用程序逻辑需要执行多个功能时。

在AWS、IBM和OpenWAKE上运行图像处理管道



# SAND

---

现有的无服务器平台通常在单独的容器中隔离和执行功能，并且不利用功能之间的交互来提高性能。这些做法导致功能执行的高启动延迟和低效的资源使用。

**SAND**是一个新颖的高性能无服务器平台。

目标是实现低延迟和高资源利用效率

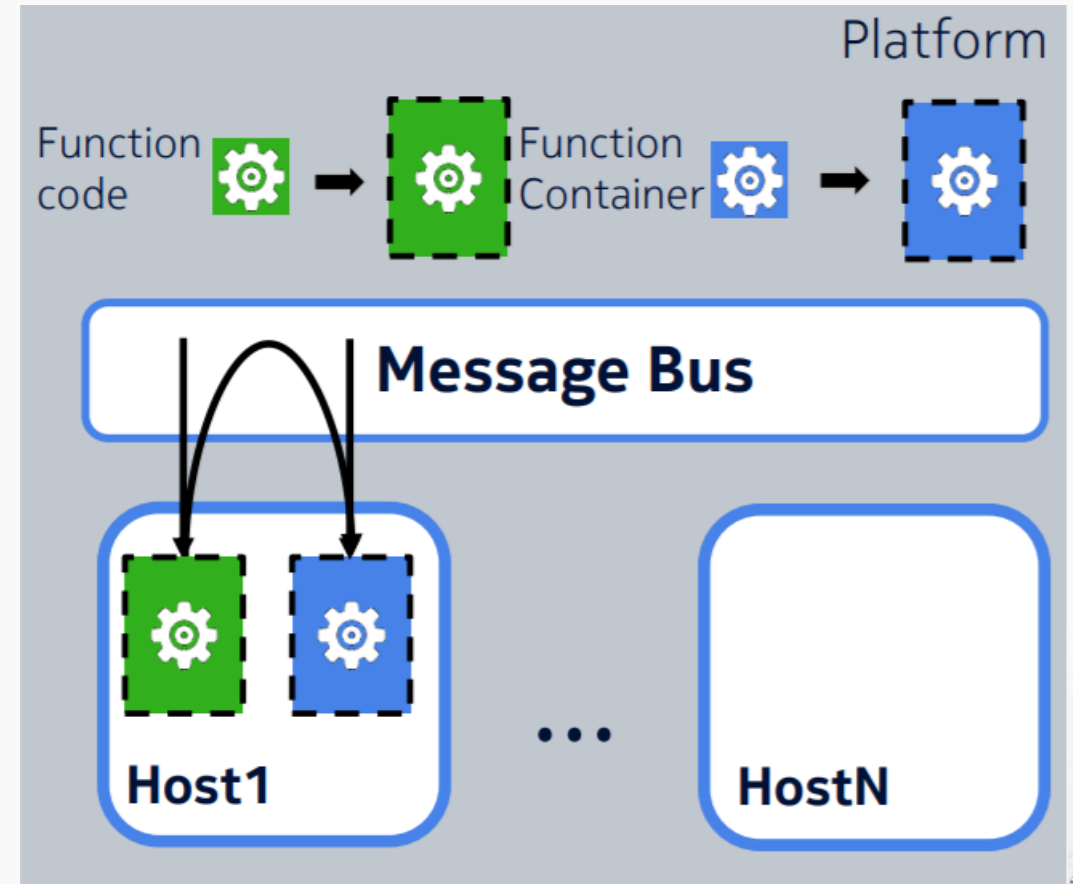
# Overview of Existing Platforms

Functions are isolated with containers

Containers are deployed where resources are available

Containers handle events and stay deployed until a timeout

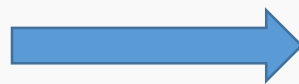
Functions interact via a distributed message bus



