微服务系统开发

陈鹏飞

中山大学
计算机学院
http://www.inpluslab.com

内容

- 背景
- 软件系统架构演化
- 微服务系统架构
- 微服务系统构建
- 微服务系统核心原则
- 微服务系统延申
  - 服务网格
  - Serverless

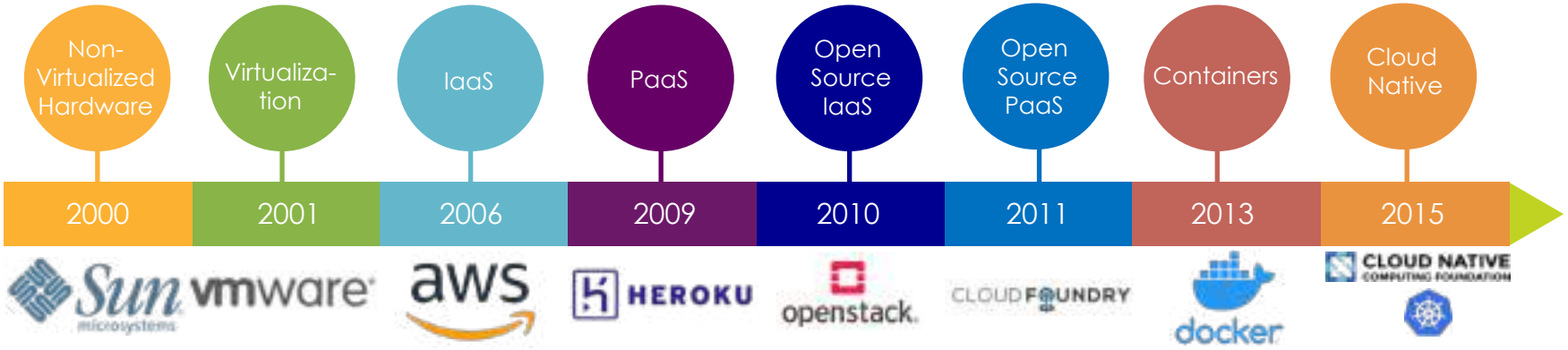
背景

- **云原生技术的崛起** https://www.cncf.io/

背景

- **云原生技术的发展**

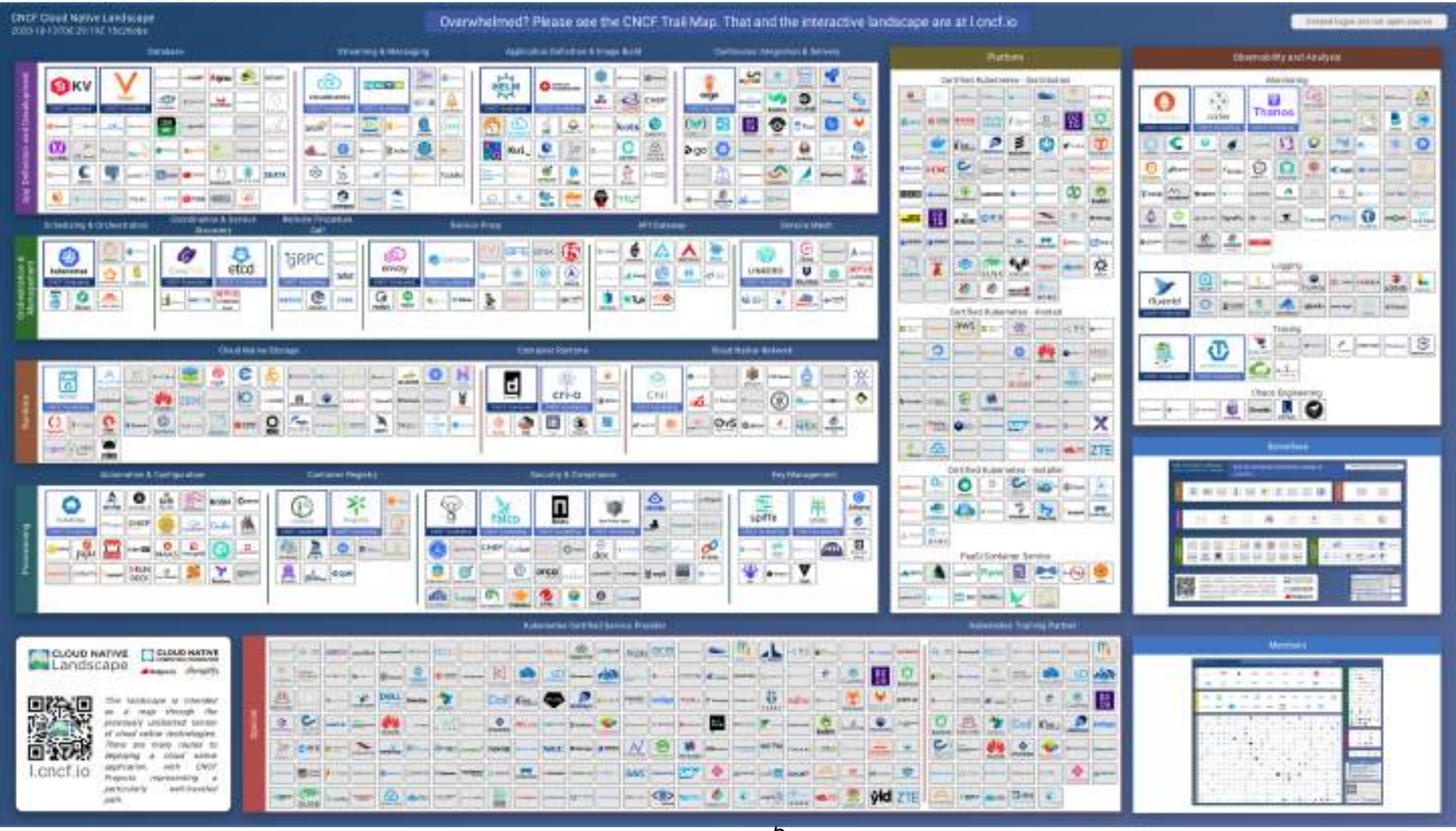

2008.01
cgroups 合并进入 Linux 内核主干
2014.06
Kubernetes 项目发布
2017.07
CNCF 成立2周年
170个成员
14个基金会项目
2004~2007
Google 大规模使用容器（cgroups）技术
2013.03
Docker 项目正式发布
2015.07
CNCF （云原生基金会）成立
22 个创始成员
Kubernetes 成为第一个 CNCF 项目
2018.07
CNCF 成立3周年
195个成员
19个基金会项目
11个孵化项目


Non-Virtualized Hardware
Virtualiza-tion
IaaS
PaaS
Open Source IaaS
Open Source PaaS
Containers
Cloud Native
2000
2001
2006
2009
2010
2011
2013
2015
Sun microsystems
vmware
aws
HEROKU
openstack.
CLOUD FOUNDRY
docker
CLOUD NATIVE COMPUTING FOUNDATION

