



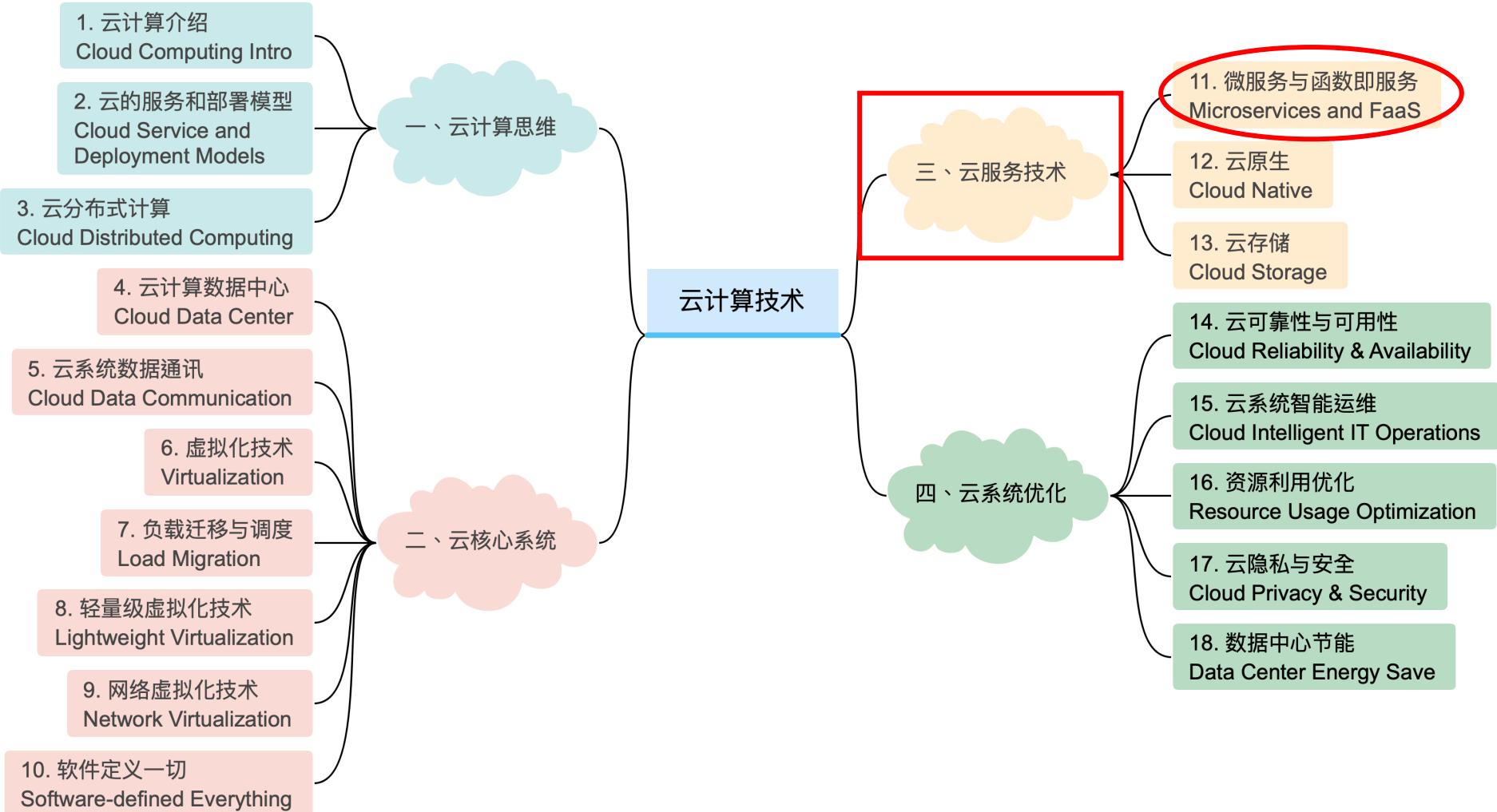
Lecture 11: 微服务与函数即服务

SSE316: 云计算技术
Cloud Computing Technologies

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn



Today's topics

□ 微服务的概念

□ 微服务架构评估

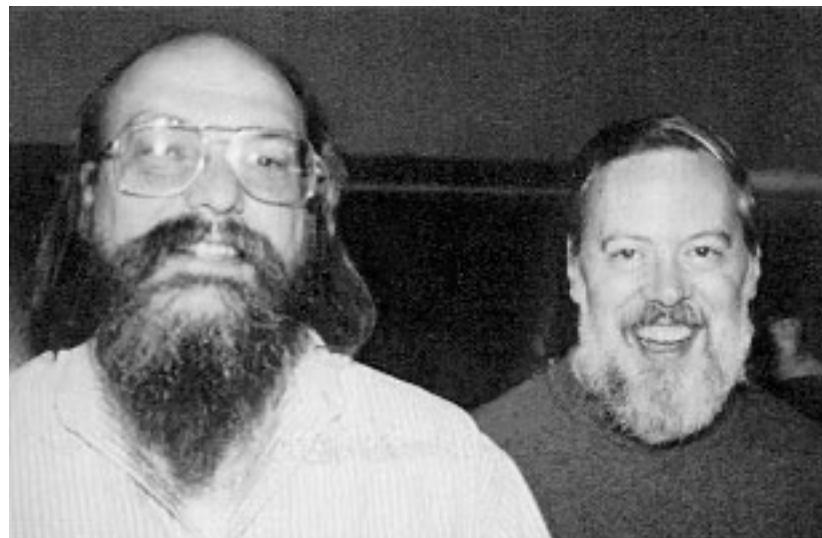
□ 无服务器计算

微服务的概念

Unix 设计哲学

□ The UNIX philosophy: (from Wikipedia [1])

- Emphasizes building **simple, short, clear, modular, and extensible** code that can be easily maintained and repurposed by developers **other than its creators**
- Favors composability as opposed to **monolithic design**
- “Do one thing and do it well”



Ken Thompson 和 Dennis Ritchie,
The Unix philosophy 的主要倡导者

单体软件架构 Monolithic architecture

口单体软件架构

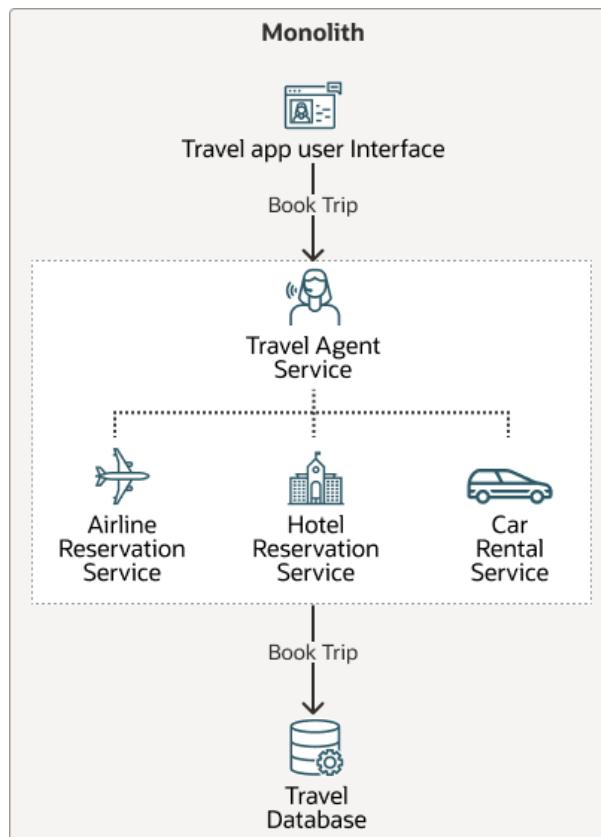
- 一种传统的软件架构模式
- 所有功能模块和组件都集成在**单个应用程序或单个可执行文件中**

口适用场景

- 小型项目
- 快速原型开发
- 资源有限的场景

口单体软件架构的优点

- 简单性
- 一致性
- 性能 (通信/数据交换)
- ...



单体软件架构的缺点

□ 可扩展性问题 (Scalability issues)

- 在单体架构中，所有功能都在一个应用中，这可能会导致应用的大小和复杂性随着时间的推移而增加

□ 维护困难 (Difficulty in maintenance)

- 更改或更新应用的某一部分可能会影响到整个系统，使得维护变得困难

□ 部署风险 (Deployment risks)

- 部署新的功能或修复 bug 通常需要重新部署整个应用，增加部署风险

□ 技术债 (Technical debt)

- 随着时间的推移，单体应用可能会积累大量的技术债务

□ 团队协作复杂性 (Complexity in team collaboration)

- 在大型项目中，所有开发人员都在同一个代码库上工作，会导致团队协作变得复杂

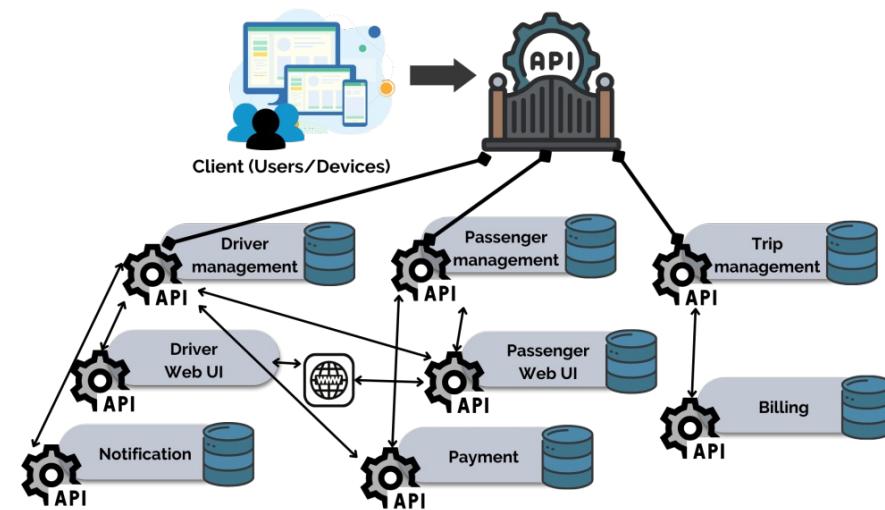
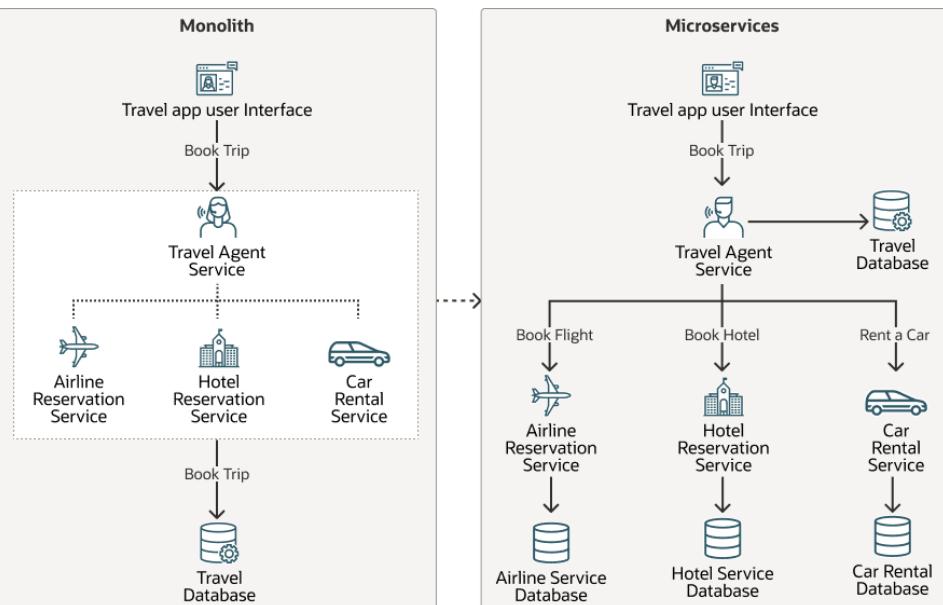
微服务 Microservices

□ Microservices (from Wikipedia [2])

- Microservices are a **software development technique** that arranges an application as **a collection of loosely coupled services**

□ Application

- A complex service provided to the user (e.g., e-commerce portal)



微服务架构的特点

“Service-oriented architecture
composed of
loosely-coupled elements
that have
bounded contexts”

微服务架构的特点

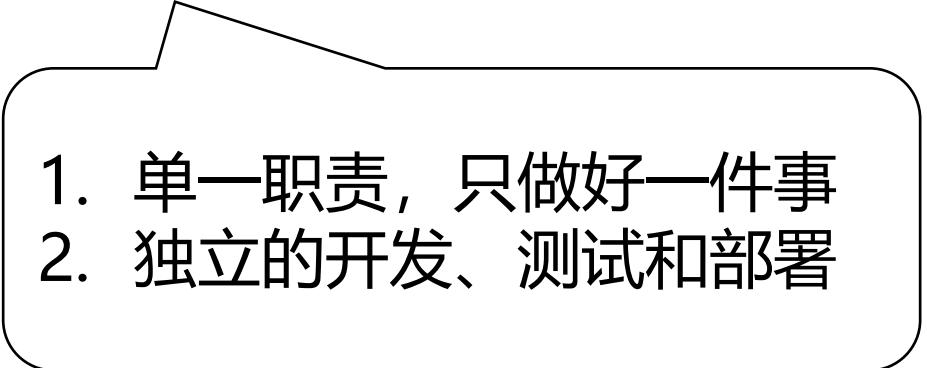
“Service-oriented architecture

composed of
loosely-coupled elements
that have
bounded contexts”

每个微服务通过轻量级的通信机制（如HTTP、RPC）进行通信

微服务架构的特点

“Service-oriented architecture
composed of
loosely-coupled elements
that have
bounded contexts”