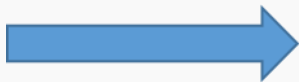
# Implications of Common Practices

## 功能执行与并行：

- 1.对于每个功能启动一个新容器
- 2.重新使用已使用的容器
- 3.并行：“冷”启动或排队等待空闲容器



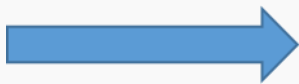
较长的启动延迟



占用资源，放宽了  
原始隔离保证

## 功能交互：

通过分布式消息总线



较长的功能交  
互延迟

# SAND

---

## Application-level Sandboxing

两个级别的隔离：

- 1) 不同应用程序之间的隔离
- 2) 同一应用程序的功能之间的隔离

不同的应用程序需要较强的隔离；同一应用程序的功能可能不需要如此强大的隔离，从而允许我们提高应用程序的性能。

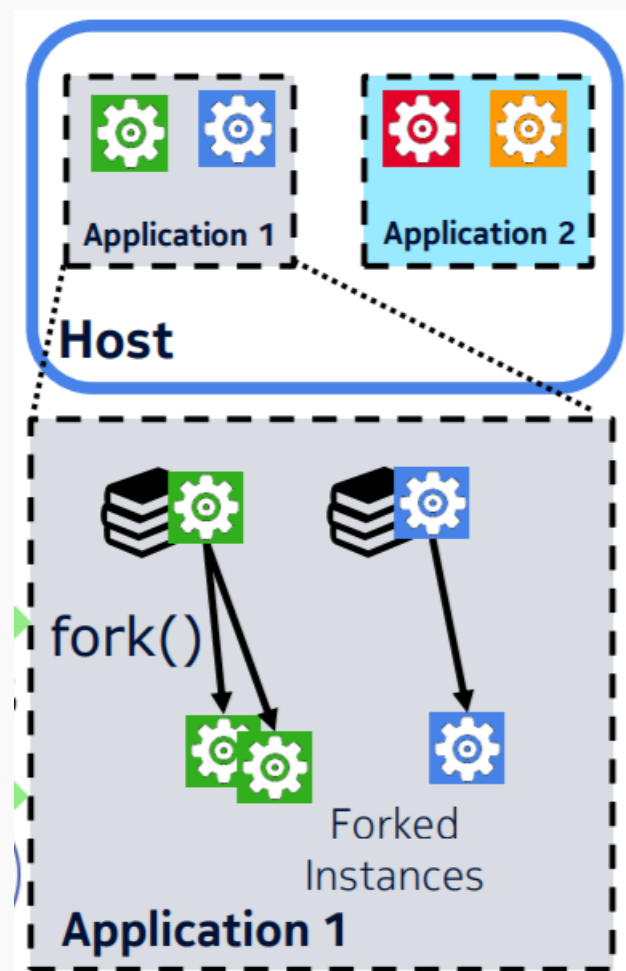
# SAND

## Application-level Sandboxing

- 1.将应用程序放到一个单独的容器中
- 2.将函数运行作为一个独立的进程放在同一个容器中
- 3.生成新进程来处理新事件

## Advantage

- 1.启动延迟较短
- 2.多个函数共享的库只需加载一次
- 3.内存占用小





# SAND

---

## Hierarchical Message Queuing

基本思想是为彼此交互的功能（例如，同一应用程序的功能）创建快捷方式。

- 1.每个主机上都有一个本地消息总线
- 2.如果两个功能在同一主机上运行，则每个主机上的本地消息总线用于将事件消息从一个功能传递到另一个功能。
- 3.用全局消息总线协调本地消息总线