背景

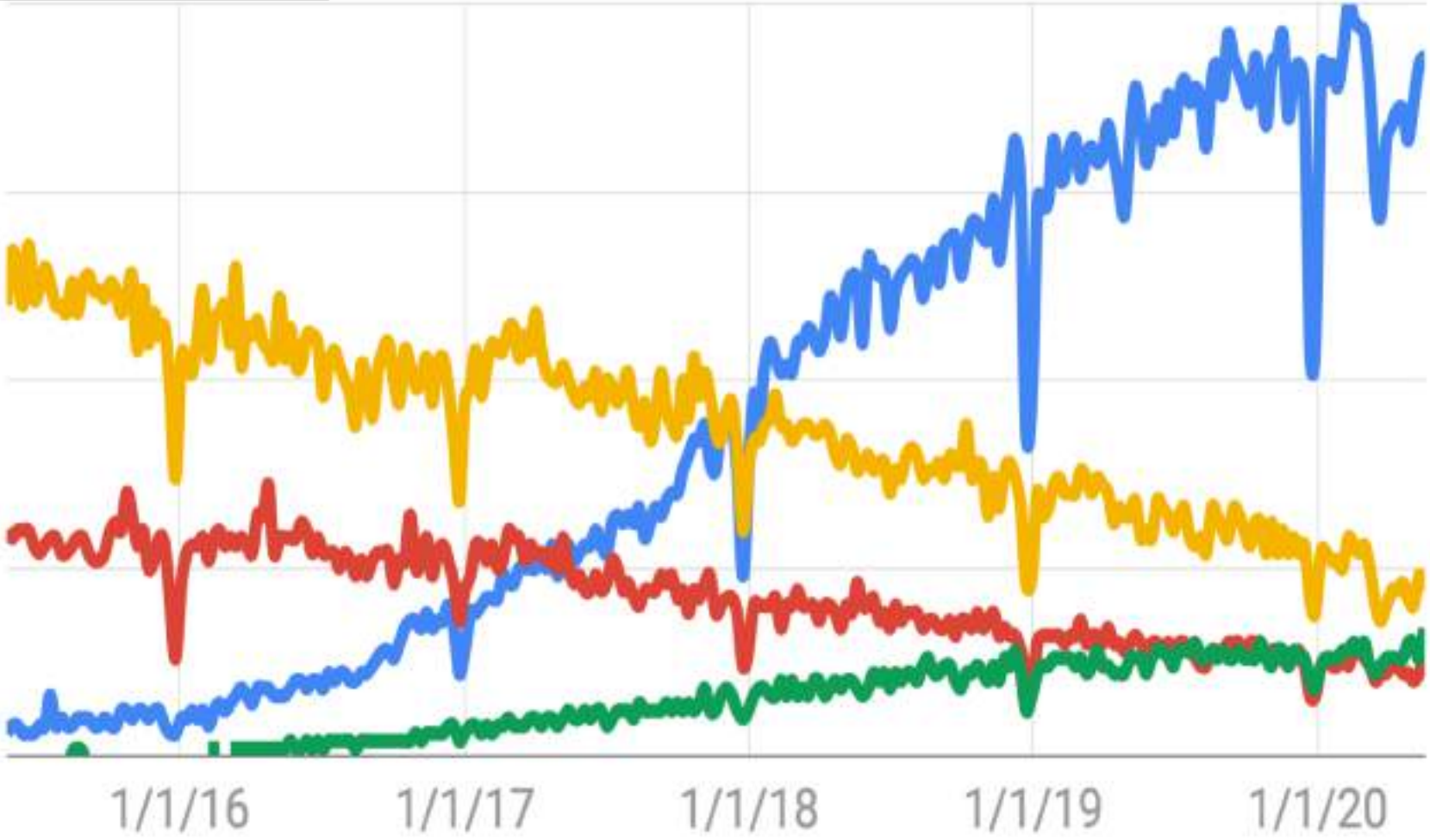
CLOUD NATIVE COMPUTING FOUNDATION

https://landscape.cncf.io/images/landscape.png



背景

- 云原生技术趋势

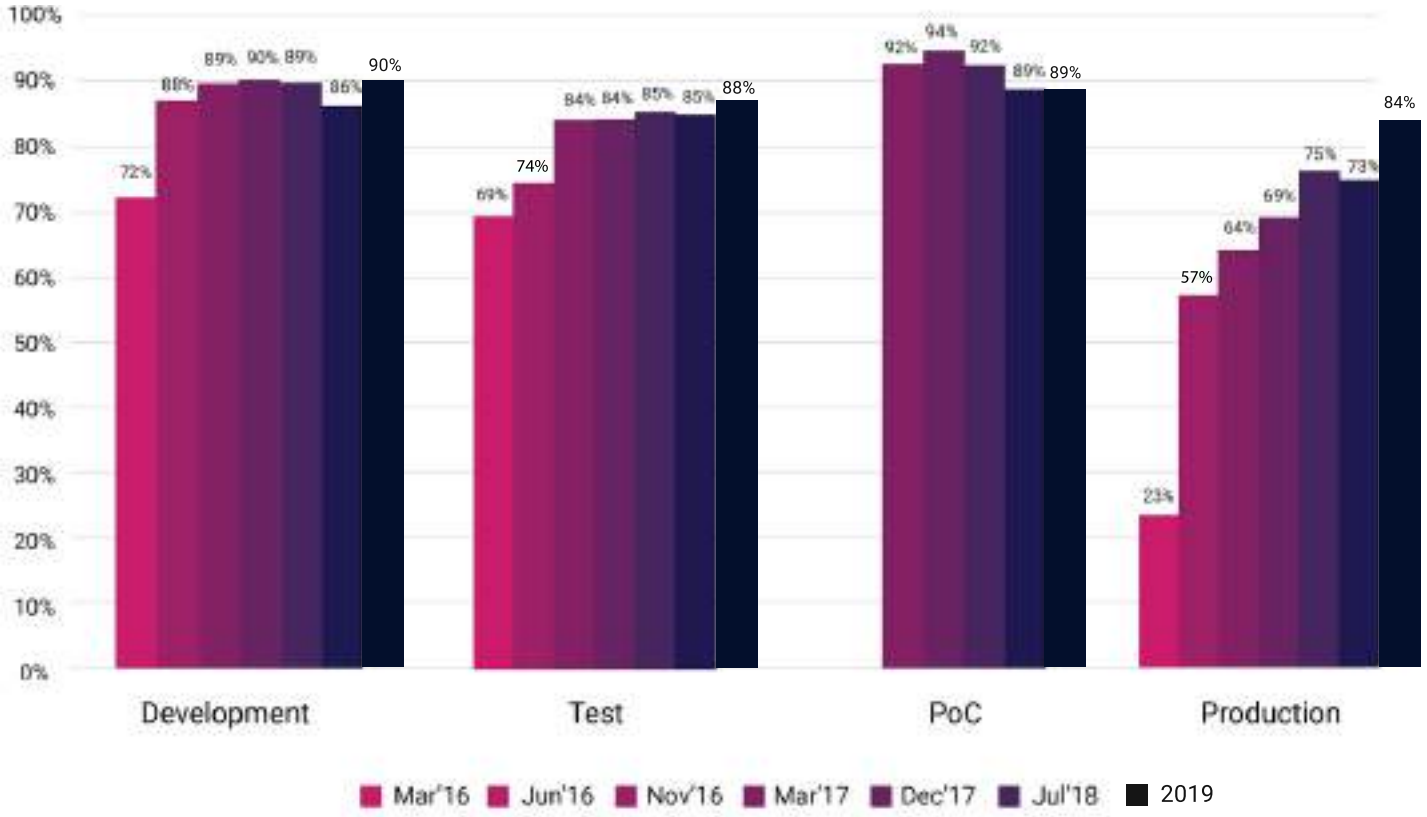

1/1/16
1/1/17
1/1/18
1/1/19
1/1/20
Kubernetes
OpenStack
Hadoop
Serverless

背景

● 云原生技术趋势

➢ 容器使用统计：

Use of Containers since 2016



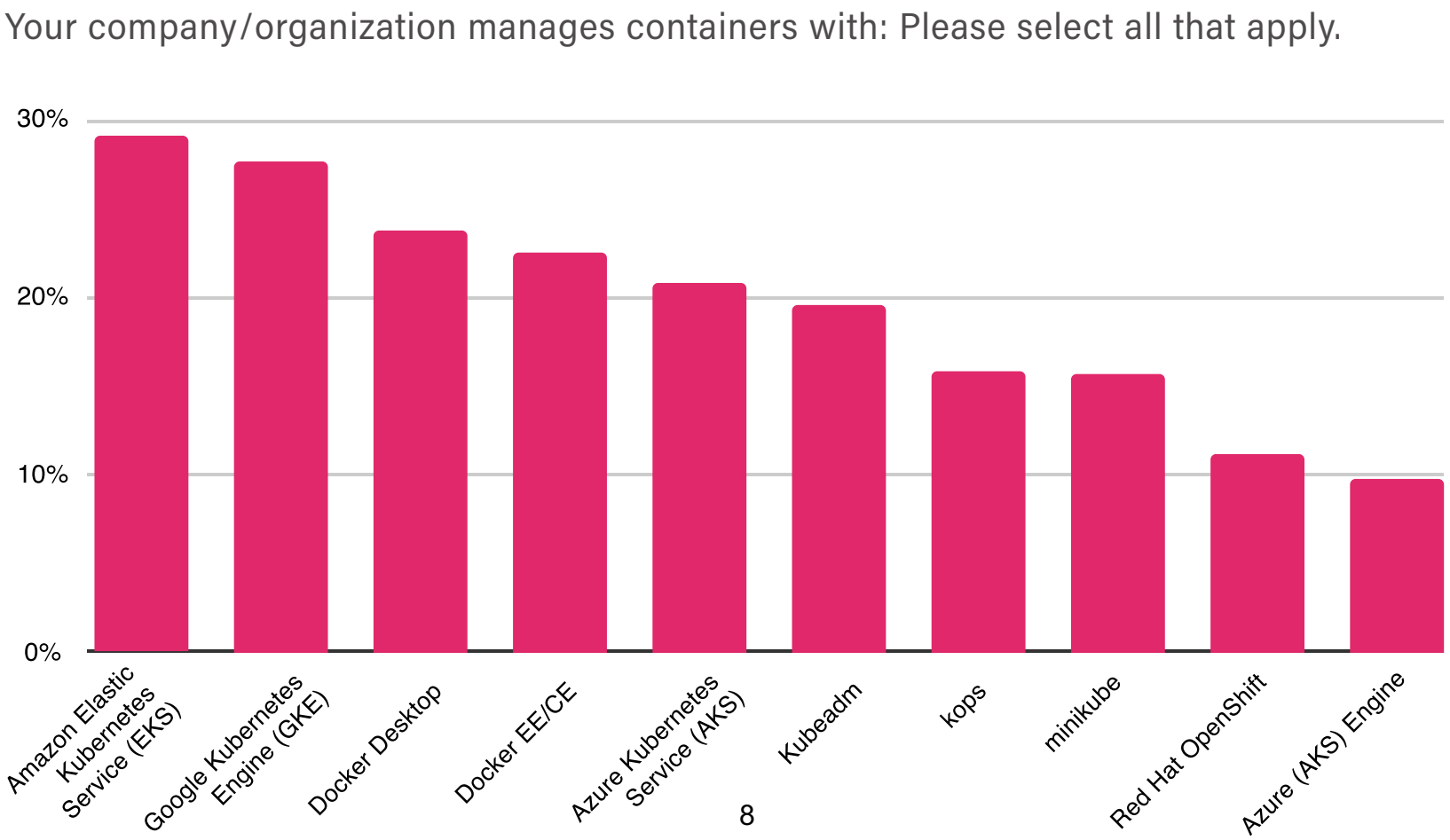
CNCF SURVEY 2019

背景

● 云原生技术趋势

CNCF SURVEY 2019

➢ 容器编排系统使用统计：

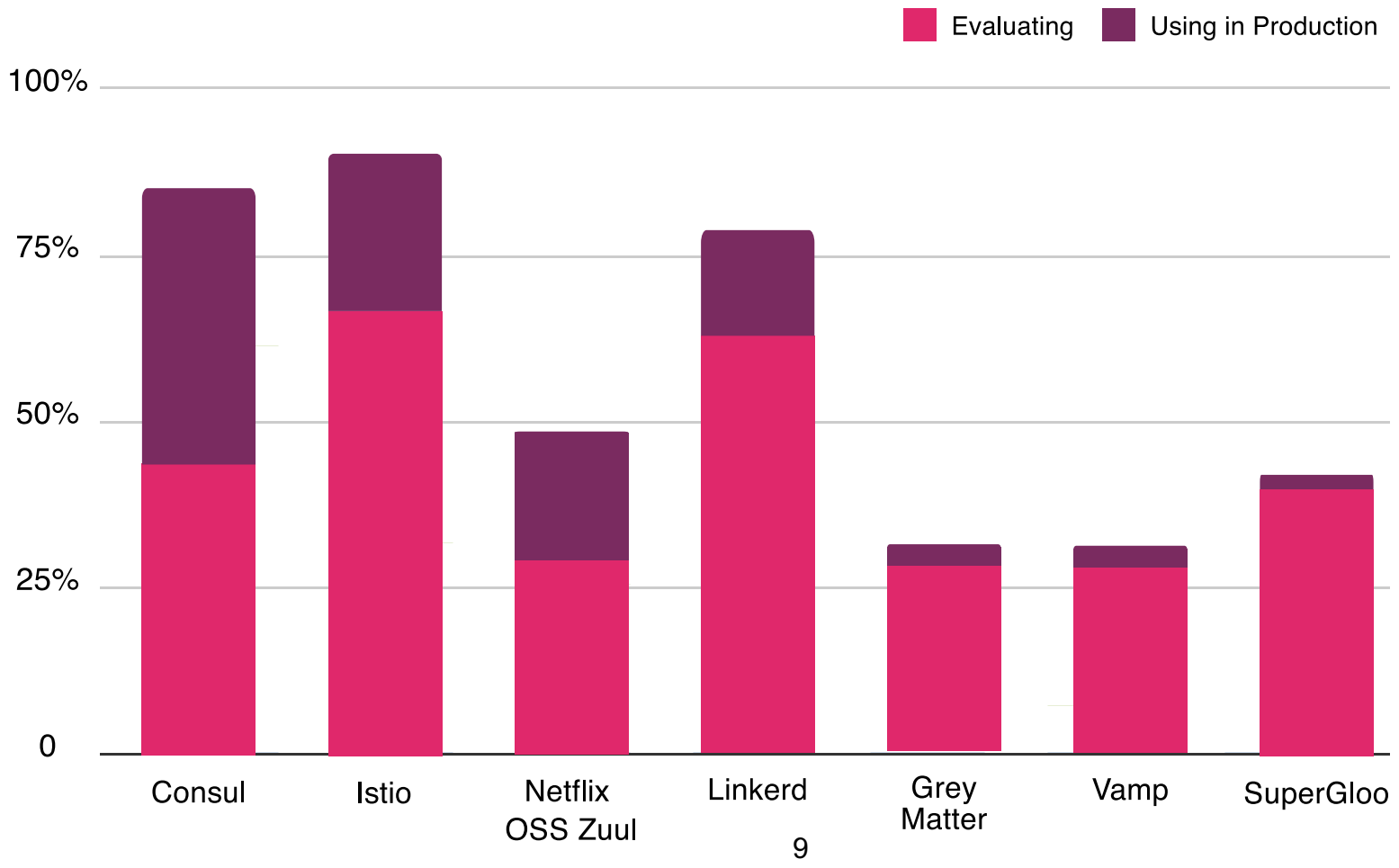


背景

- **云原生技术趋势**

CNCF SURVEY 2019

➢ 服务网格使用情况：

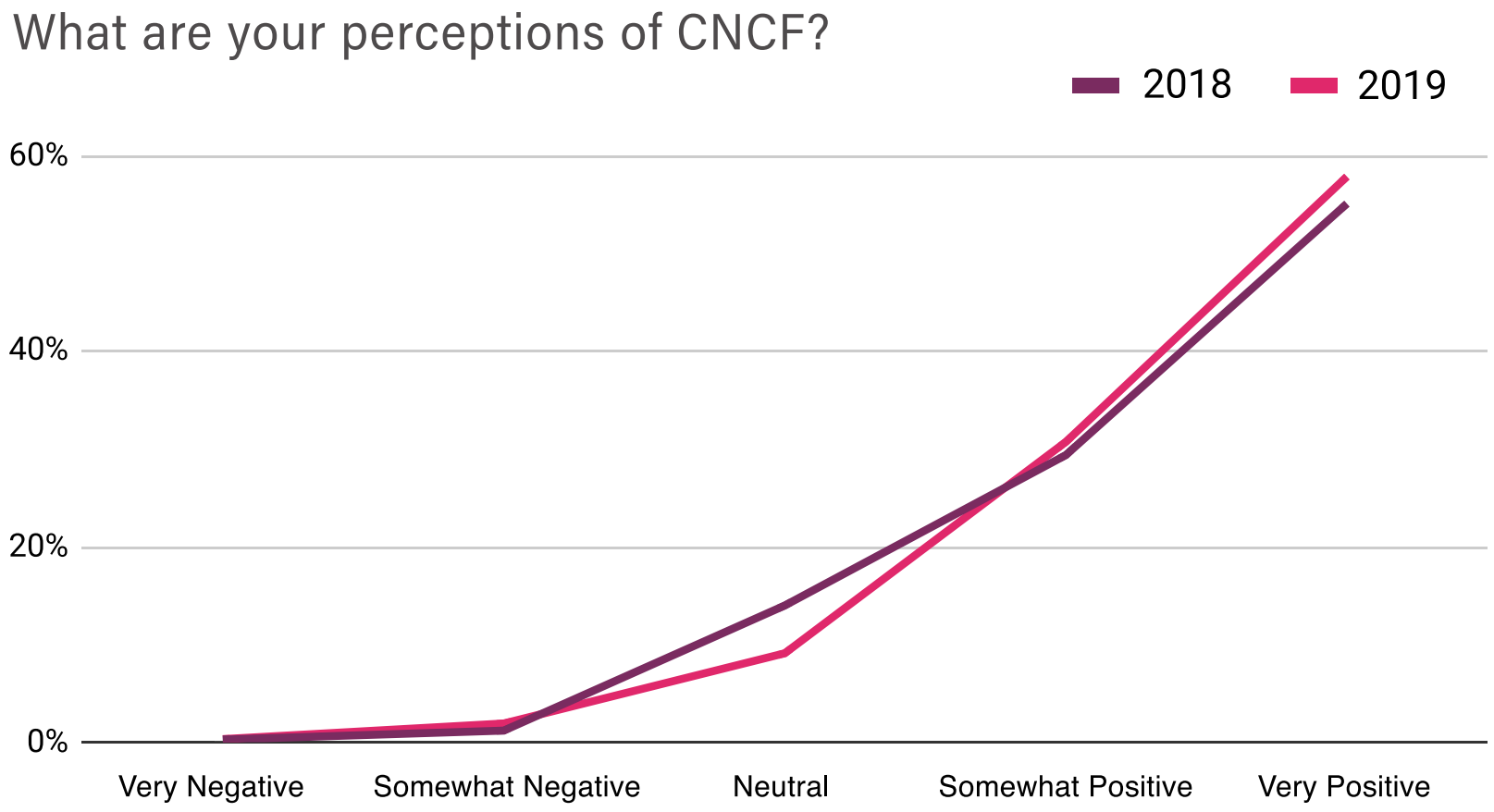


背景

- **云原生技术趋势**

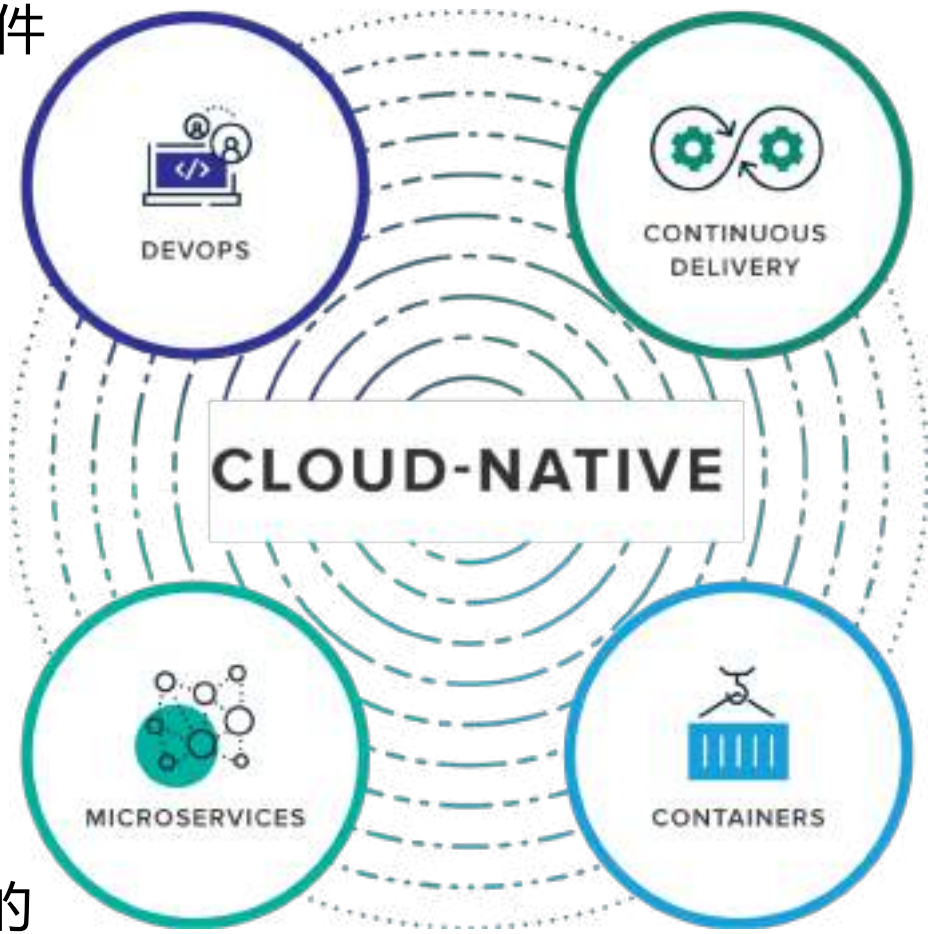
CNCF SURVEY 2019

➢ 对云原生的态度：



背景

DevOps：新的软件开发模式，加速软件的开发速度；



连续交付：连续的开发和交付，减少业务Go-To-Market的时间

微服务：小而精的软件产品，易于开发、交互和维护；

容器：基础使能技术，使开发和部署软件系统的速度加快

软件系统架构演化

- 什么是微服务

微服务架构风格是指通过开发一组**微型服务**来组成一个单一应用的方法。这每一个小的服务都自成一体，运行在自己的进程里，彼此之间通过**轻量级的机制**，通常是HTTP资源API的方式，来进行通信。这些小服务一般基于既定的业务能力范围来进行构建，通过**完全自动化部署**的机制来进行**独立部署**。

In short, the microservice architectural style is an approach to developing a single application as a suite of smallservices, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployableby fully automated deployment machinery.

—Martin Fowler & James Lewis.

https://martinfowler.com/articles/microservices.html

软件系统架构演化

- 什么是微服务

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

服务之间通过网络进行通信；

可以独立更新服务，不需要更改其他服务；

自包含；独立更新代码而不需要知道其他微服务的内部实现；

Adrian Cockcroft (VP, Cloud Architecture Strategy at AWS)