- 
1. 单一职责，只做好一件事
 2. 独立的开发、测试和部署

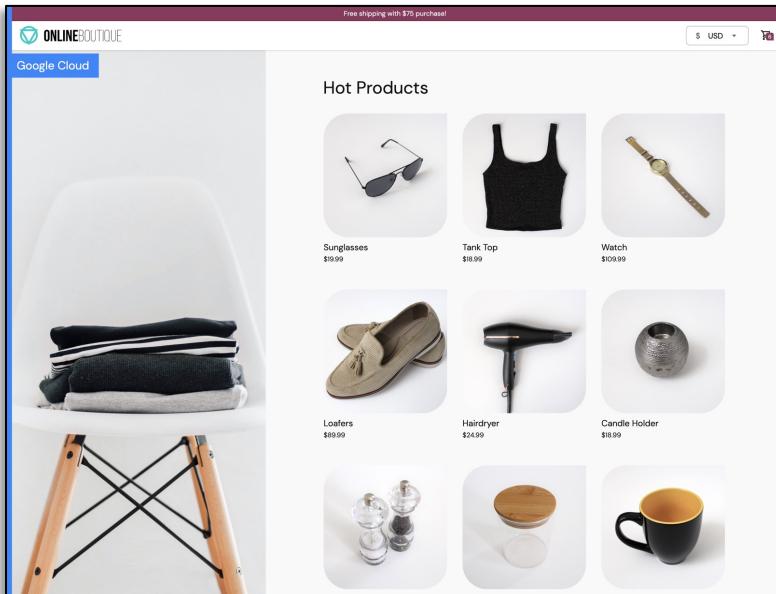
微服务架构的特点

“Service-oriented architecture
composed of
loosely-coupled elements
that have
bounded contexts”

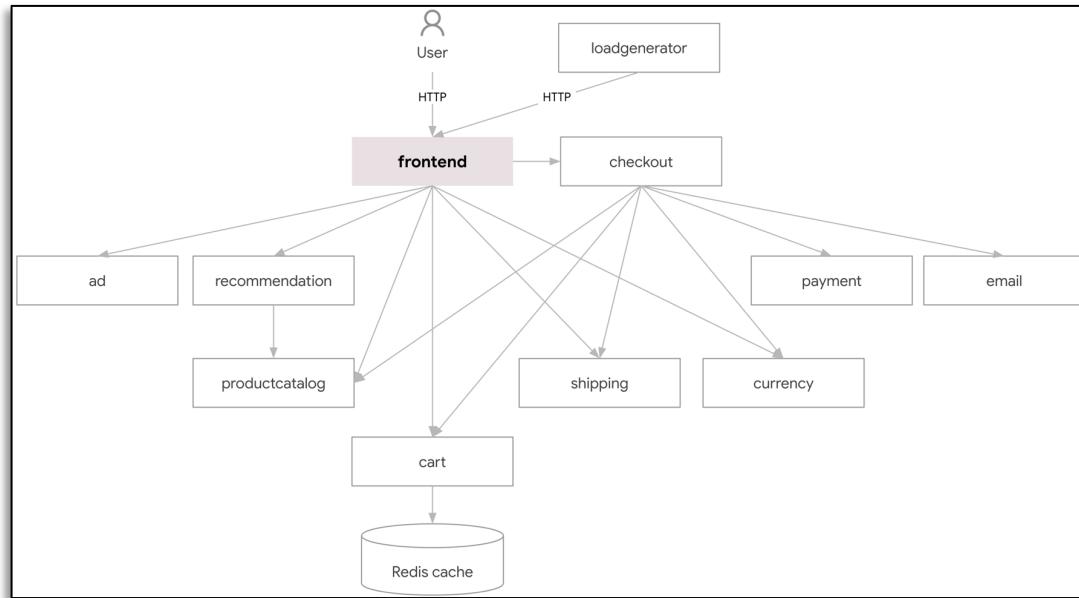
每个服务都有自己独立
的代码库和数据库等

微服务例子

用户视角



系统管理员视角



传统部署方法：将所有组件部署在几个 VM 中
(通常只有 1 个)

微服务部署方法：不同组件部署在独立的容器中

微服务架构的优势

韧性 (Resilience)

- 微服务瘫痪不会影响整个应用程序（无单点故障）
- 可以启动多个微服务实例，实例出故障时服务请求可重定向到正常实例

可扩展性 (Scalability)

- 可以在不同的机器上生成微服务（共享负载）
- 可以独立扩展每个微服务，例如， N 个文字处理应用程序与 M ($M \ll N$) 个拼写检查服务

微服务架构的优势

敏捷性 (Agility)

- 可以同时独立启动多个微服务，减少加载时间
- 并行启动多个微服务的不同版本，逐渐将流量重定向到新版本
- 可以与多个请求者共享同一服务（例如，数据库）

亚马逊微服务例子

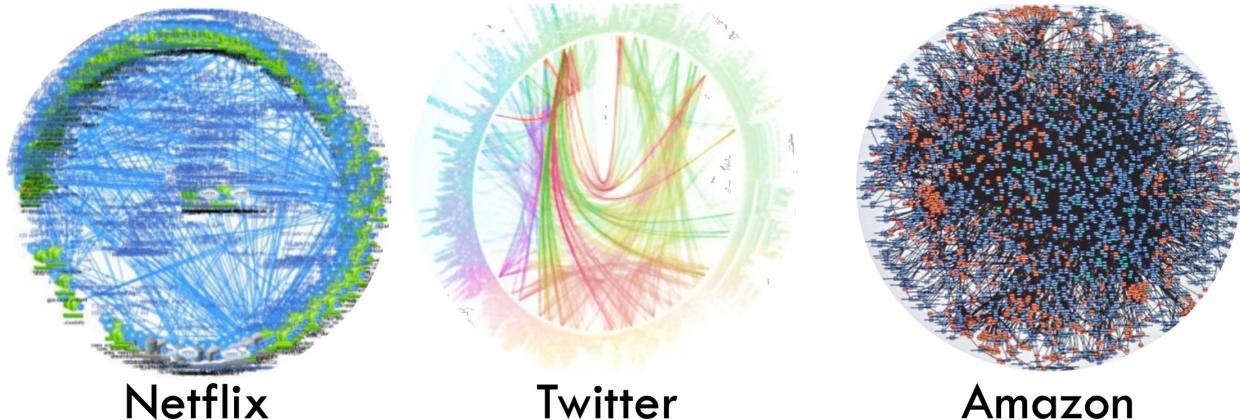
Thousands of teams	= 50 million deployments
× Microservices	a year
× Continuous delivery	每小时 5708 个更新，或每0.63秒1个
× Multiple environments	

灵活性 (Flexibility)

- 每个微服务都可以使用不同的编程语言开发

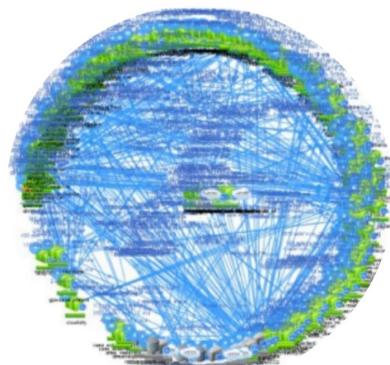
微服务架构的挑战

- **分布式系统的复杂性**: 微服务本质上是分布式系统，需要处理网络延迟、网络分区、服务发现、负载均衡等问题
- **服务间通信**: 服务间的通信需要考虑性能、可靠性和安全，常见的通信模式包括同步的 HTTP 或 gRPC 调用，以及异步的消息队列
- **数据一致性**: 在微服务架构中，不同的服务可能拥有自己的数据库，保持数据一致性变得更加复杂

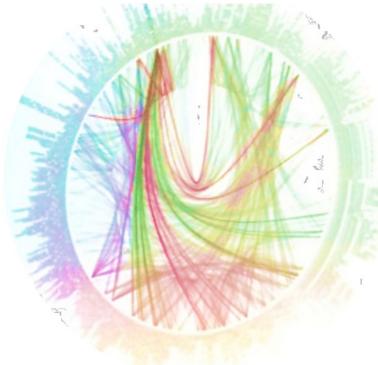


微服务架构的挑战

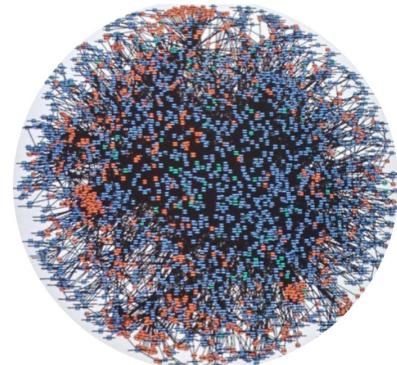
- **测试复杂性**: 测试微服务系统比测试单体系统更加复杂，因为需要模拟和测试服务间的交互，以及分布式事务。
- **部署和发布管理**: 多个服务的部署和发布过程更加复杂，需要自动化的部署工具和流程。
- **运维挑战**: 监控、日志记录和故障排查在分布式环境中更加困难，需要集中式日志管理和监控系统。
- **安全性**: 在微服务架构中，每个服务都可能成为潜在的安全风险点，需要仔细管理认证和授权



Netflix



Twitter



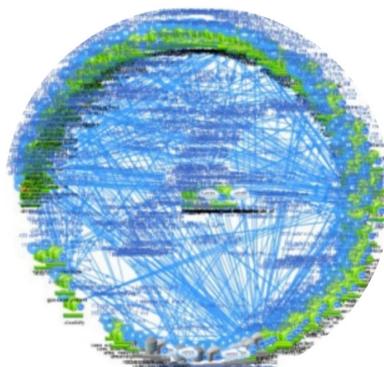
Amazon

微服务架构评估

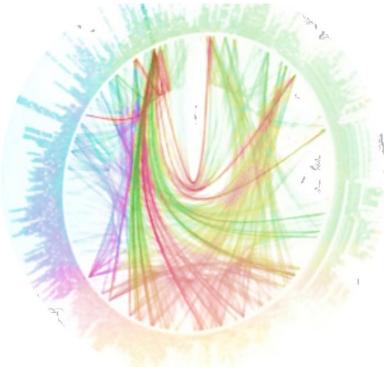
微服务架构设计的挑战

□ 微服务架构设计高度依赖专家经验

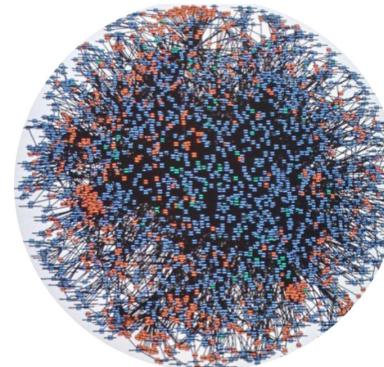
- 工业微服务系统可以拥有**数百到数千个服务**
- 随着服务的不断演化，微服务架构可能会退化，出现各种架构问题，如**重复服务、责任混乱、循环依赖和复杂的调用链等**



Netflix



Twitter



Amazon

如何对微服务的架构设计进行评估？

基于微服务交互的架构评估

口微服务之间复杂的交互

- 单个服务可能非常小且简单，但服务之间的交互可能非常复杂
- 即复杂性从服务内部转移到了架构级别，即服务之间的交互

Trace Analysis Based Microservice Architecture Measurement

Xin Peng^{*†}
Fudan University
China

Akasaka Isami^{*}
Fudan University
China

Chenxi Zhang^{*}
Fudan University
China

Xiaofeng Guo
Alibaba Group
China

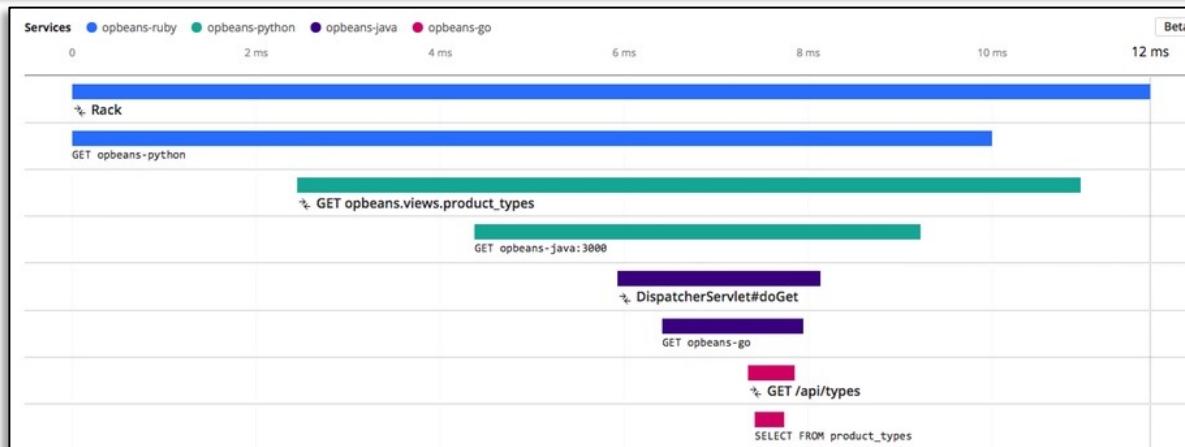
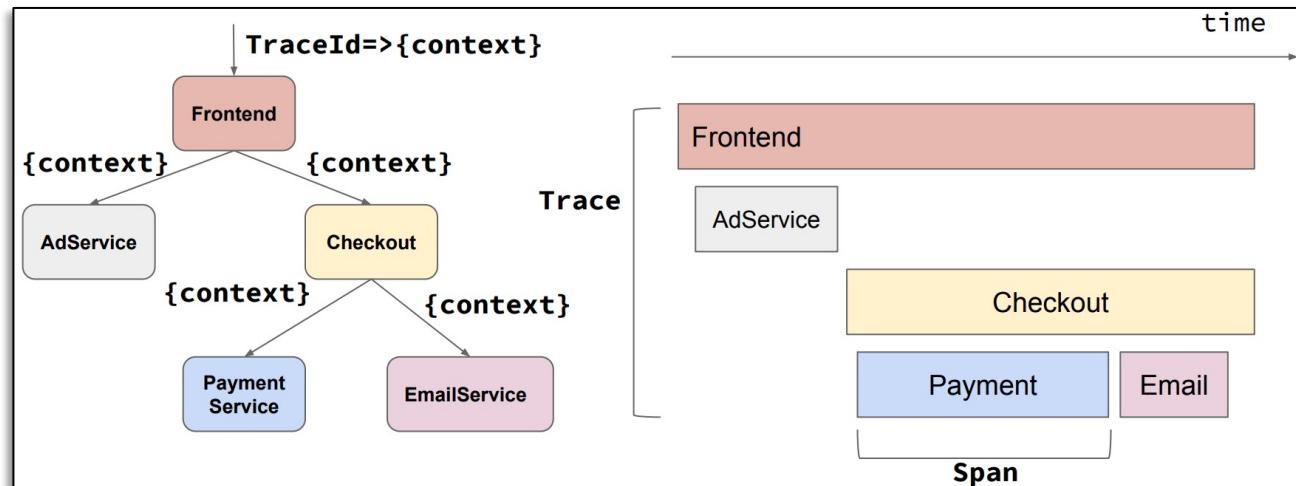
Zhongyuan Zhao^{*}
Fudan University
China