## Advantage

- 1.较低的交互延迟
- 2.故障恢复<sup>1</sup>

# SAND System Design

## System Components

在SAND中，应用程序的函数称为**grain**

**Workflow** 定义了**grain**之间的交互关系

应用程序的容器为**sandbox**，当沙箱托管特定的**grain**时，它会运行一个专门的操作系统进程，称为**grain worker**。

**Grain worker**加载相关的**grain**代码及其库，订阅主机本地消息总线中的**grain**队列，并等待事件消息。3

收到相关的事件消息后，**grain worker**会自行创建一个处理事件消息的**grain instance**

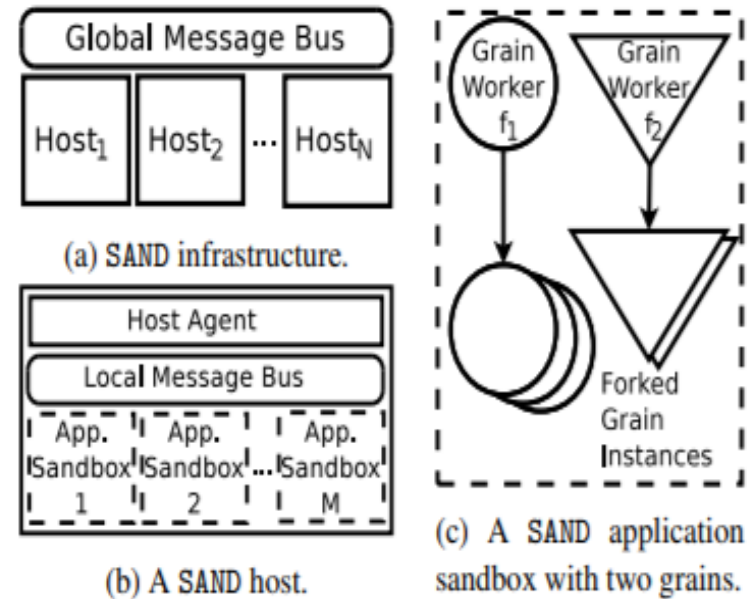


Figure 3: High-level architecture of SAND.

# SAND System Design

## System Components

在本地消息总线中，为在此主机上运行的每个**grain**创建单独的消息队列。

在全局消息总线中，为整个基础架构中托管的每个**grain**创建单独的消息队列。每个这样的消息队列被分区以增加并行性，使得每个分区可以被分配给运行相关联的**grain**的不同主机。

主机代理负责本地和全局消息总线之间的协调；启动**sandbox**；产生**grain worker**；订阅全局消息总线；跟踪**grain instance**的进度。

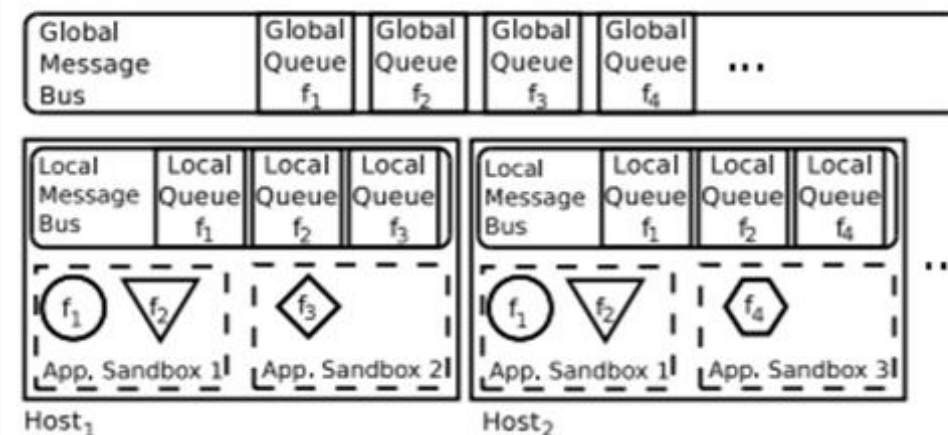


Figure 2: SAND's key building blocks: application sandboxing and hierarchical message queuing.

# SAND System Design

- 0.存在对Grain1的用户请求，全局总线根据负载平衡策略将其放入GQ1的分区GQ1,1
- 1.主机代理检索到此事件消息
- 2.主机代理将消息发布到本地总线中的LQ1队列
- 3.GW1检索LQ1生成grain instance来处理消息
- 4.处理消息
- 5a. Grain instance将此事件消息直接发布到Grain2的关联本地队列LQ2
- 5b. Grain instance将事件消息的副本发布到Hostx上的主机代理的本地队列LQHA