软件系统架构演化

- 传统软件

“大而全”的系统

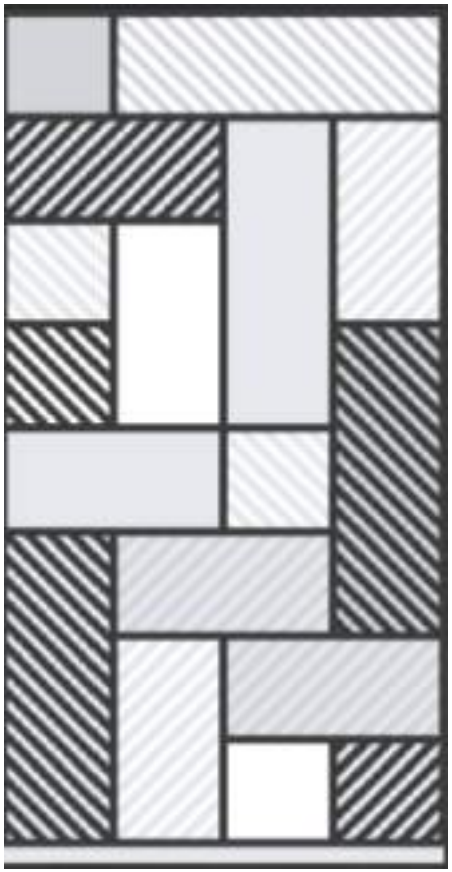
软件系统架构演化

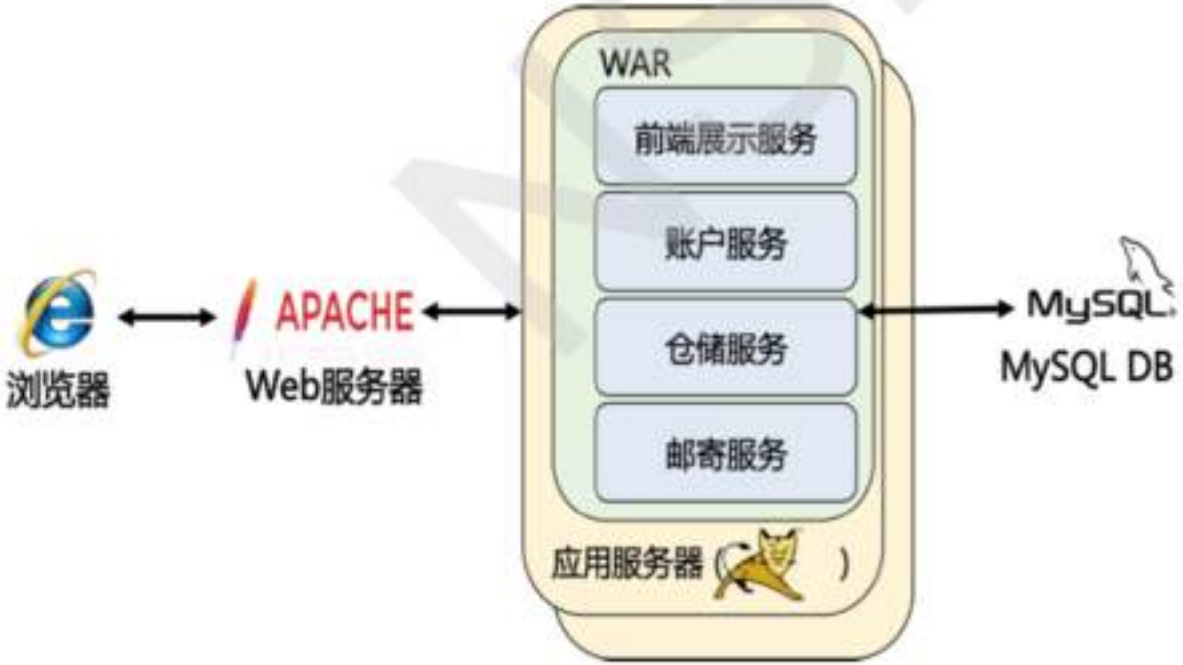
- 微服务

“小而精”的系统

软件系统架构演化

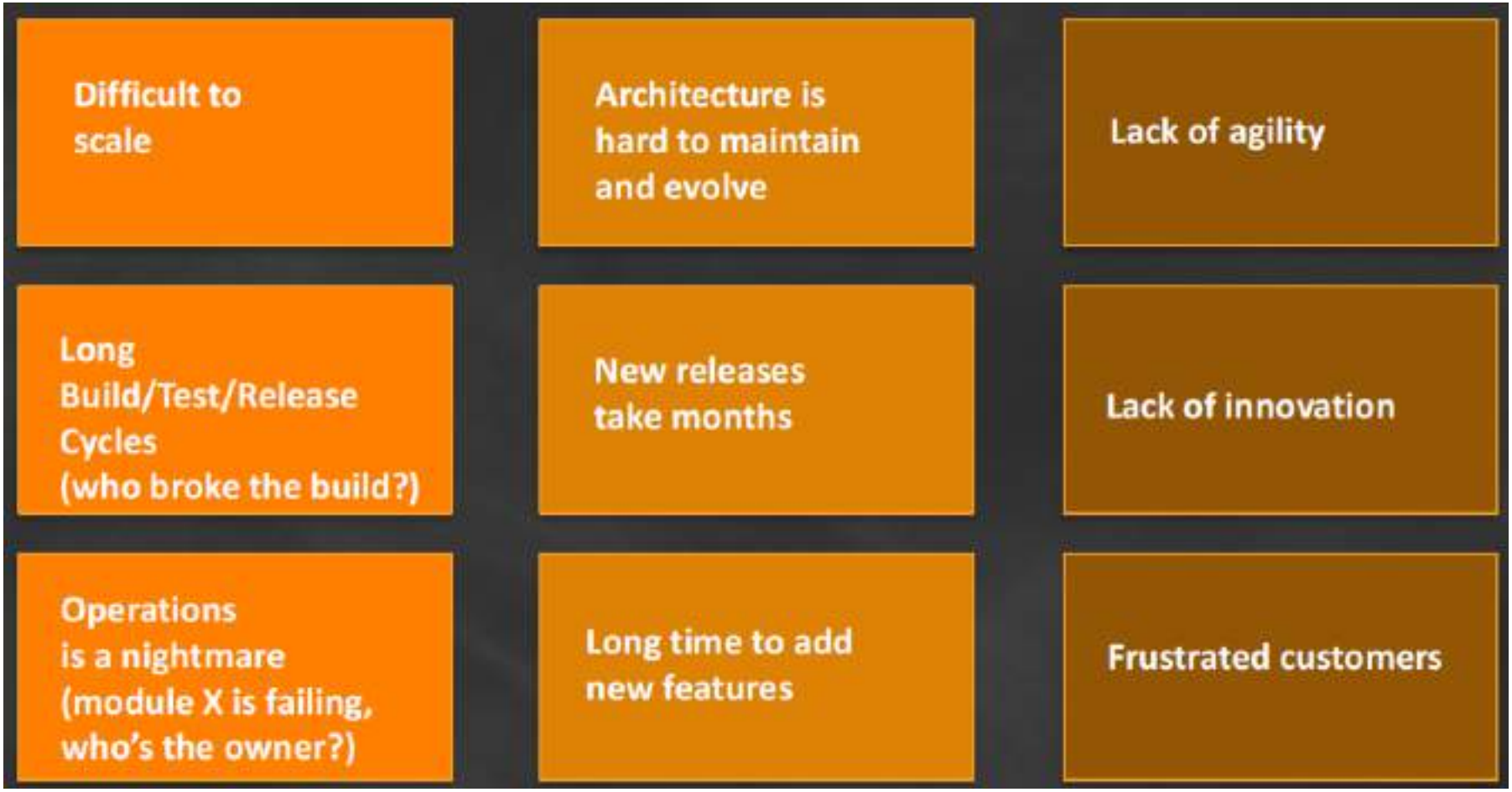
- 巨石（Monolith）应用





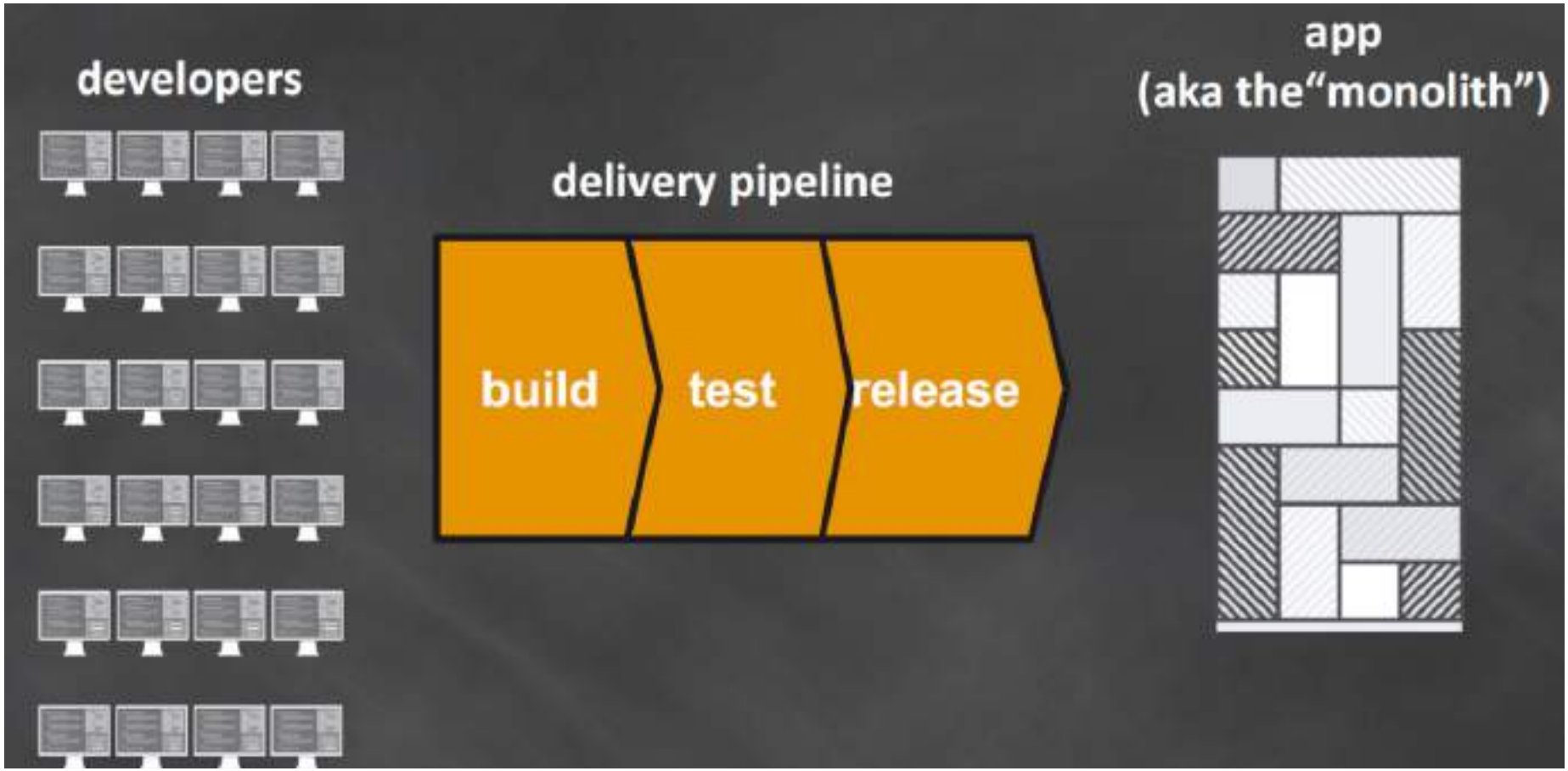
软件系统架构演化

- 巨石应用的挑战



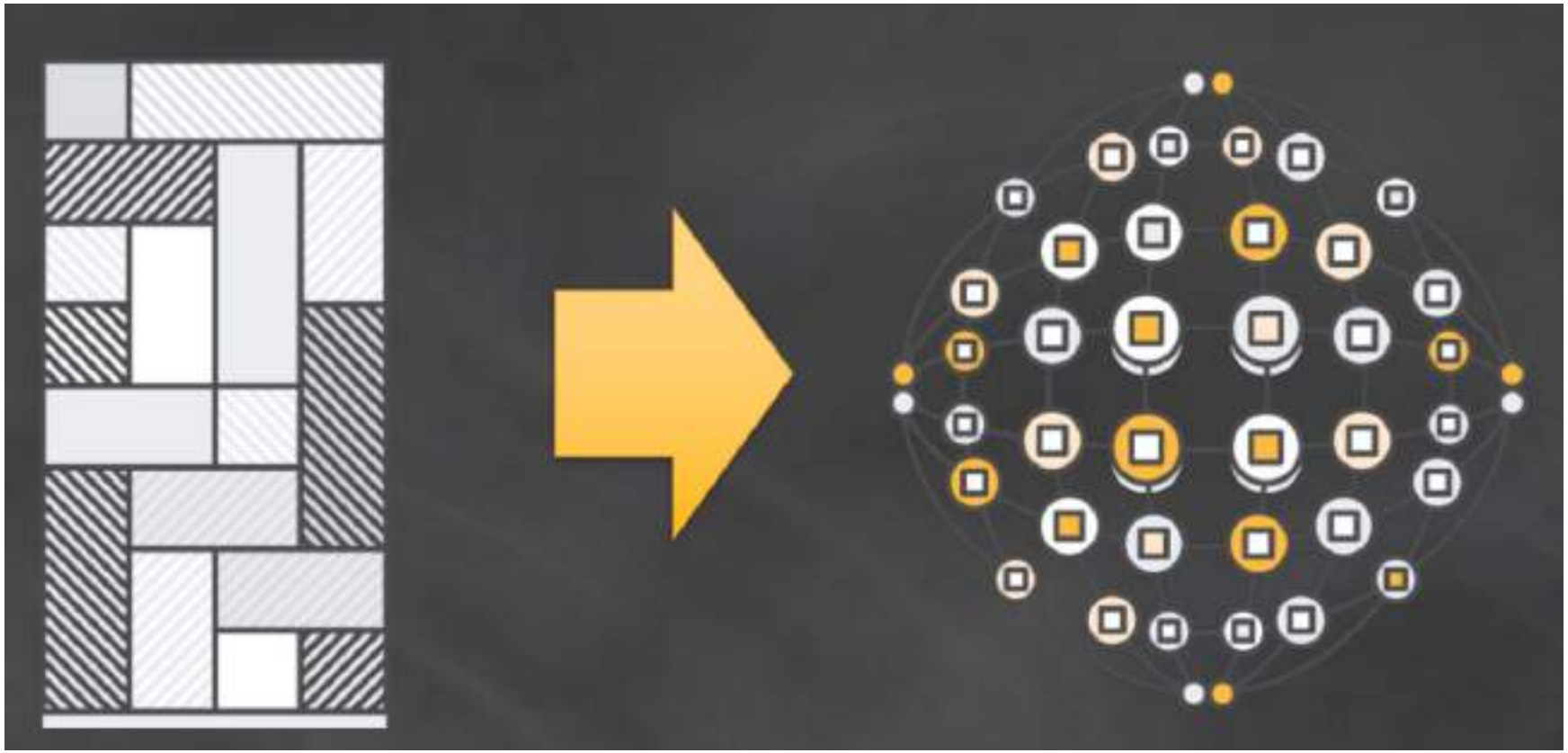
软件系统架构演化

- 巨石应用的开发周期



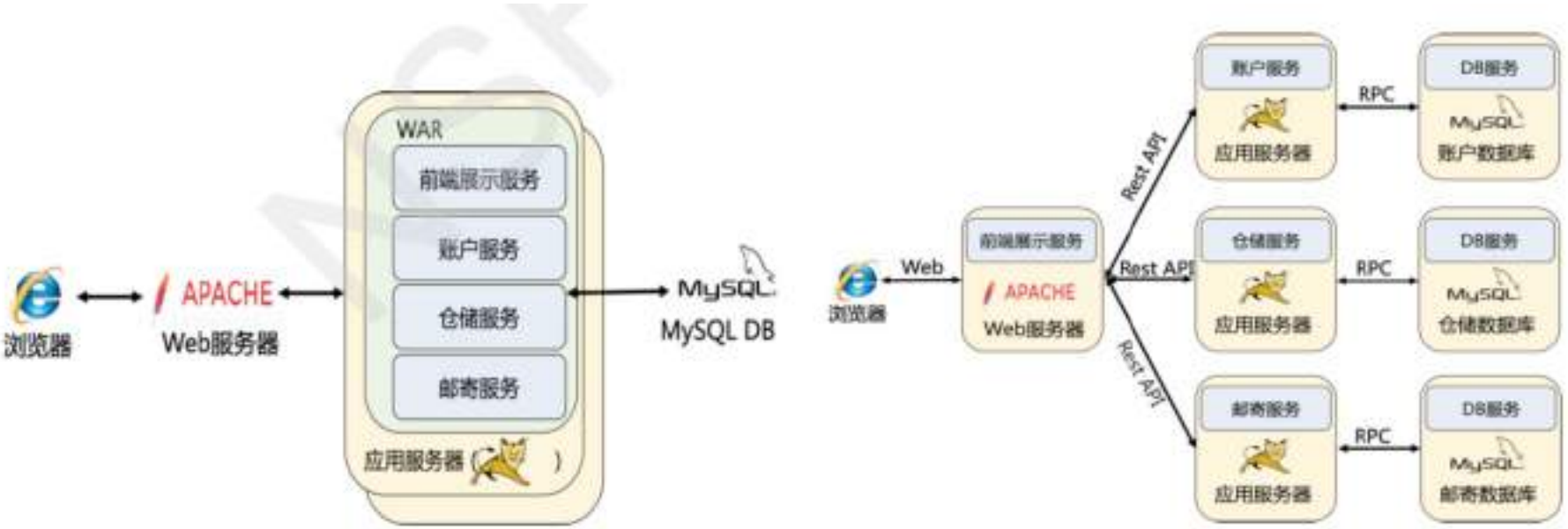
软件系统架构演化

- 巨石应用解耦



软件系统架构演化

- 巨石应用解耦



软件系统架构演化

- 微服务发展的驱动力

——新的商业模式和IT技术的驱动

CNCF: 以云为中心开发和运行的系统，整合了微服务、**DevOps**、连续交付技术的软件系统

微服务系统架构

- 如何理解“微”？

  一句话，是物理服务部署单元（物理部署组件）变小了，微服务的微指的是物理组件的微。

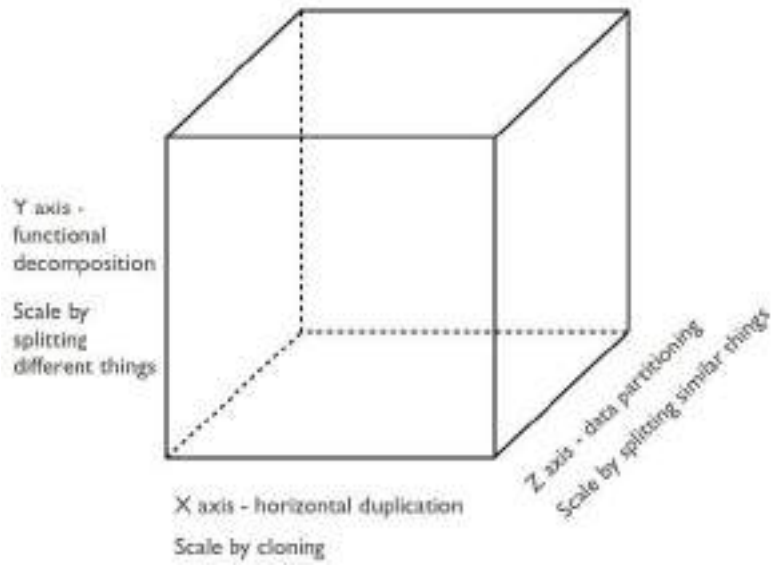