Yunna Cui^{*}
Fudan University
China

提出一个用于微服务架构度量的 trace 数据模型，该模型能够对
请求的执行过程以及**接口和服务之间的交互**进行细粒度分析

分布式追踪 Distributed tracing

Trace 记录一个请求从接收到处理并最终响应的完整路径，包括经过的所有服务和这些服务的交互过程

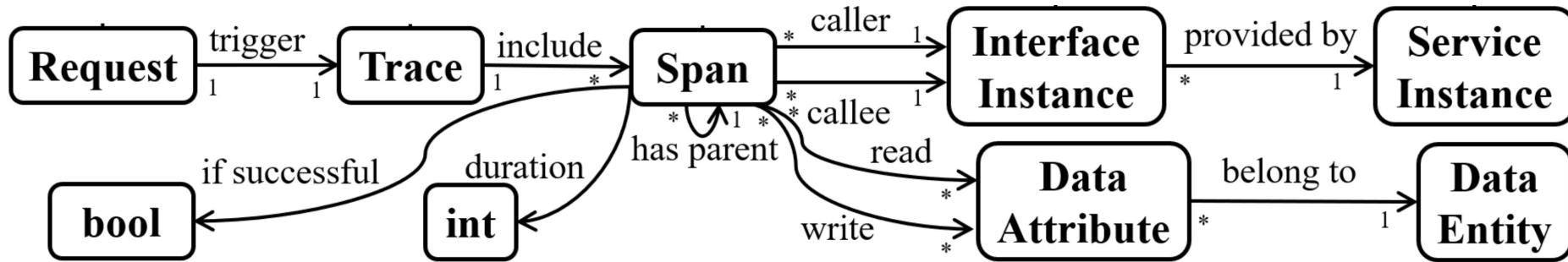


```
{
  "name": "hello",
  "context": {
    "trace_id": "0x5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "0x051581bf3cb55c13"
  },
  "parent_id": null,
  "start_time": "2022-04-29T18:52:58.114201Z",
  "end_time": "2022-04-29T18:52:58.114687Z",
  "attributes": {
    "http.route": "some_route1"
  },
  "events": [
    {
      "name": "Guten Tag!",
      "timestamp": "2022-04-29T18:52:58.114561Z",
      "attributes": {
        "event_attributes": 1
      }
    }
  ]
}
```

Trace 数据模型

包含以下对象：服务 (service)、接口 (interface)、请求 (request)、追踪 (trace) 和 span

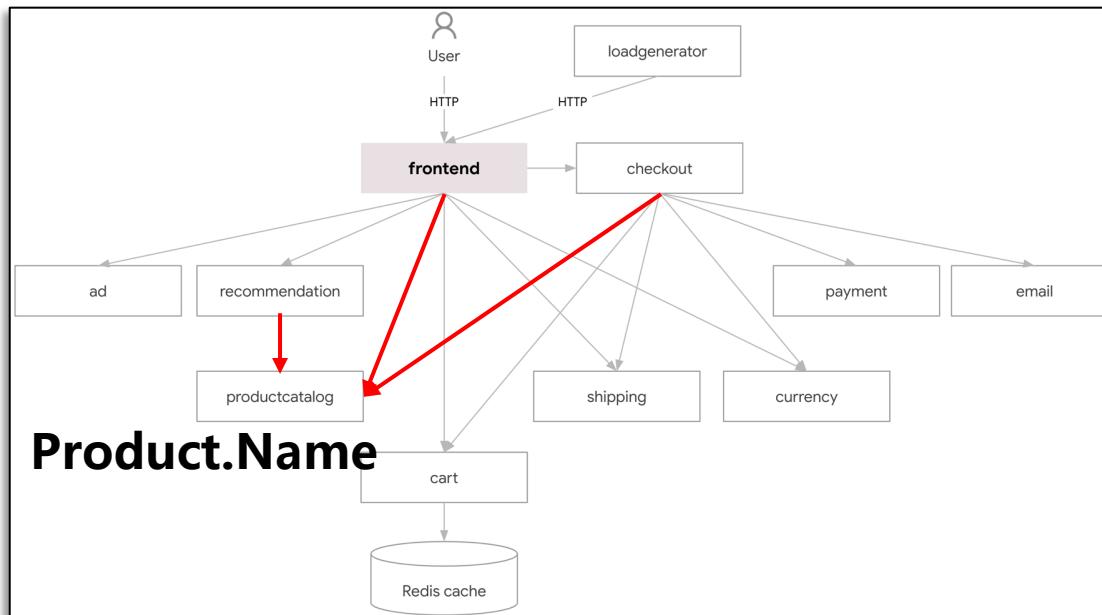
- 一个服务提供多个接口，并可以实例化为多个服务实例
- 一个请求触发一系列的服务调用，这些调用由一个追踪来表示
- 一条 trace 包括一组 span，每个 span 代表一个接口实例到另一个接口实例的调用
- 每个 span (除 root span 外) 都有一个 parent span，并记录其执行结果 (成功与否) 和持续时间 (以毫秒为单位)



Trace 数据模型

口微服务的数据访问 (Data access)

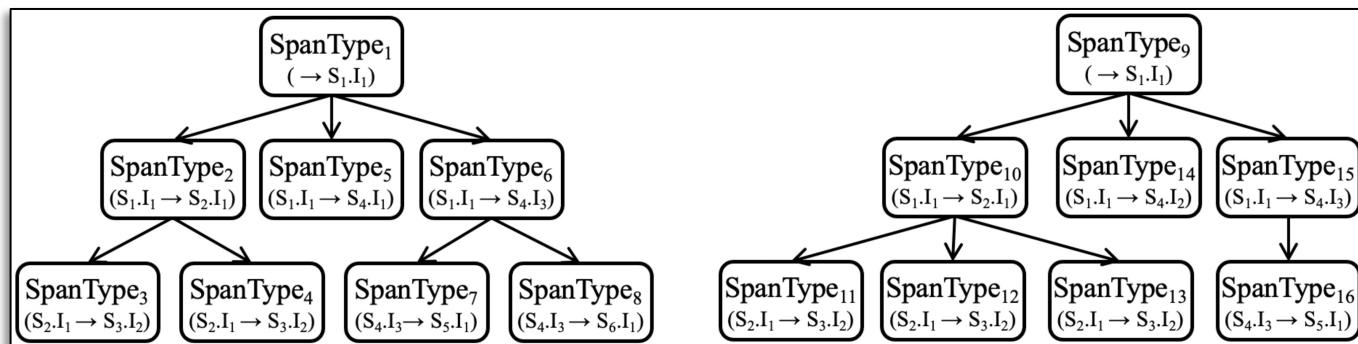
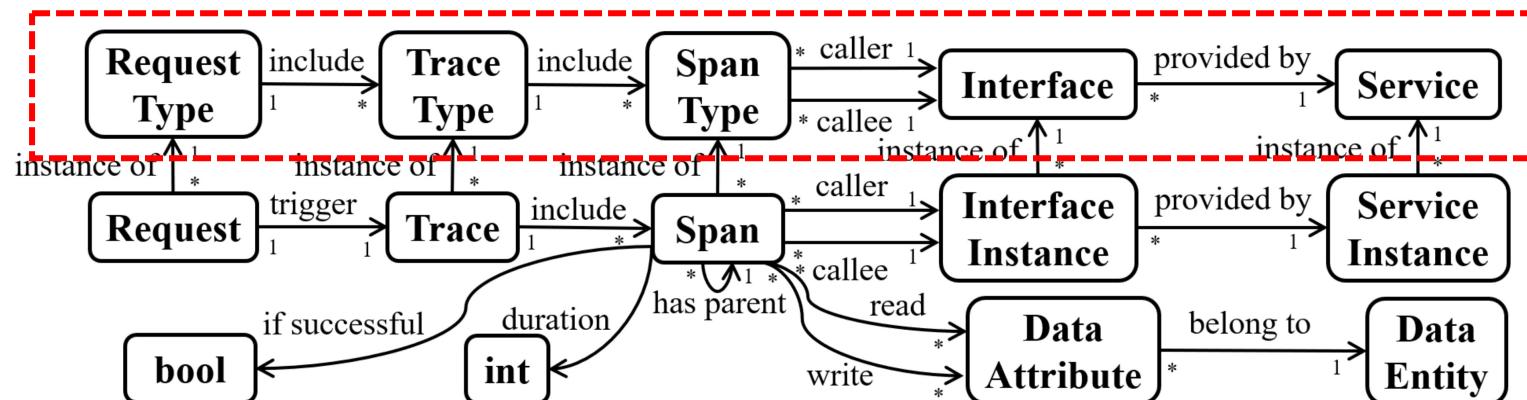
- Span 的数据访问行为：在 span 期间读取或写入的数据字段 (**data fields**)，不包括其 child spans
- 不同服务的数据库可能包含相同的数据字段
- 因此，不同数据库中的数据字段可能对应于具有唯一名称的相同数据属性，例如 “**Product.Name**” 和 “**User.ID**”



Trace 数据模型

□ 追踪抽象 (Trace abstraction)

- Trace 抽象基于**请求类型**将 trace 抽象为不同的类型
- 一种 trace 类型具有完全相同结构，即涉及同一组**按相同顺序调用的服务接口**



具有相同请求类型的不同 trace 类型

基于微服务交互的架构评估

Trace Analysis Based Microservice Architecture Measurement

Xin Peng*[†]
Fudan University
China

Akasaka Isami*
Fudan University
China

Chenxi Zhang*
Fudan University
China

Xiaofeng Guo
Alibaba Group
China

Zhongyuan Zhao*
Fudan University
China

Yunna Cui*
Fudan University
China

该论文定义了 14 个架构度量标准来衡量微服务系统的
服务独立性和调用链复杂性

□ 服务独立性 (Service Independence)

- 服务在开发 (Development)、演变 (Evolution) 和扩展 (Scaling) 方面如何独立于彼此？

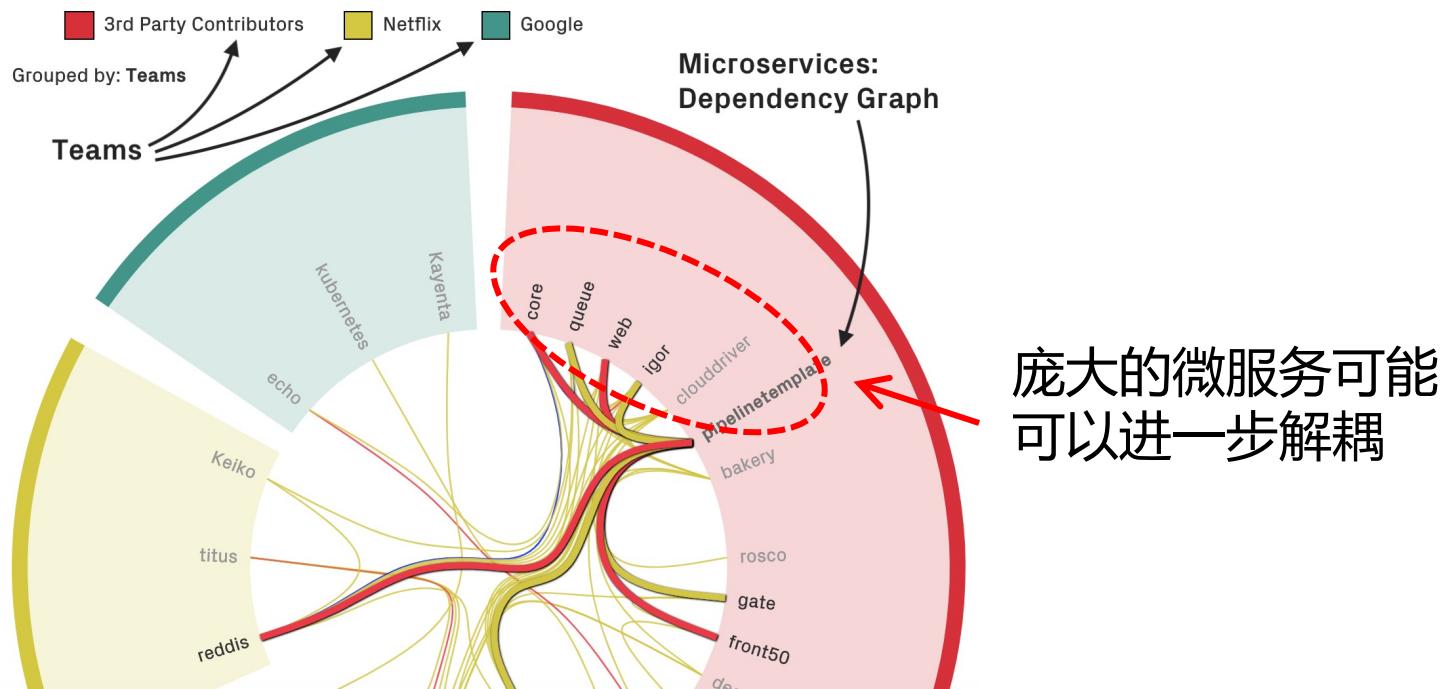
□ 调用链复杂性 (Invocation Chain Complexity)