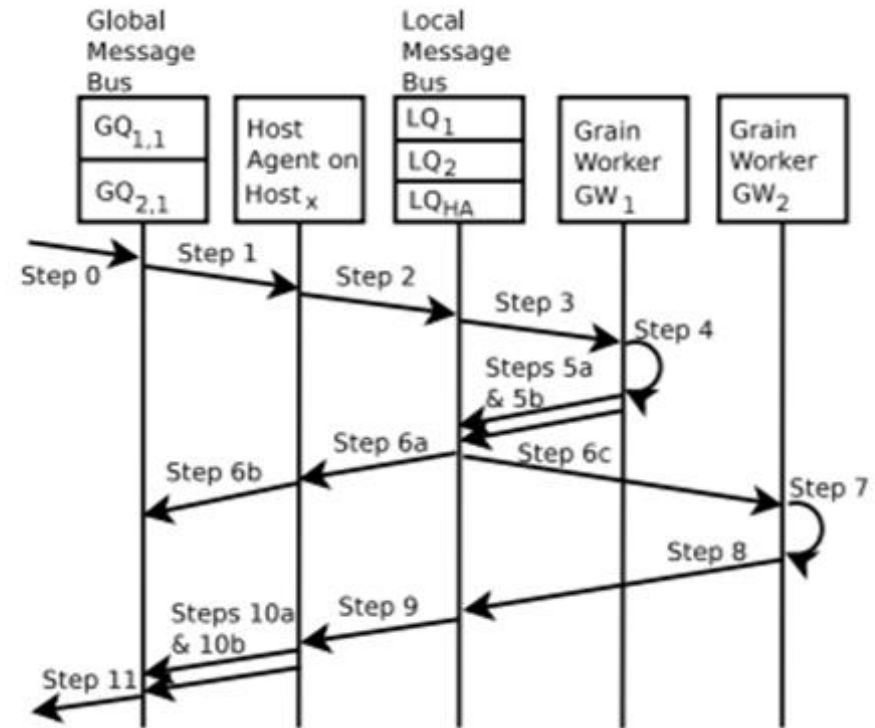


Figure 4: Handling of a user request to a simple application that consists of two grains in a workflow.

# SAND System Design

6a&6b.主机代理检索消息并将其作为备份发布到GQ2,1，条件标志为‘processing’。

6c. GW2检索LQ2中的事件消息生成grain instance

7.处理事件消息

8. GW2向主机代理LQHA的本地队列生成新的事件消息

9&10a.主机代理检索新的事件消息并直接将其发布到全局消息总线

10b.Host代理更新本地产生的事件消息的条件标志为‘ finished’

11.将响应发给用户

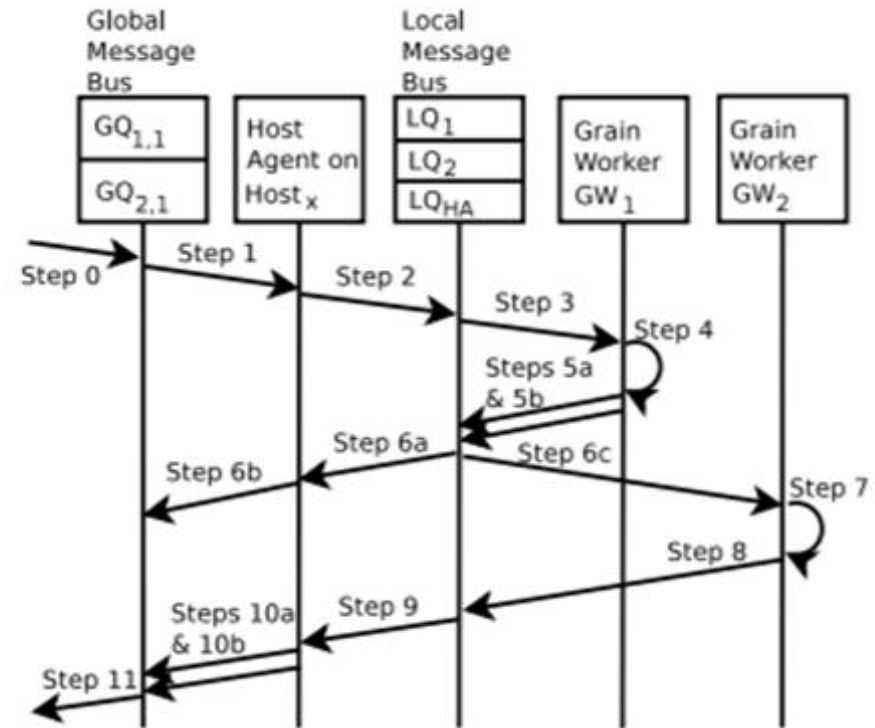


Figure 4: Handling of a user request to a simple application that consists of two grains in a workflow.




# SAND System Design

---

## Additional System Components

前端服务器是开发人员部署其应用程序以及管理**grain**和**work flow**的接口。它充当**SAND**上任何应用程序的入口点。

**Grains**可以通过在事件消息中传递引用来共享数据，而不是传递数据本身。本地数据层在每个主机上运行，类似于本地消息总线，并且可以通过内存中的键值存储快速访问在本地**grain**之间共享的数据。全局数据层是跨云基础架构运行的分布式数据存储，类似于全局消息总线。本地和全局数据层之间的协调类似于本地和全局消息总线之间的协调