“山还是那座山”，逻辑上业务服务还是那个业务服务。但具体部署时的“服务”已经发生了变化，可以变得更小了，以便于灵活和扩展。

扩展立方体（scale cube）是一个很好地解释微服务粒度的方法

X轴（技术架构）——水平扩展。在负载均衡器后利用多进程/服务其分担负载。

Y轴（应用架构）——功能分解。按功能分解为更小的粒度。可以参考MDD、SOA以及DDD等分析设计建模方法。但部署粒度较之单体架构要小。

Z轴（数据架构）——数据分片。每个服务仅访问一个数据子集。



参考：http://microservices.io/articles/scalecube.html

微服务系统架构

- 微服务与传统架构的比较

	传统	微服务
Architecture 架构	作为单个逻辑可执行单元进行搭建（例如典型服务器端的客户-服务器-数据库三层架构）	作为一组小服务进行搭建，每个独立运行并采用轻量通信机制。
模块性	基于语言特性	基于业务能力
敏捷性	当对系统进行变更时，需要构造和部署整个应用的一个新版本	变更可以在每个服务上独立完成
扩展性	每个应用在负载均衡程序后水平扩展	每个服务在需要时独立扩展
实现	通常用一种语言编写	每个服务均可采用最适合的语言来实现
维护性	大量的代码对新开发人员是一大挑战	更少量的代码，更容易管理
交易	ACID 强一致性	BASE 最终一致性