- 服务的调用链有多复杂？

基于微服务交互的架构评估

□ 服务独立性

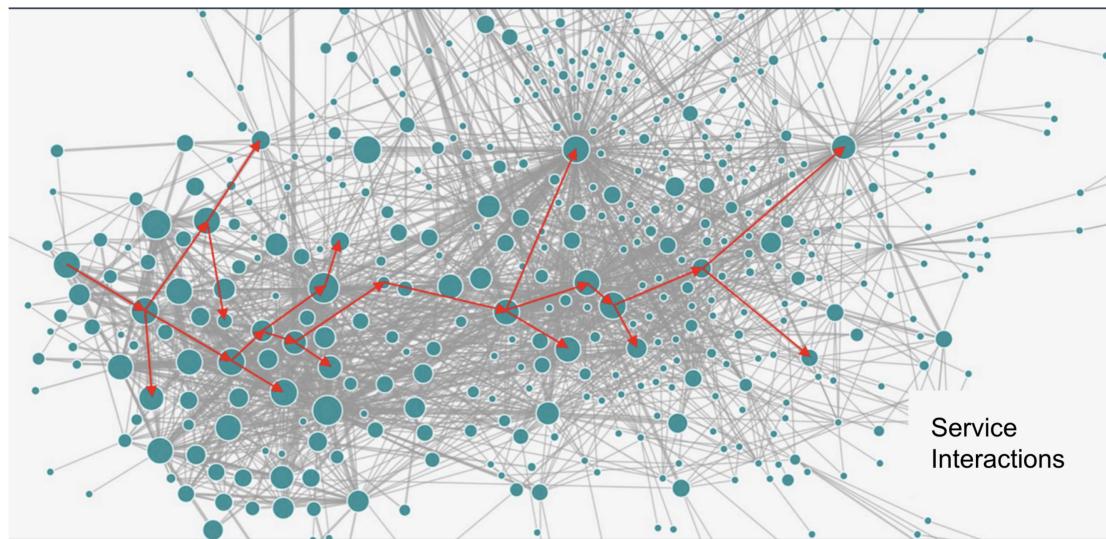
- 表明了优化服务解耦的机会 (service decomposition), 例如将一个服务分解为多个小服务, 或合并多个服务
- 包括三部分的指标, 即**服务内聚性 (service cohesion)**, **服务耦合性 (service coupling)**, 以及**读写比 (read-write ratio)**



基于微服务交互的架构评估

口调用链 (即 trace) 复杂性

- 表明开发人员在理解**服务交互**和调用链中的**故障传播**时的复杂性
- 它也可能影响系统的性能和可用性，因为复杂的调用链需要**更多的处理时间**，并且在面对服务故障时**更容易出现问题**



Complex microservice RPC call graph at Uber

架构评估指标

□ 14 个指标定义在不同级别上

- **系统级 (System level)** 的指标反映整个微服务系统的特性，例如，系统服务之间的循环依赖数量
- **请求类型级别 (Request type level)** 的指标反映了一个请求类型的特性，例如，请求类型的执行路径的复杂性
- **服务级别 (Service level)** 的指标反映了一个单独服务的特性，例如，从数据访问的角度看一个服务的接口的相关性
- **服务对级别 (Service pair level)** 的指标反映了两个服务之间的依赖或相关性，例如，两个服务共享数据访问的程度
- **接口对级别 (Interface pair level)** 的指标，例如，对于相同数据字段的两个接口的读/写比例

服务独立性 – 服务内聚性

服务内聚性：衡量同一服务的接口之间关联程度以及服务目标的集中程度

服务独立性 – 服务内聚性

□ 数据访问内聚性 (Data Access Cohesion, DAH)

- 对于依赖数据库的服务，其接口可能会读取或写入一些数据字段
- 这些接口预期会**共享一些数据字段的访问**，否则它们可以被分解为访问独立数据字段的不同服务

$$dah(in_1, in_2) = \frac{|in_1.accAttrs \cap in_2.accAttrs|}{|in_1.accAttrs \cup in_2.accAttrs|}$$

计算两个接口访问的共同数据属性的百分比

$$DAH(S) = \frac{\sum_{in_i, in_j \in S.infs, i < j} dah(in_i, in_j)}{(|S.infs| \times (|S.infs| - 1)) / 2}$$

平均所有接口对的数据访问内聚性

$in.accAttrs$: 接口 in 访问的数据属性集合

$S.infs$: 服务 S 所包含的所有接口

服务独立性 – 服务内聚性

口前驱调用内聚性 (Predecessor Invocation Cohesion, PIH)

- 服务的每个接口可能会被其他服务调用以达到特定目的
- 如果服务的接口关联紧密并且服务目标集中，那么它们可能被相似的服务调用

$$pih(in_1, in_2) = \frac{|in_1.predSrvs \cap in_2.predSrvs|}{|in_1.predSrvs \cup in_2.predSrvs|}$$

计算调用两个接口的服务的百分比

$$PIH(S) = \frac{\sum_{in_i, in_j \in S.infs, i < j} pih(in_i, in_j)}{(|S.infs| \times (|S.infs| - 1)) / 2}$$

平均所有接口对的前驱调用内聚性

in.predSrvs: 接口 *in* 的直接前驱服务集合，即那些直接调用 *in* 的服务。

服务独立性 – 服务内聚性

口后继调用内聚性 (Successor Invocation Cohesion, SIH)

- 一个服务的每个接口可能会调用其他服务以完成其功能
- 如果一个服务的接口之间关系紧密且服务目标明确，那么它们可能会调用相似的后继服务

$$sih(in_1, in_2) = \frac{|in_1.succSrvs \cap in_2.succSrvs|}{|in_1.succSrvs \cup in_2.succSrvs|}$$

计算两个接口直接或间接调用相同
服务的百分比

$$SIH(S) = \frac{\sum_{in_i, in_j \in S.infs, i < j} sih(in_i, in_j)}{(|S.infs| \times (|S.infs| - 1)) / 2}$$

平均所有接口对的后继调用内聚性

$in.succSrvs$: 接口 in 的直接和
间接后继服务集合，即那些直
接或间接被 in 调用的服务。

服务独立性 – 服务耦合

服务耦合：度量两个服务之间的关联程度或依赖程度

服务独立性 – 服务耦合

□ 数据访问耦合 (Data Access Coupling, DAP)

- 如果两个服务访问同一数据属性，那么它们就是相关的
- 共享数据属性访问并不要求两个服务共享数据库

$$DAP(S_1, S_2) = \frac{|S_1.accAttrs \cap S_s.accAttrs|}{|S_1.accAttrs \cup S_s.accAttrs|}$$

由两个服务访问的相同数据属性的百分比计算

$S.accAttrs$: 服务 S 访问的数据属性集合，
即 $\bigcup_{in \in S.infs} in.accAttrs$

服务独立性 – 服务耦合

口前驱调用耦合 (Predecessor Invocation Coupling, PIP)

- 每个服务可能会被其他服务为了特定目的而调用
- 如果两个服务经常被同一服务直接调用 (表明它们服务于相关的目的), 那么这两个服务就是相关的

$$PIP(S_1, S_2) = \frac{|S_1.\text{predSrvs} \cap S_2.\text{predSrvs}|}{|S_1.\text{predSrvs} \cup S_2.\text{predSrvs}|}$$

由调用了 S_1 和 S_2 的服务的百分比计算

$S.\text{predSrvs}$: 服务 S 的直接前驱服务集合, 即
 $\bigcup_{in \in S.infs} in.\text{predSrvs}$

服务独立性 – 服务耦合

口后继调用耦合 (Successor Invocation Coupling, SIP)

- 每个服务可能会调用其他服务来完成其功能
- 如果两个服务经常直接或间接调用同一服务，因为它们通过调用同一服务来完成相似的功能，那么这两个服务就是相关的

$$SIP(S_1, S_2) = \frac{|S_1.\text{succSrvs} \cap S_2.\text{succSrvs}|}{|S_1.\text{succSrvs} \cup S_2.\text{succSrvs}|}$$

由被 S_1 和 S_2 直接或间接调用的相同服务的百分比计算

$S.\text{succSrvs}$: 服务 S 的直接或间接后继服务集合，
即 $\bigcup_{in \in S.infs} in.\text{succSrvs}$

服务独立性 – 服务耦合

□ 直接调用耦合 (Direct Invocation Coupling, DIP)

- 一个服务依赖于被它调用的服务，即存在依赖性
- 两个服务的直接调用耦合，是基于在给定 trace 类型中其他服务出现的条件下，从一个服务的调用出现的条件概率来定义的

$$dip_{cp}(S_1, S_2) = \frac{|traTypes(S_1 \rightarrow S_2)|}{|S_2.traTypes|}$$

$$DIP(S_1, S_2) = \max(dip_{cp}(S_1, S_2), dip_{cp}(S_2, S_1))$$

两个服务的直接调用耦合由不同调用方向的最大值决定

$traTypes(S_1 \rightarrow S_2)$: 涉及从 S_1 到 S_2 的直接调用的 trace 类型集合

$S.traTypes$: 涉及服务 S 的 trace 类型集合

服务独立性 – 服务耦合

□ 追踪共现耦合 (Trace Co-occurrence Coupling, TCP)

- 每个服务可能出现在许多不同类型的 trace 中
- 如果两个服务经常出现在同一种类型的 trace 中，那么它们就被认为是相关的，因为它们通常被用来满足相同的请求

$$tcp_{cp}(S_1, S_2) = \frac{|S_1.\text{traTypes} \cap S_2.\text{traTypes}|}{|S_1.\text{traTypes}|}$$

追踪共现耦合是根据一个服务在 trace 类型中出现的条件概率给出的

$$TCP(S_1, S_2) = \max(tcp_{cp}(S_1, S_2), tcp_{cp}(S_2, S_1))$$

两个服务的追踪共现耦合由不同调用方向的最大值决定

调用链复杂性

口调用链长度 (Length of Invocation Chain, LIC)

- 与源代码中的长方法/函数类似，长服务调用链也增加了开发者的认知负担
- 长服务调用链影响性能和可用性

$$LIC(R) = \max_{T \in R.traceTypes} |T.spanTypes|$$

请求类型 R 的 LIC 由其所有 trace 类型中最大的跳数决定

$R.traceTypes$: 请求类型 R 包含的 trace 类型集合

$T.spanTypes$: 追踪类型 T 包含的 span 类型集合

调用链复杂性

口重复调用次数 (Repeated Invocation Number, RIN)

- 在一个 trace 中，一个接口可能多次调用另一个接口
- 这种重复的调用通常由循环调用引起，使得调用链更长，理解起来更复杂，可能会影响性能和可用性

$$RIN(R) = \max_{T \in R.traceTypes, in \in T.infs} |spanTypes(T, in)|$$

包含在 R 的 trace 类型中对同一服务接口的最大调用次数

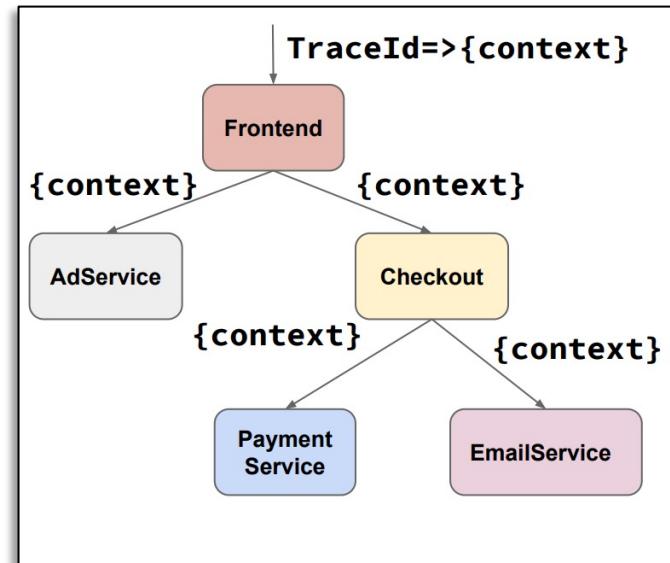
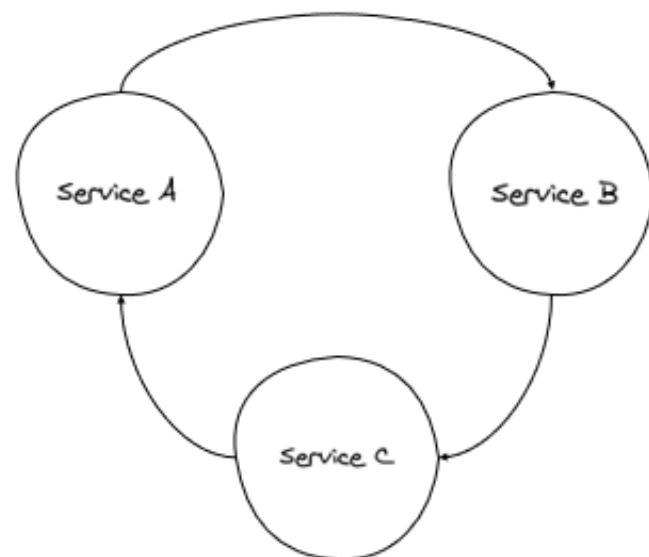
$T.infs$: Trace 类型 T 的所有 span 类型涉及的接口