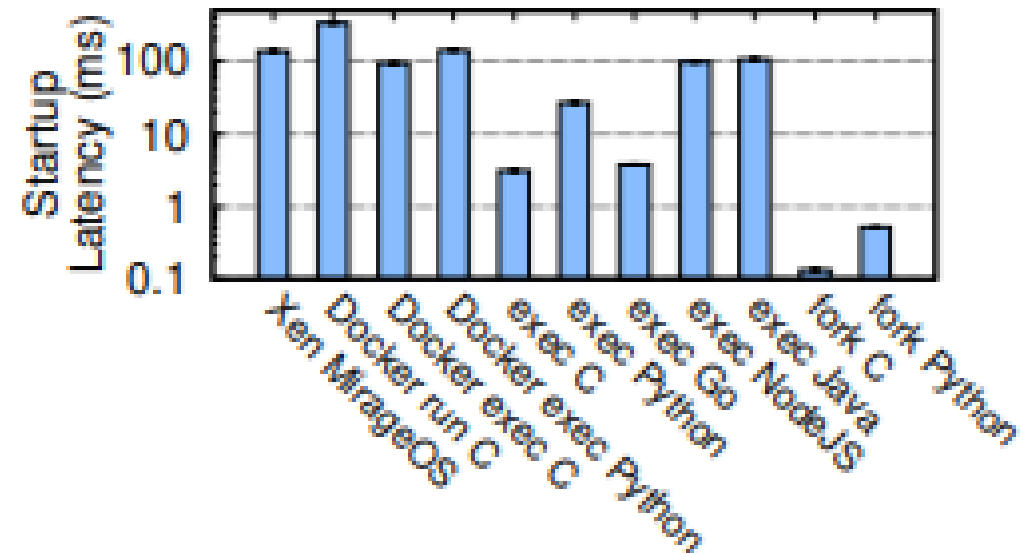


# Evaluation

## Sandboxing and Startup Latency

Intel Xeon E5-2609 with 4 cores at 2.40GHz and 32GB RAM running CentOS 7.4 (kernel 4.9.63).

测试了 Docker (CE-17.11和runc 1.0.0) 和unikernel (Xen 4.8.1和MirageOS 2.9.1) 以及生成进程在各语言下的延迟。



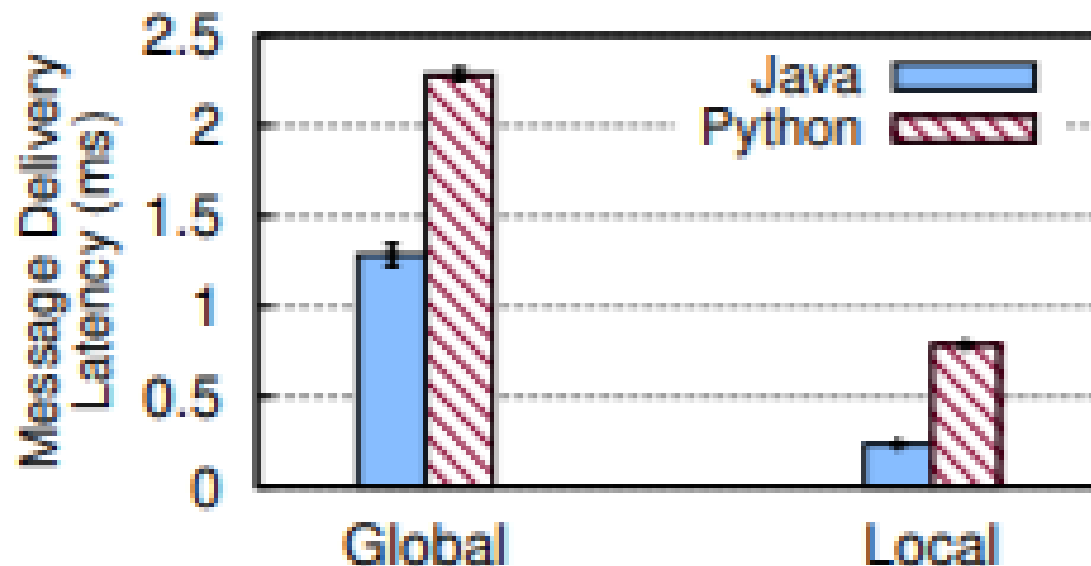
(a) Function startup latencies.

# Evaluation

## Hierarchical Message Bus

在同一主机上创建了两个进程，这些进程在无负载条件下以生产者 - 消费者方式进行通信。使用Python和Java客户端，测量了通过全局消息总线（Kafka 0.10.1.0, 3个主机，3个副本，默认设置）和本地消息总线传递的消息的延迟。

Python客户端快2.90倍；  
Java客户端快5.42倍



(b) Message delivery latencies.