微服务系统架构

- 采用微服务的前提

快速交付能力
云计算

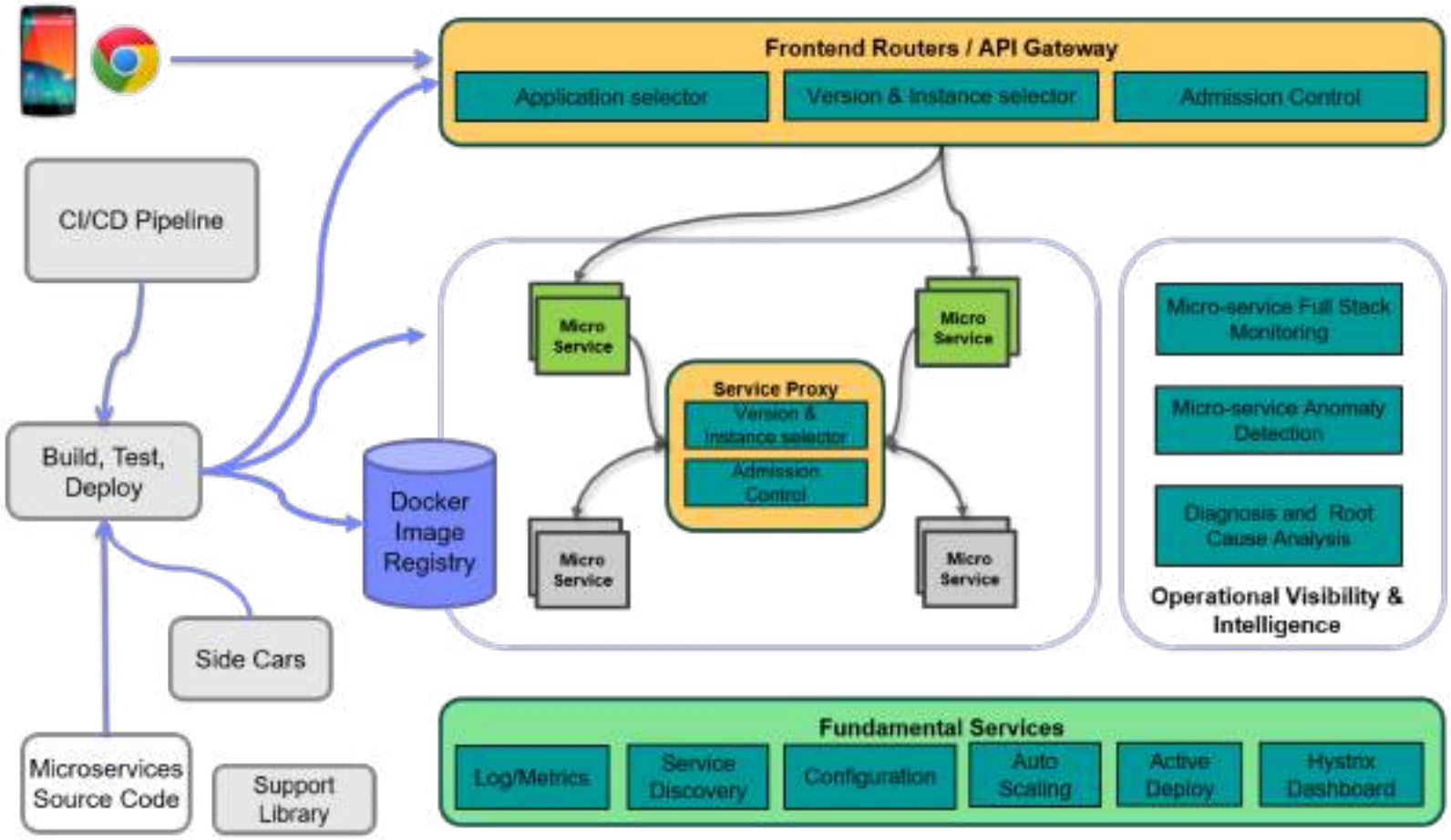
基础监控能力
应用、基础环境、（业务）

快速部署能力
Devops体系与文化

代码复杂度

微服务系统架构

- 典型的微服务架构的使能技术

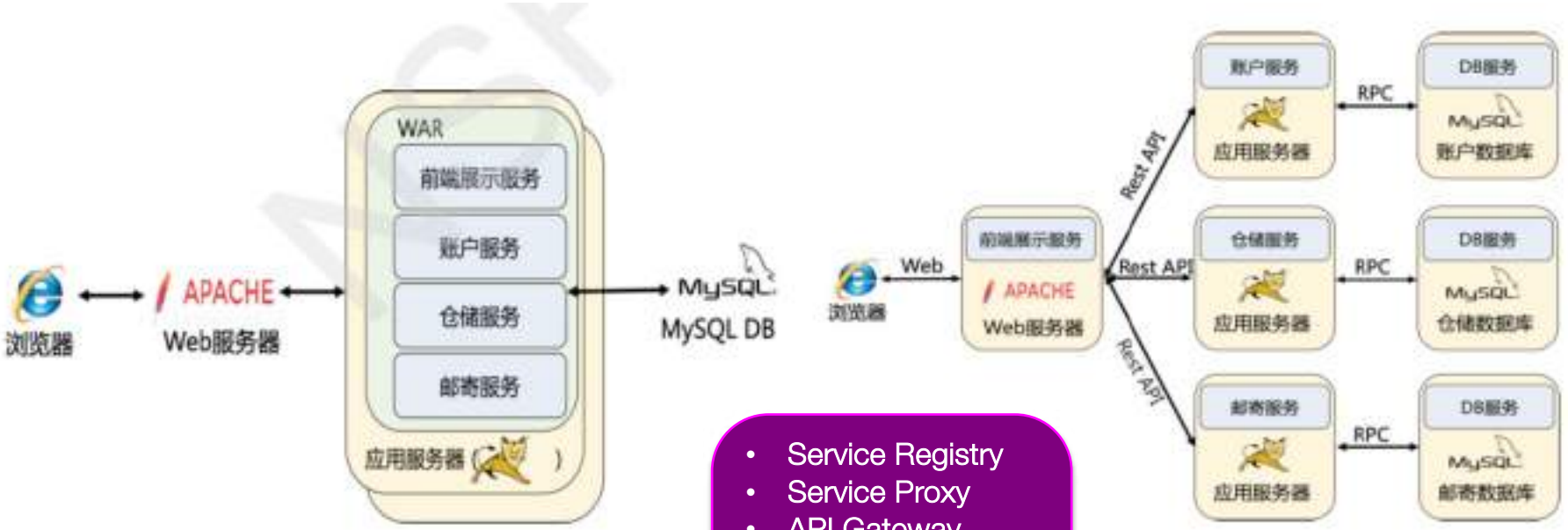


微服务系统架构

- 微服务架设的两个关键问题：支撑与应用

微服务系统架构

- 微服务架构的使能技术

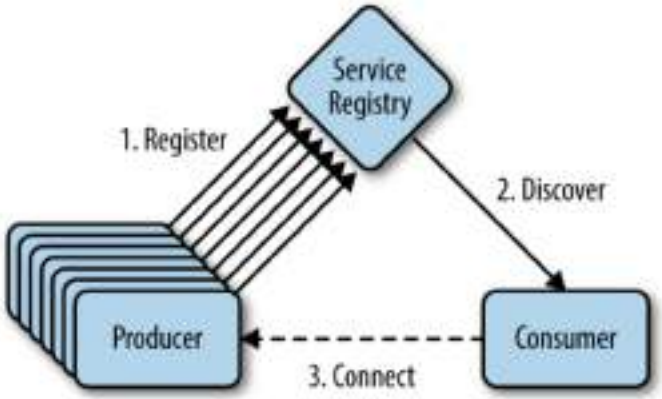


- Service Registry
- Service Proxy
- API Gateway
- Failure testing
- 'Circuit-Breaker'
- HA/DR
- CI/CD
- …

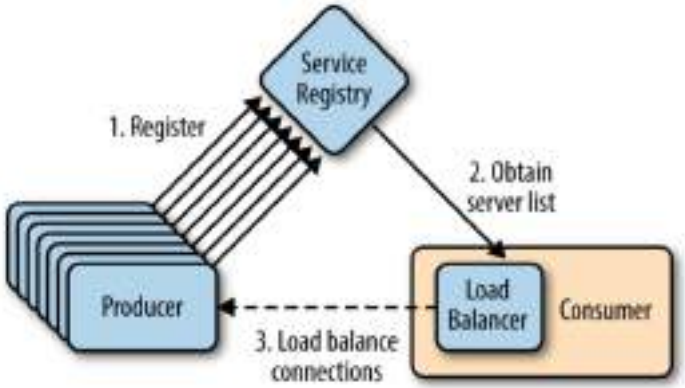
微服务系统架构

- 微服务中的基本模式

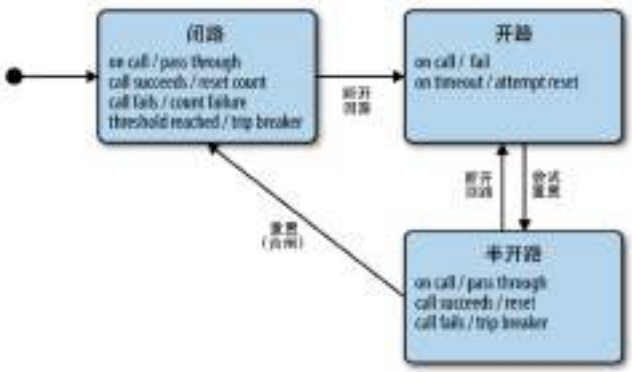
1. 服务发现



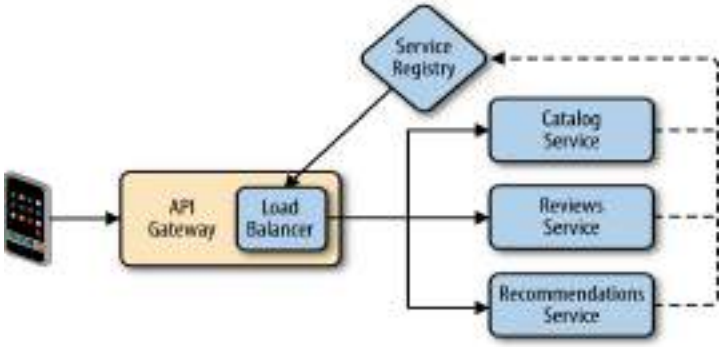
2. 路由和负载均衡



3. “断路器”和“水密舱”