$spanTypes(T, in)$: 属于追踪类型 T 并以 in 为被调用者的 span 类型集合

调用链复杂性

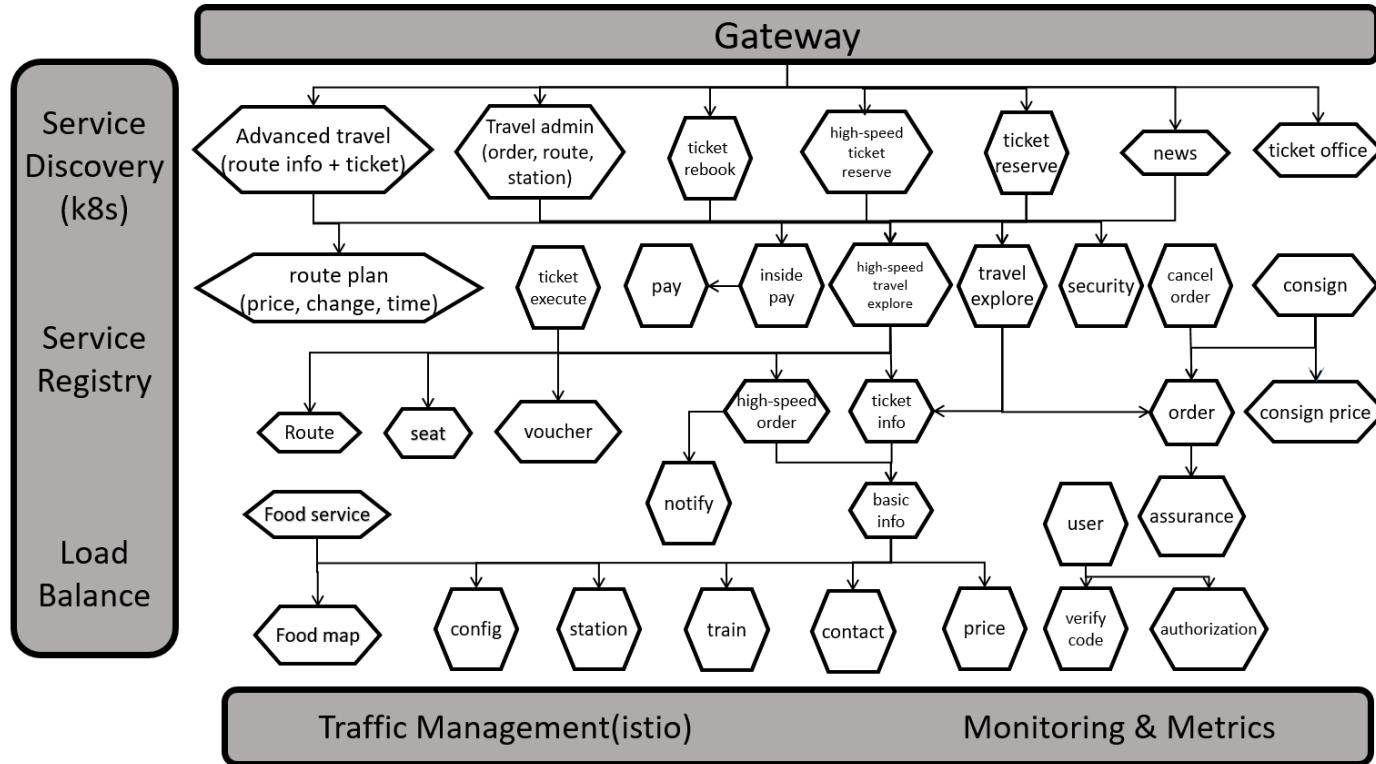
口请求级别循环依赖 (Request-level Cyclic Dependency, RCD)

- 循环依赖是微服务架构中的典型反模式
- 请求类型 R 的请求级别循环依赖，由 $RCD(R)$ 表示，用于衡量 R 中的循环依赖数量
- RCD 有两个子度量，即 RCD-IN 和 RCD-SE，分别考虑**接口依赖**和**服务依赖**

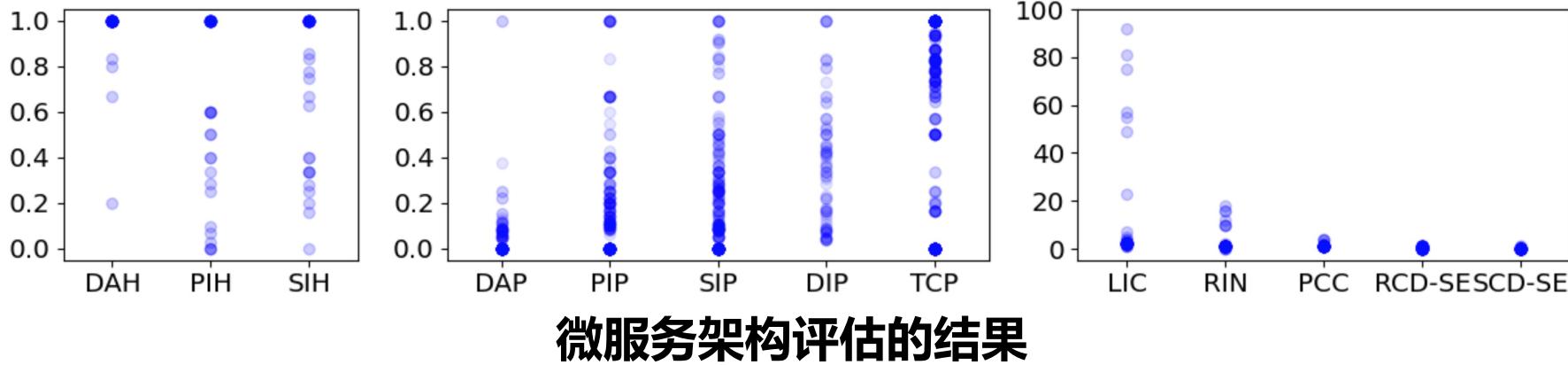


案例分析 – Train Ticket

- 论文基于提出的指标分析开源的微服务项目 – Train Ticket
- Train Ticket 是一个拥有 41 个微服务的铁路票务系统，用户可以在这里搜索、预订、支付和取消火车票



案例分析 – Train Ticket

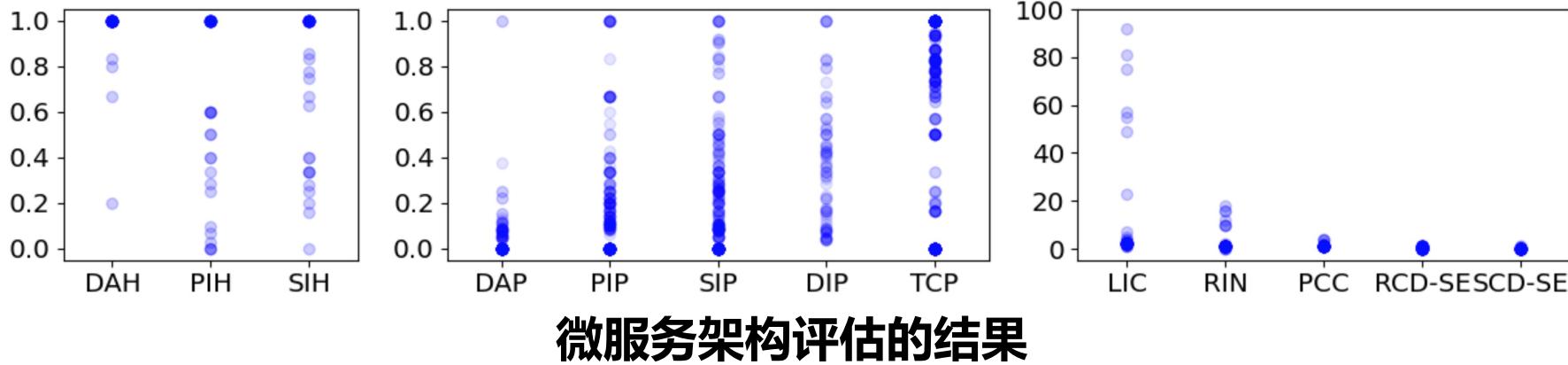


微服务架构评估的结果

□ Issue 1: Too Small Services

- 一些服务过小且高度耦合
- 比如 ts-food-service 和 ts-food-map-service
- 首先，它们的**直接调用耦合 (DIP)** 为 1，表明 ts-food-map-service 的唯一调用者是 ts-food-service
- 其次，ts-food-service 的**后续调用内聚度 (SIH)** 为 1，表明 ts-food-service 的所有接口都依赖于 ts-food-map-service
- **可以考虑合并过小的服务**

案例分析 – Train Ticket

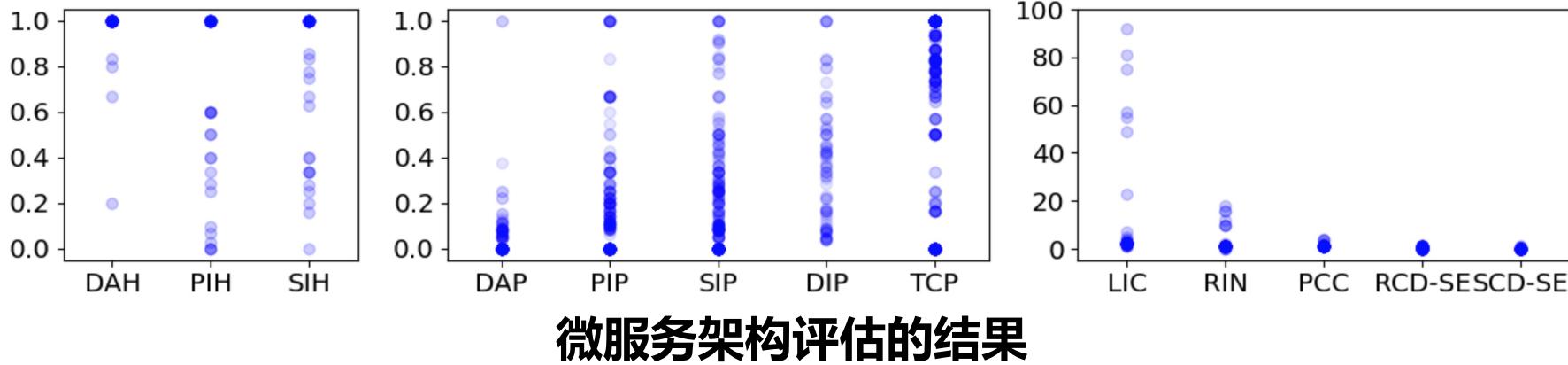


微服务架构评估的结果

□ Issue 2: Duplicate Services

- 比如 ts-order-service 和 ts-order-other-service
- 它们的数据访问耦合 (DAP)、前驱调用耦合 (PIP) 和后继调用耦合 (SIP) 分别为 1、0.545 和 1
- **可以考虑合并功能相似的服务**

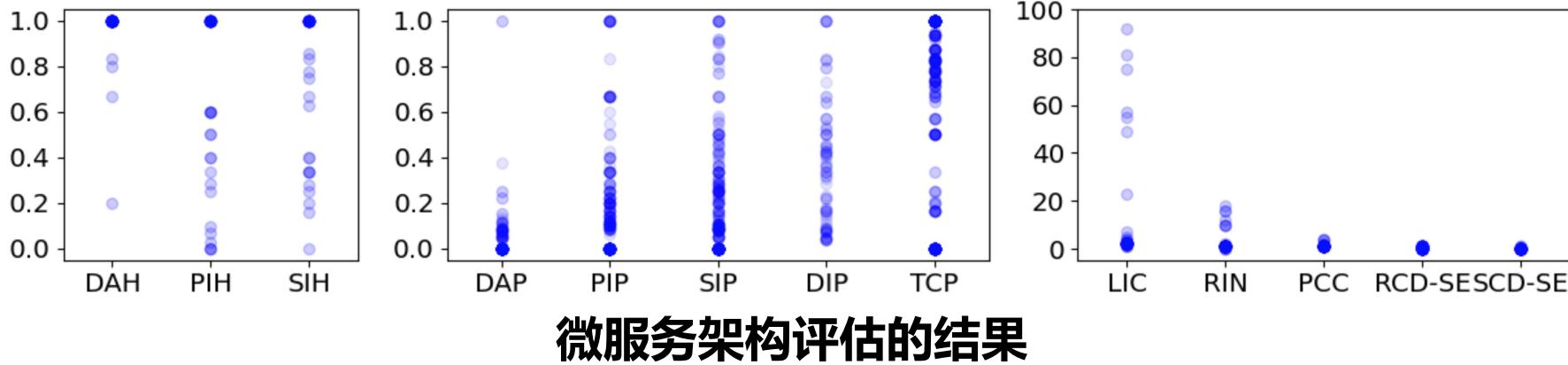
案例分析 – Train Ticket



□ Issue 3: Chaotic Responsibility

- 一些服务的职责不清晰，实现了多个并不密切相关的任务
- ts-travel-service实现了多个职责，如列车类型查询、路线信息查询、行程信息管理和余票查询。其前驱调用内聚度 (PIH) 和后继调用内聚度 (SIH) 分别为 0.28 和 0.067
- 混乱职责的问题也可能通过循环依赖反映出来。例如，ts-travel-service 和 ts-seat-service 之间存在请求级循环依赖
- **可以考虑分解职责混乱的服务**

案例分析 – Train Ticket



□ Issue 4: Complex Invocation Chains

- 存在一些调用链复杂，涉及到重复的服务调用
- 一个典型的例子是寻找最快行程票的请求，其调用链长度 (LIC) 为 92，重复调用次数 (RIN) 为 16
- 复杂调用链有两种原因
 - 在一条 trace 的不同层级多次调用同一基础服务
 - 服务在循环中被调用
- 前者需要重构服务的调用层次结构，后者用批处理代替循环调用

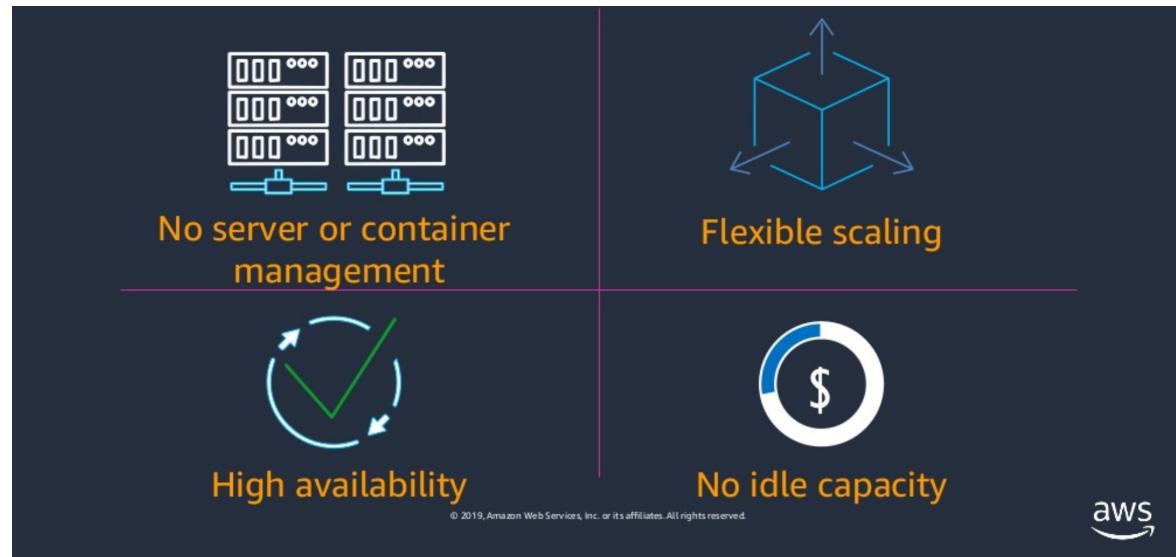
无服务计算 Serverless computing

什么是无服务计算?