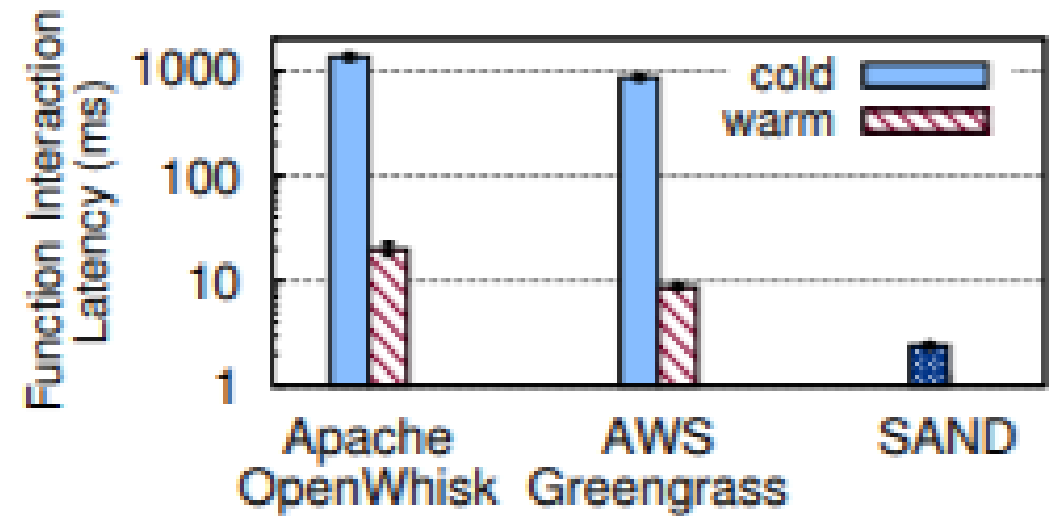
# Evaluation

## Function Interaction Latency

在OpenWhisk 中使用了一个Action Sequence，匹配了Greengrass 中的MQTT topic subscriptions和SAND的 workflow description

Warm: 8.32倍和3.64倍

Cold: 562倍和358倍



(c) Python function interaction latencies.

# Evaluation

---

## Memory Footprint

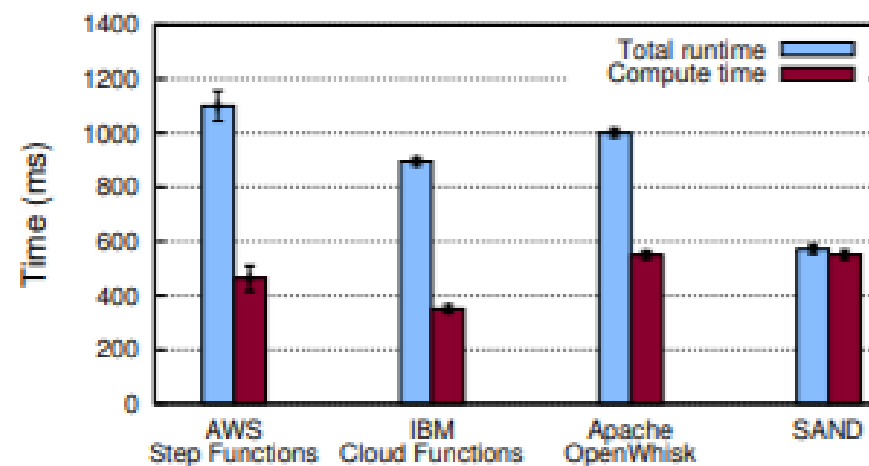
我们对单个Python函数进行了50次并发调用。我们确保所有执行都是并行的，并通过docker stats和ps命令测量每个平台的内存使用情况。


我们发现OpenWhisk和Greengrass都显示内存占用量随着并发执行数量的线性增加。每次调用分别在OpenWhisk和Greengrass中增加了大约14.61MB和13.96MB的内存占用。在SAND中，每个执行仅在grain worker消耗的16.35MB之上增加1.1MB。

# Evaluation

## Image Processing Pipeline

每个函数都记录了执行开始和结束时的时间戳，我们用它来生成实际的计算时间。总时间和计算时间之间的差异给了每个平台的开销。图像总是从与每个函数调用相关联的临时本地存储中读取。我们在AWS Step Functions，IBM Cloud Functions，带有Action Sequences的Apache OpenWhisk和SAND上运行了管道。





谢谢

THANK YOU