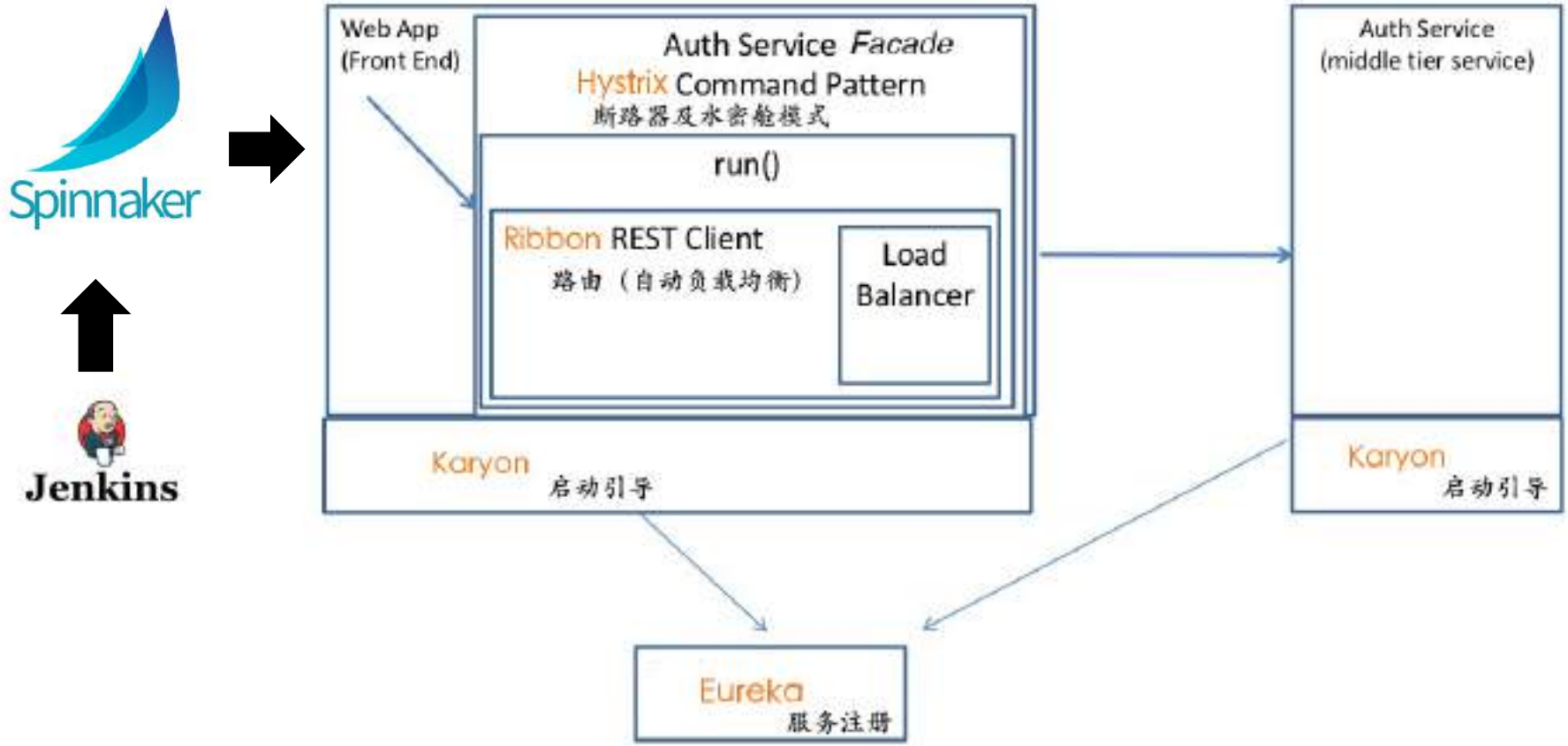


4. API 网关



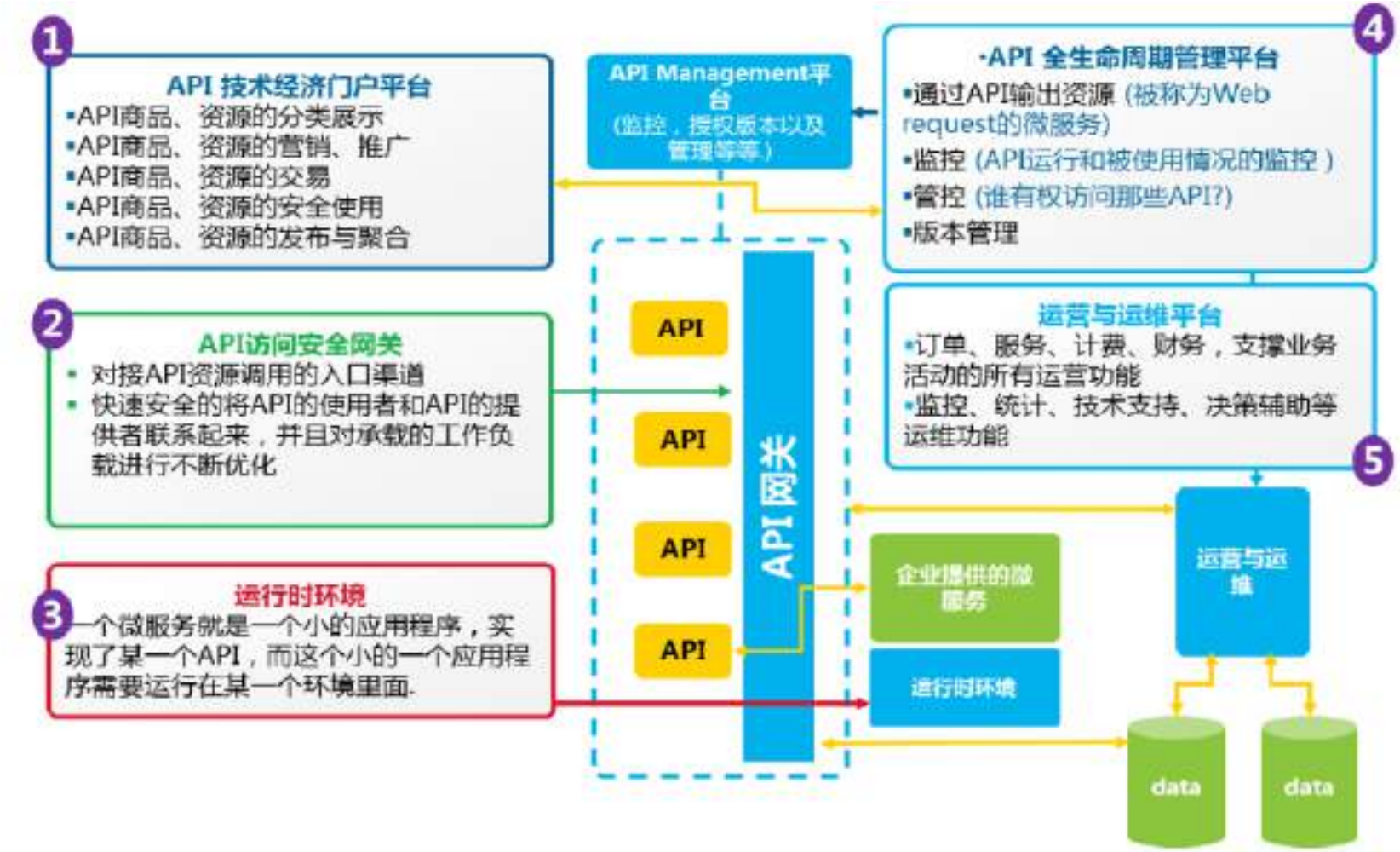
微服务系统架构

- Netflix的OSS参考架构



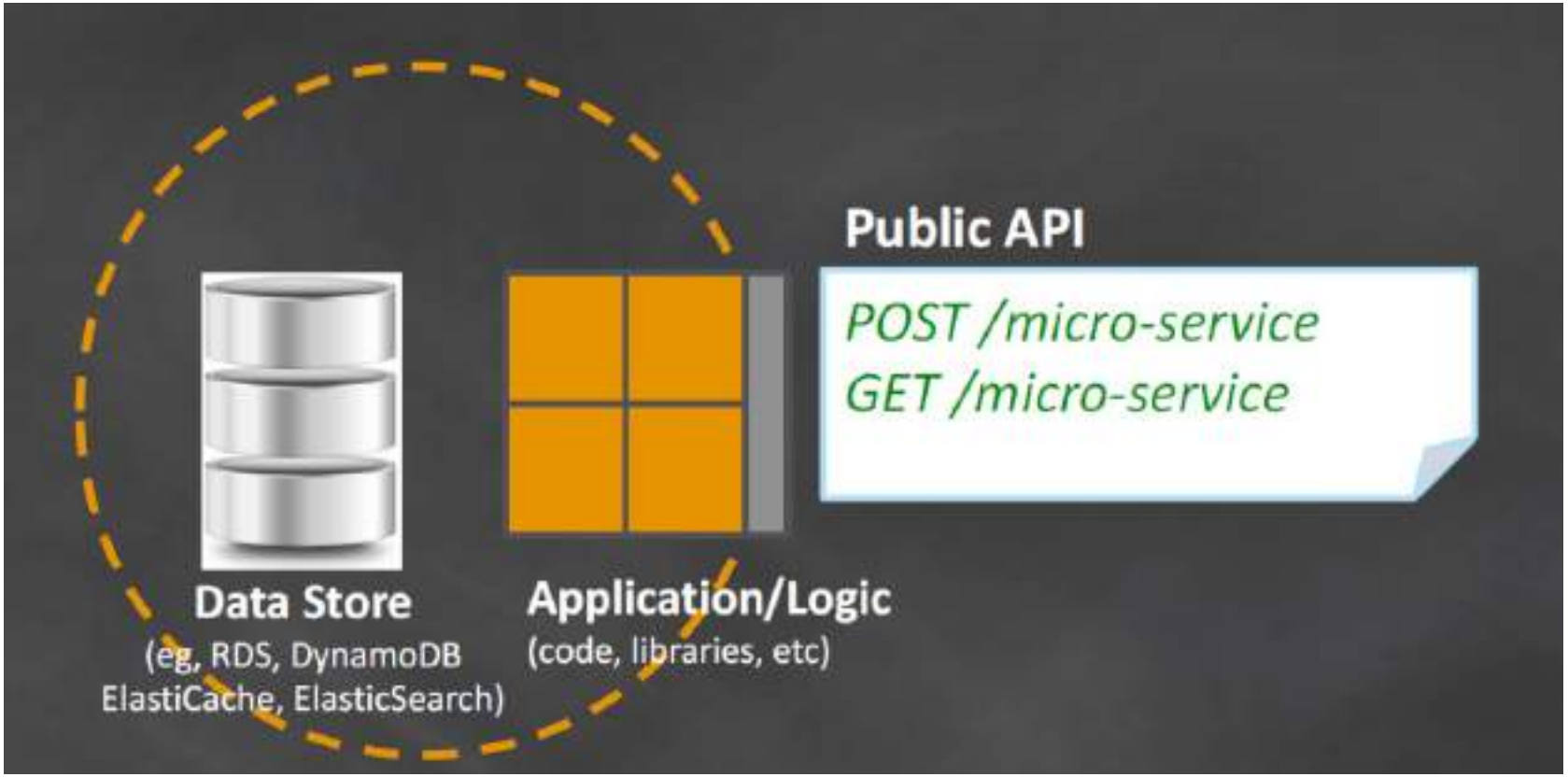
微服务系统架构

- 开放API平台



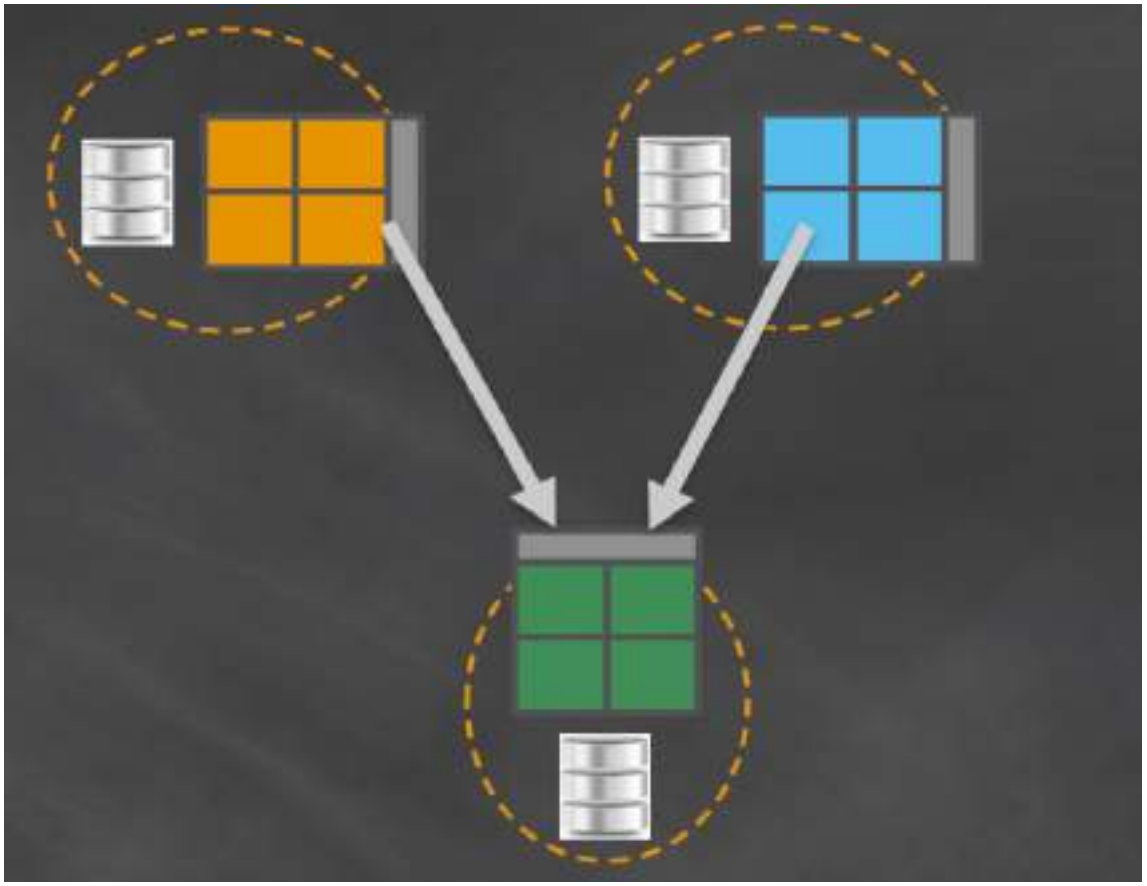
微服务系统构建

- 微服务构成要素



微服务系统构建

- 系统解耦

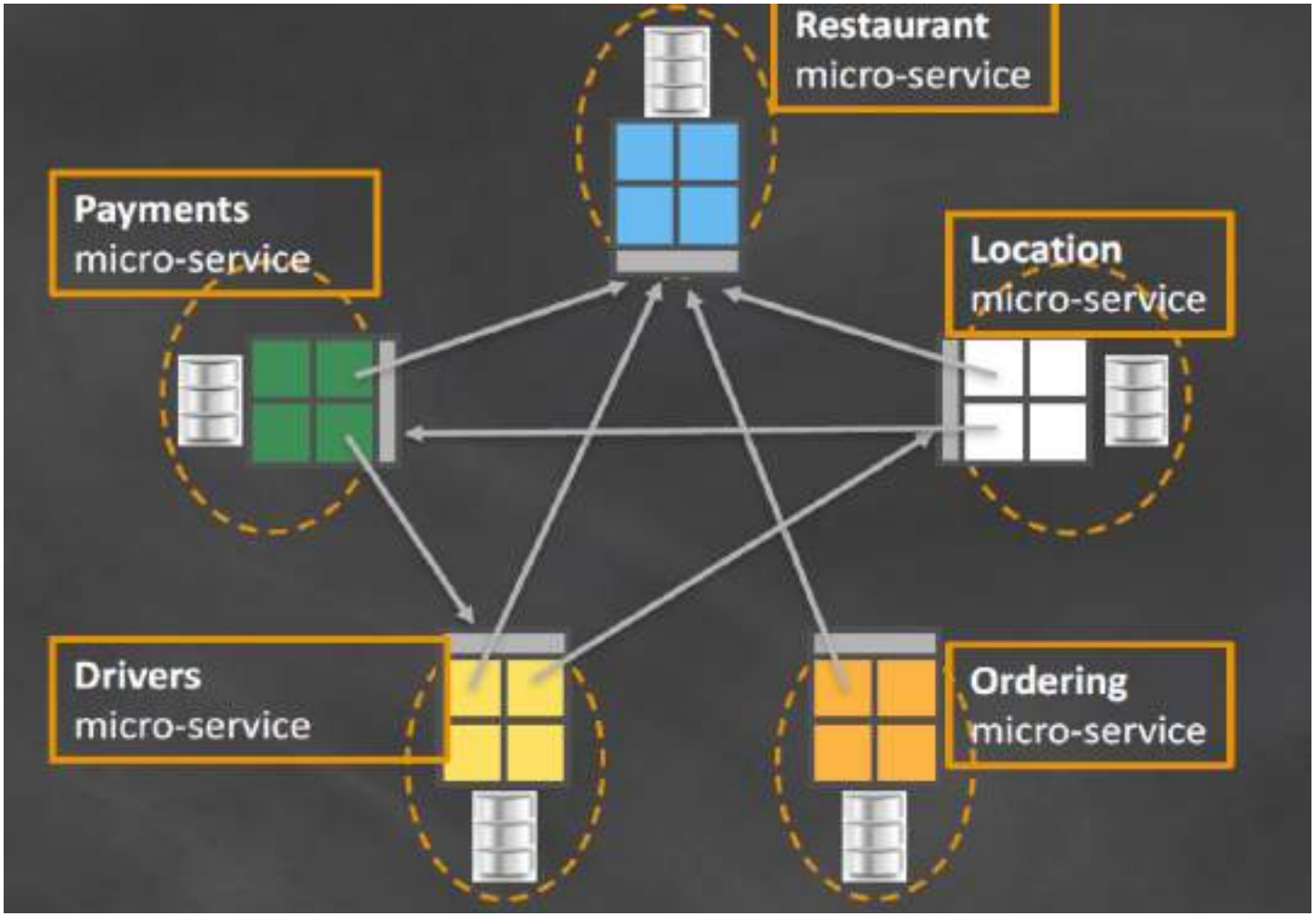


- 最小化依赖关系；
- 较早划分“粘性”部分；
- “垂直解耦”，较早释放数据；
- 解耦变化频繁但是对业务影响比较大的部分；