□ Computing without servers?

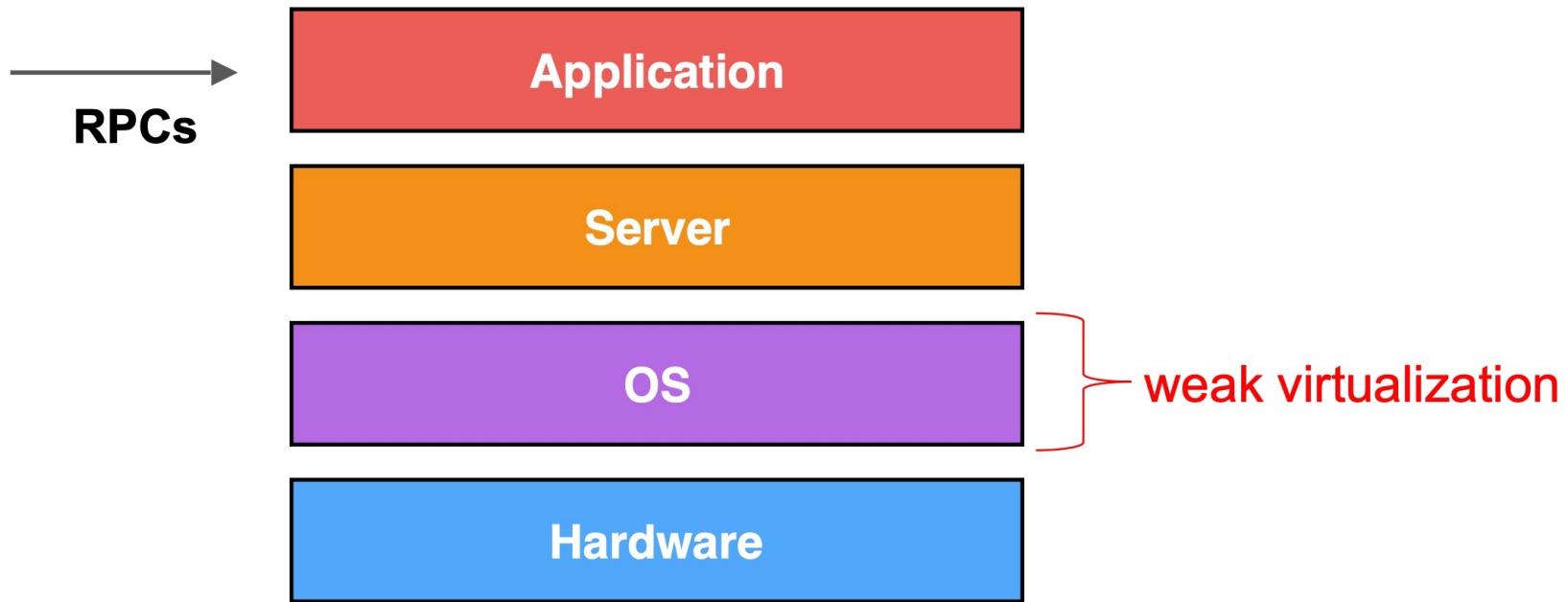
□ More precisely definition:

- Running applications without server management
- Running functions instead of containers/VMs
- Infinite scaling
- More “precise” pay-as-you-go

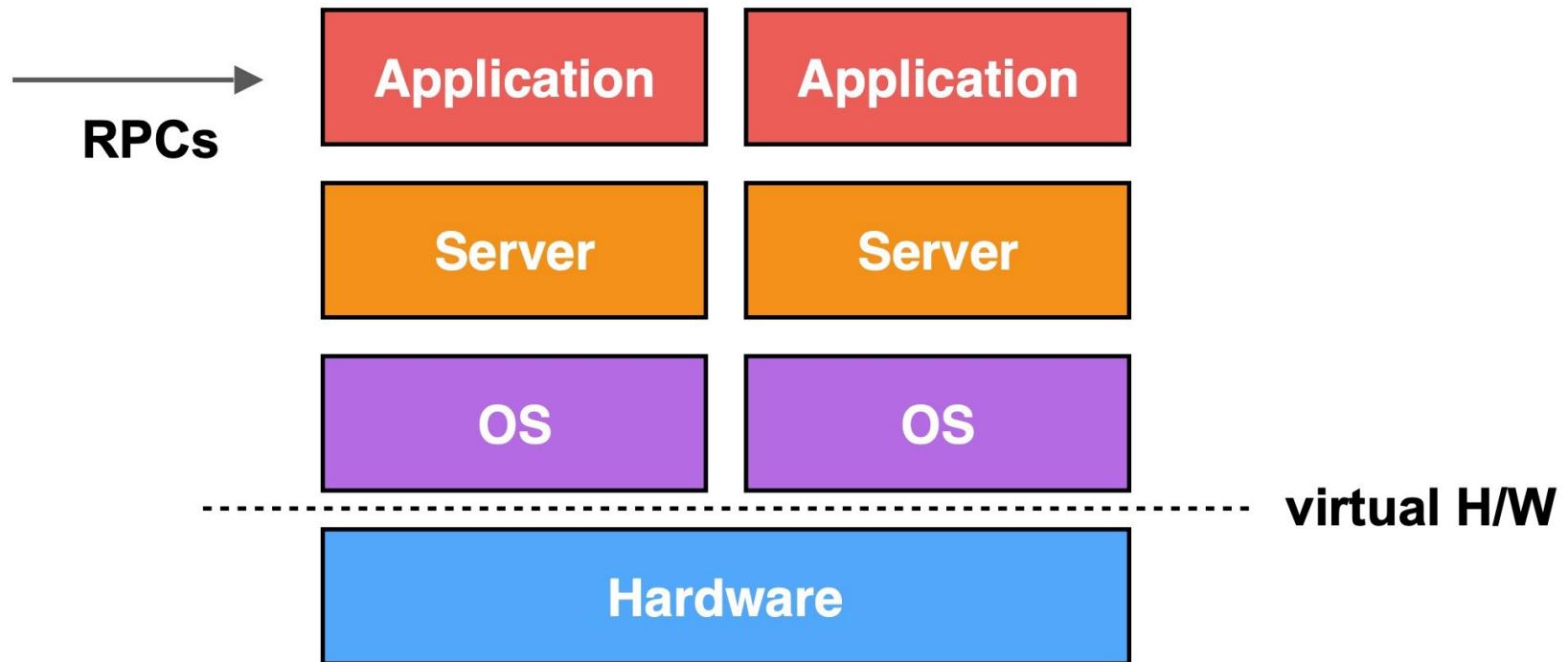


理解无服务计算的一种视角： 云和虚拟化的发展过程

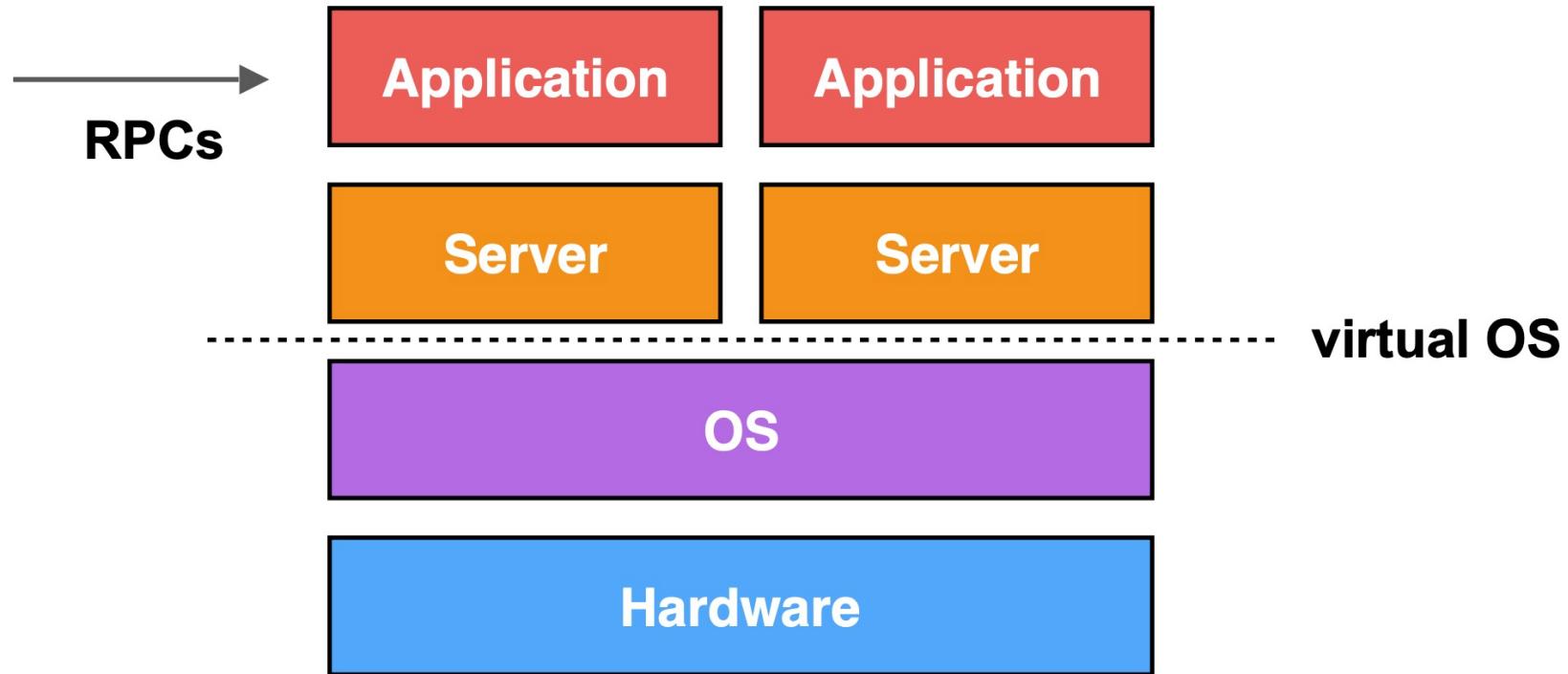
经典 Web 系统栈



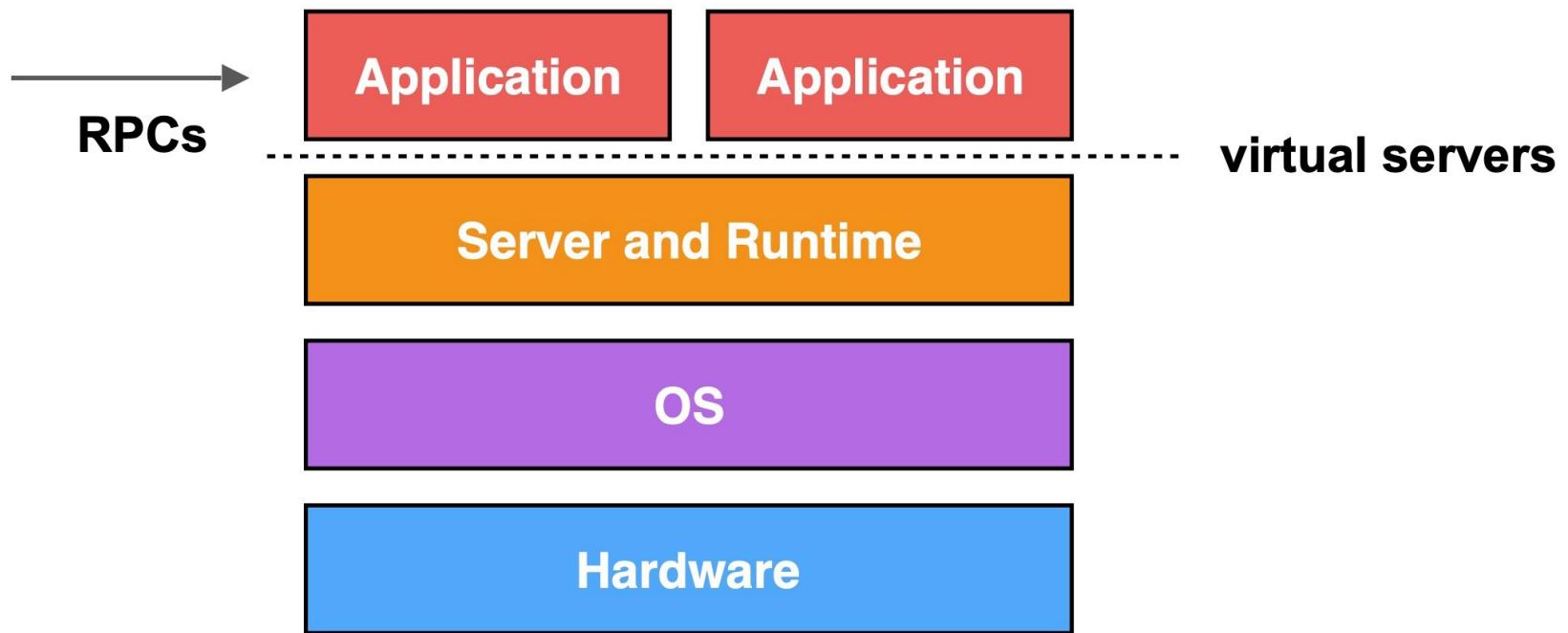
第一代：虚拟机



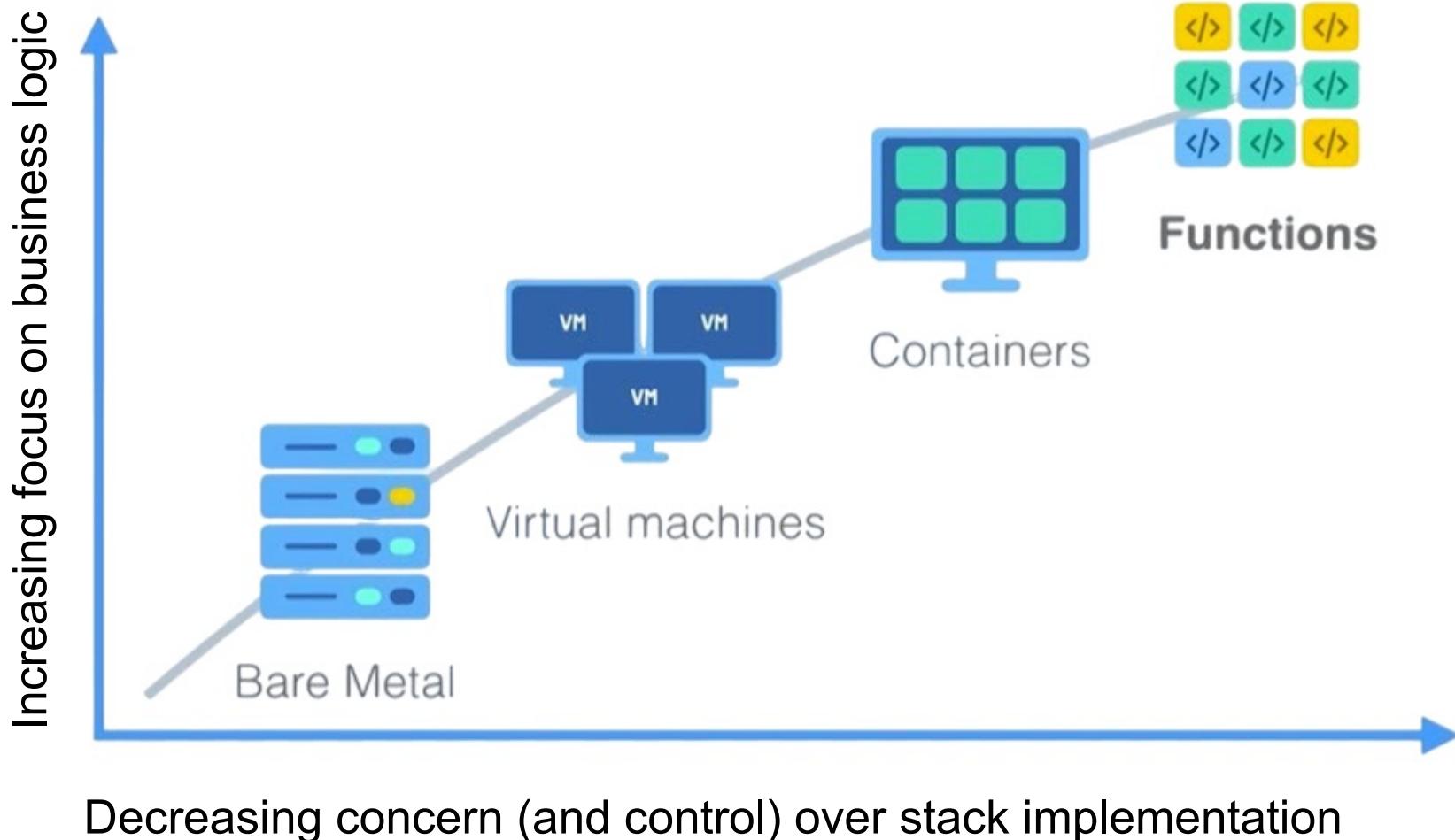
第二代：容器



第三代：无服务计算



无服务计算的发展过程



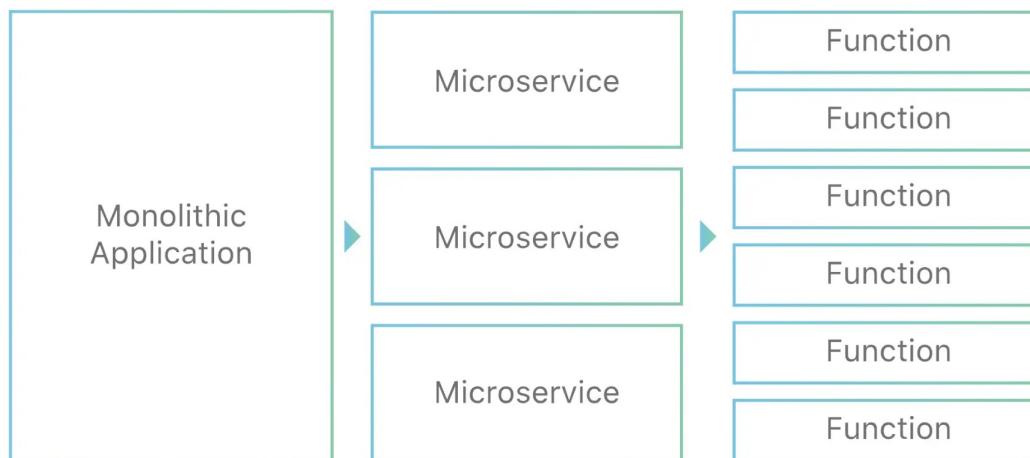
今天的无服务器计算是什么样的？

□ 主要以函数即服务 (Function as a Service, FaaS) 提供

- 云用户编写函数并部署 (不需要配置、启动虚拟机)
- 云提供商运行和管理它们

□ 仍然需要服务器，但开发人员不必管理这些服务器

- 服务器数量及其容量等决策由无服务器平台负责
- 服务器容量根据工作负载的需要自动调配



Source: Cloudflare glossary - FaaS



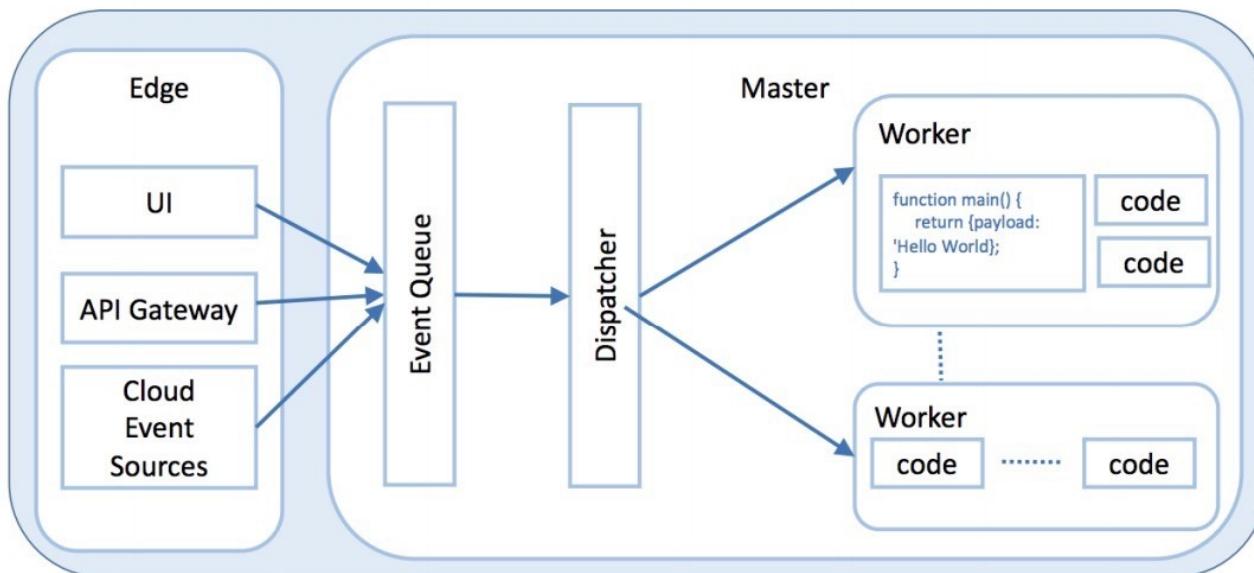
无服务器计算架构

口无服务器平台的核心功能是**事件处理系统**

- 平台管理一组用户定义的函数，接受通过 HTTP 发送或从事件源接收的事件

口挑战在于考虑成本、可扩展性和容错等指标

- 平台必须快速有效地启动函数并处理其输入
- 平台还需要对事件进行排队



第一个无服务器应用：BigQuery

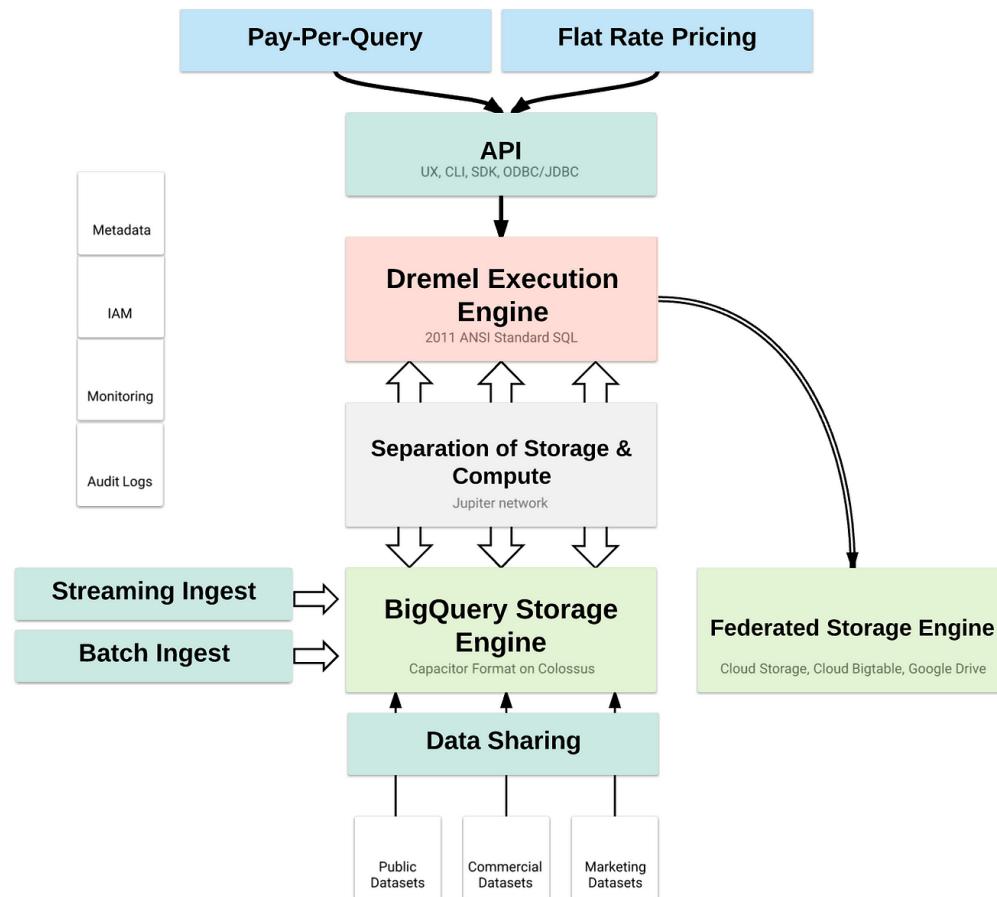
□ Fully managed Data Warehouse

- Arbitrarily large data and queries
- Pay per bytes being processed
- No concept of cluster



□ Other similar systems

- AWS Athena
- Snowflake
- ...

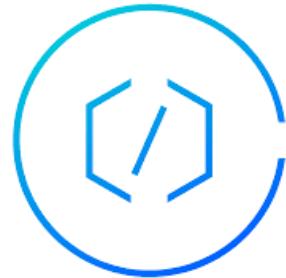


商业平台



Google Cloud Functions

 **Alibaba Cloud**



Tencent Cloud
Serverless Cloud Function



AWS Lambda



- 在 2014 年推出的事件驱动无服务器计算 FaaS 平台
- 函数可以用 Node.js, Python, Java, Go, Ruby, C# 编写
- 每个函数允许使用 128MB - 3GB 内存, 最长 15 分钟
- 最多 1000 个并发函数
- 与许多其他 AWS 服务连接

AWS Lambda 入门: <https://aws.amazon.com/cn/lambda/getting-started/>

AWS Lambda 介绍: <https://www.bilibili.com/video/BV1vf4y1X7pd/>

Lambda 函数触发和计费模型

口运行用户处理程序响应事件

- 网络请求 (RPC 处理程序)
- 数据库更新 (触发器)
- 计划事件 (计划任务)

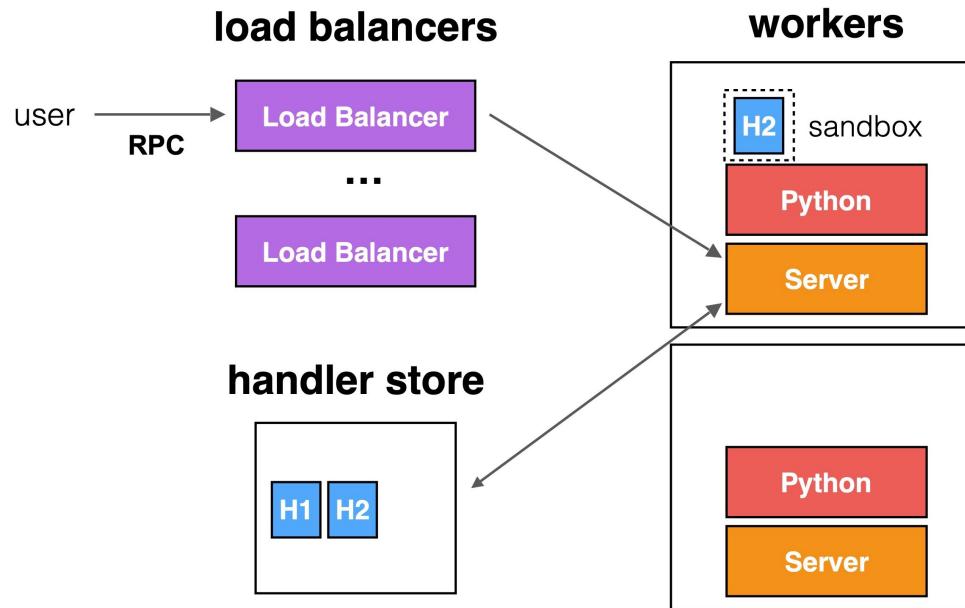


口按函数调用付费

- 没有函数运行时不收费 (无触发事件)
- 根据函数的持续时间, 配置的内存大小以及函数数量来计费
 - $\text{actual_time} * \text{memory_cap}$

内部执行模型

- 开发人员将函数代码上传到处理程序存储区 (并与URL关联)
- 事件通过 RPC (到 URL) 触发函数
- 负载平衡器通过在工作程序上启动处理程序来处理RPC请求
 - 对同一函数的调用通常发送到同一工作程序
- 处理程序在容器中进行沙盒处理
 - 如果可能，AWS Lambda 会重用同一容器来执行多个处理程序



当今天服务器产品的局限性

□ 管理状态困难且缓慢

- 必须使用 (缓慢的) 云存储!

□ 没有简单或快速的跨函数通信方式

- 必须通过云存储或其他服务

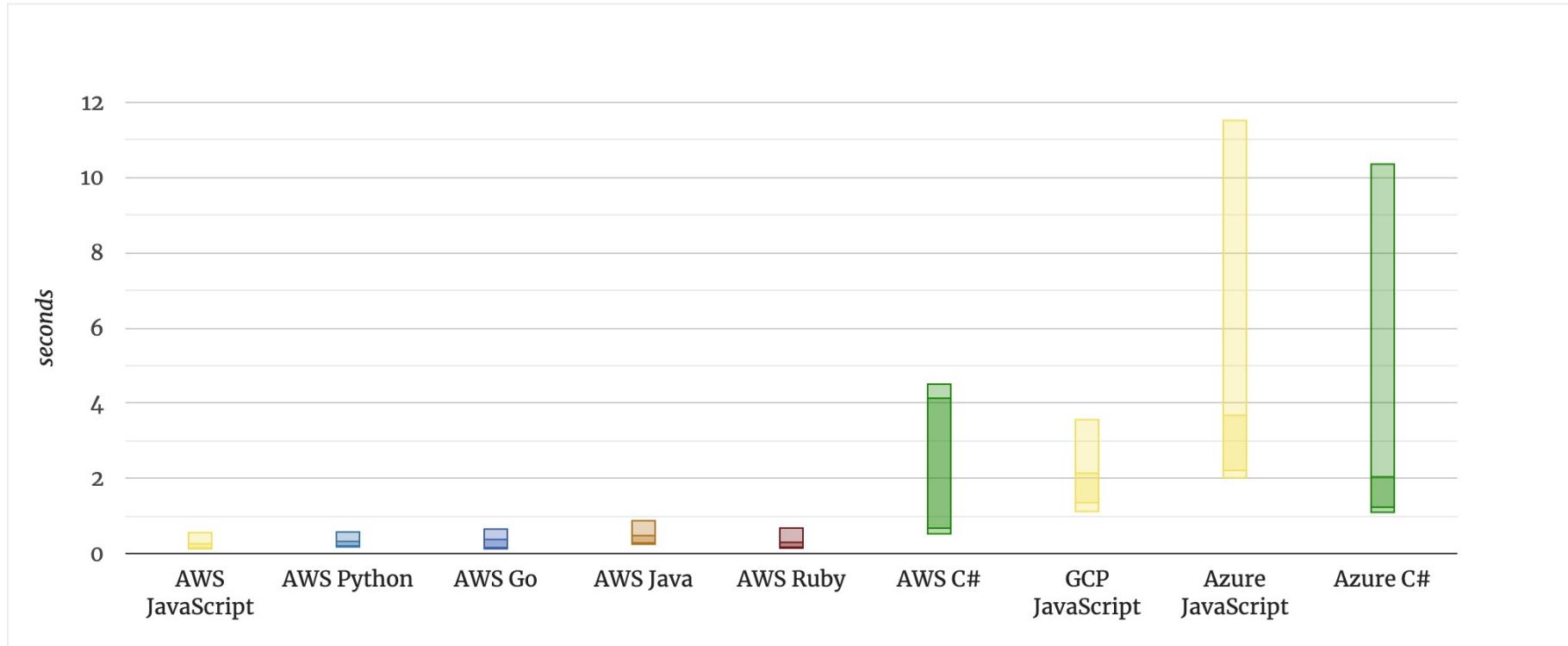
□ 函数只能使用有限的资源

□ 无法控制函数的放置或位置

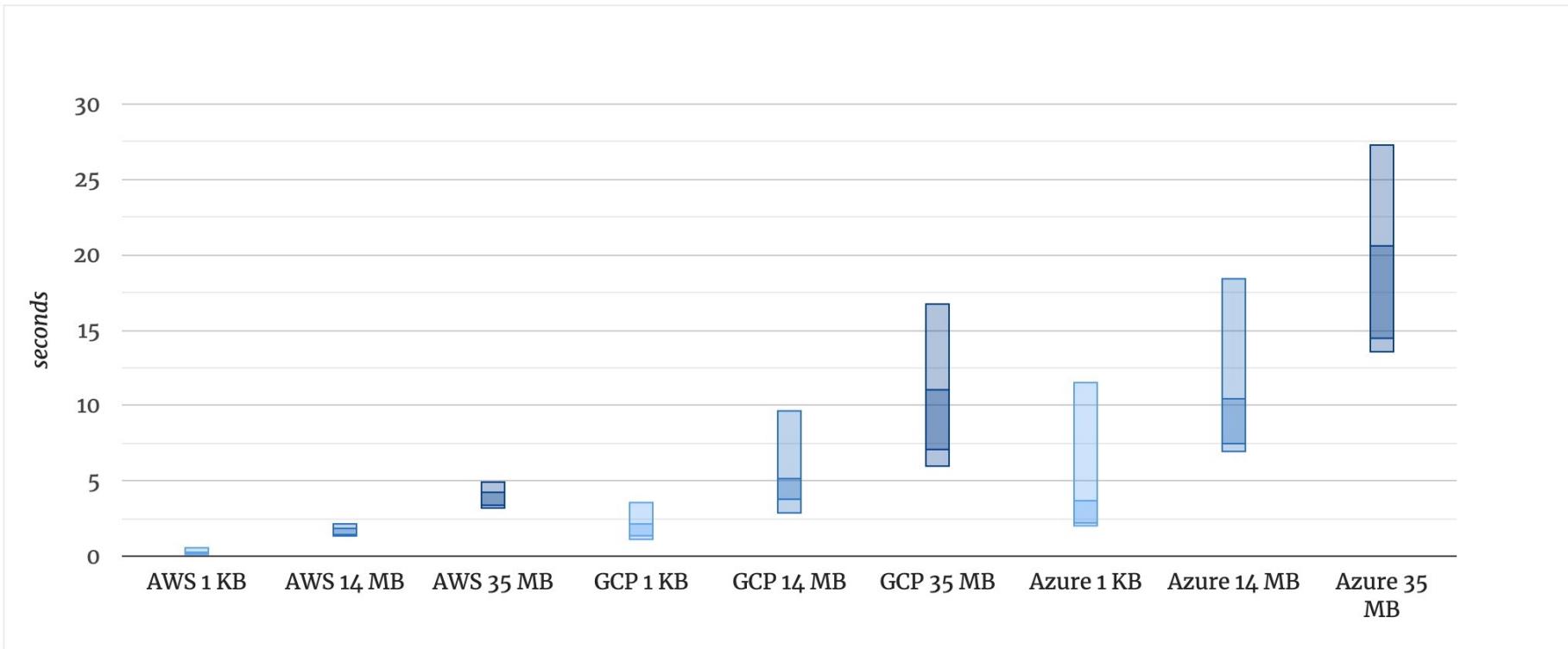
- 例如，在“冷”机器上启动函数可能会很慢

□ 计费模型并不能满足所有需求

不同语言/产品的冷启动时间



不同部署尺寸的冷启动时间



无服务器计算一些适用和不适用的场景

□ 适用的场景包括：

- 并行、独立、**无状态**的任务
- 多个函数组合在一起能创建新的、更复杂的任务

□ 不适用的场景包括：

- **有状态**的应用
- 分布式应用和协议
- 需要大量资源（尤其是内存）的应用

Stateful vs. Stateless

□ 系统在处理请求或事务时是否保留信息

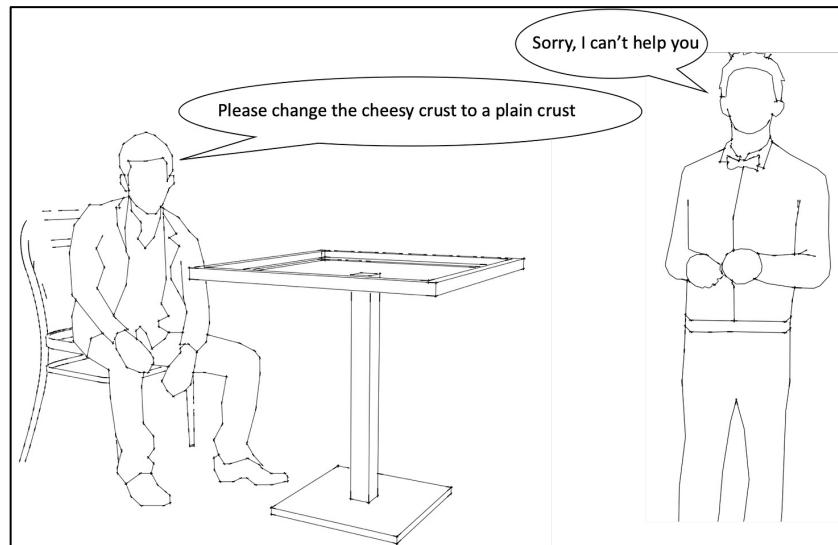
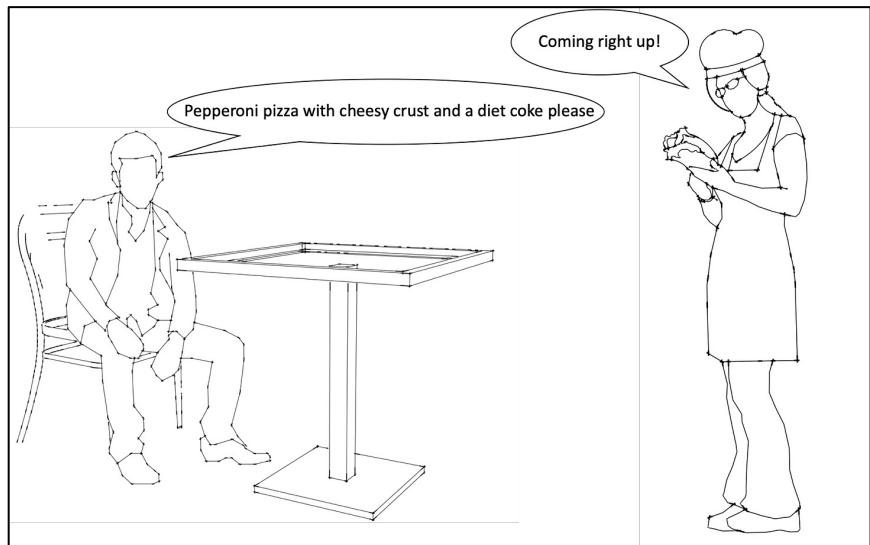
□ 有状态 (Stateful)

- 在有状态的系统中，**系统保存了客户端之前活动的信息或状态**
- 每次交互时，系统会根据客户端当前状态来做出响应
- 例如，购物网站会记住你添加到购物车中的商品

□ 无状态 (Stateless)

- 无状态系统在处理请求时**并不保留任何状态信息**
- 每个请求都是独立的，系统并不关心之前或未来的请求
- 例如，许多 RESTful Web 服务通过每个请求中的足够信息来处理该请求，不依赖服务器端存储状态

Stateful vs. Stateless



Microservice vs. FaaS

□ 粒度 (Granularity)

- 通常来说，FaaS的粒度更小，通常只包含一小段函数式代码，而微服务则可能包括一个完整的业务功能，包含多个接口和函数

□ 状态 (State)

- 微服务通常是有状态的，可以保存用户的会话信息；而 FaaS 通常是无状态的，每次函数调用都是独立的，不保留状态信息

□ 运行和扩展 (Running and extension)

- 微服务需要开发者管理服务的运行和扩展；而 FaaS 则由云平台自动管理，开发者只需编写和上传函数代码

□ 费用 (Cost)

- 微服务可能需要持续运行，会产生持续的费用；FaaS则按函数运行的次数和运行时间计费，当没有函数调用时，不产生费用



中山大學 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬
软件工程学院
chenzhb36@mail.sysu.edu.cn