- 解耦系统能力而不是代码；
- 先进行宏观解耦，然后微观；
- 原子性演化；

https://martinfowler.com/articles/break-monolith-into-microservices.html

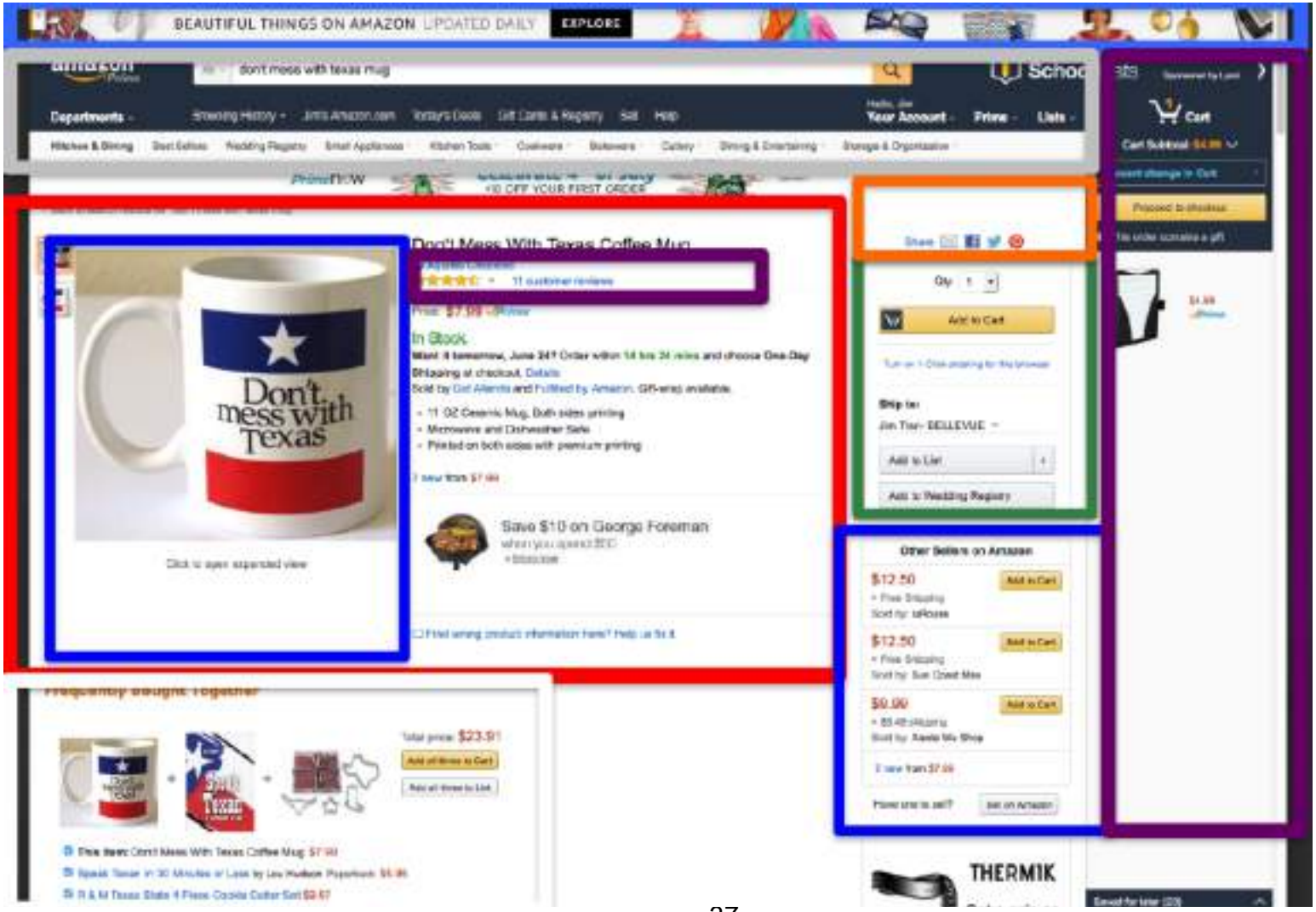
微服务系统构建

- 微服务生态



微服务系统构建

- 微服务生态



微服务系统构建

- 微服务开发实现

Eclipse Microprofile

Github MicroBuilder

Istio (ServiceMesh)

NETFLIX

Netflix OSS

spring

Spring Cloud

微服务实战

Spring Cloud

微服务系统构建

- 微服务系统部署

IBM Microservice Builder

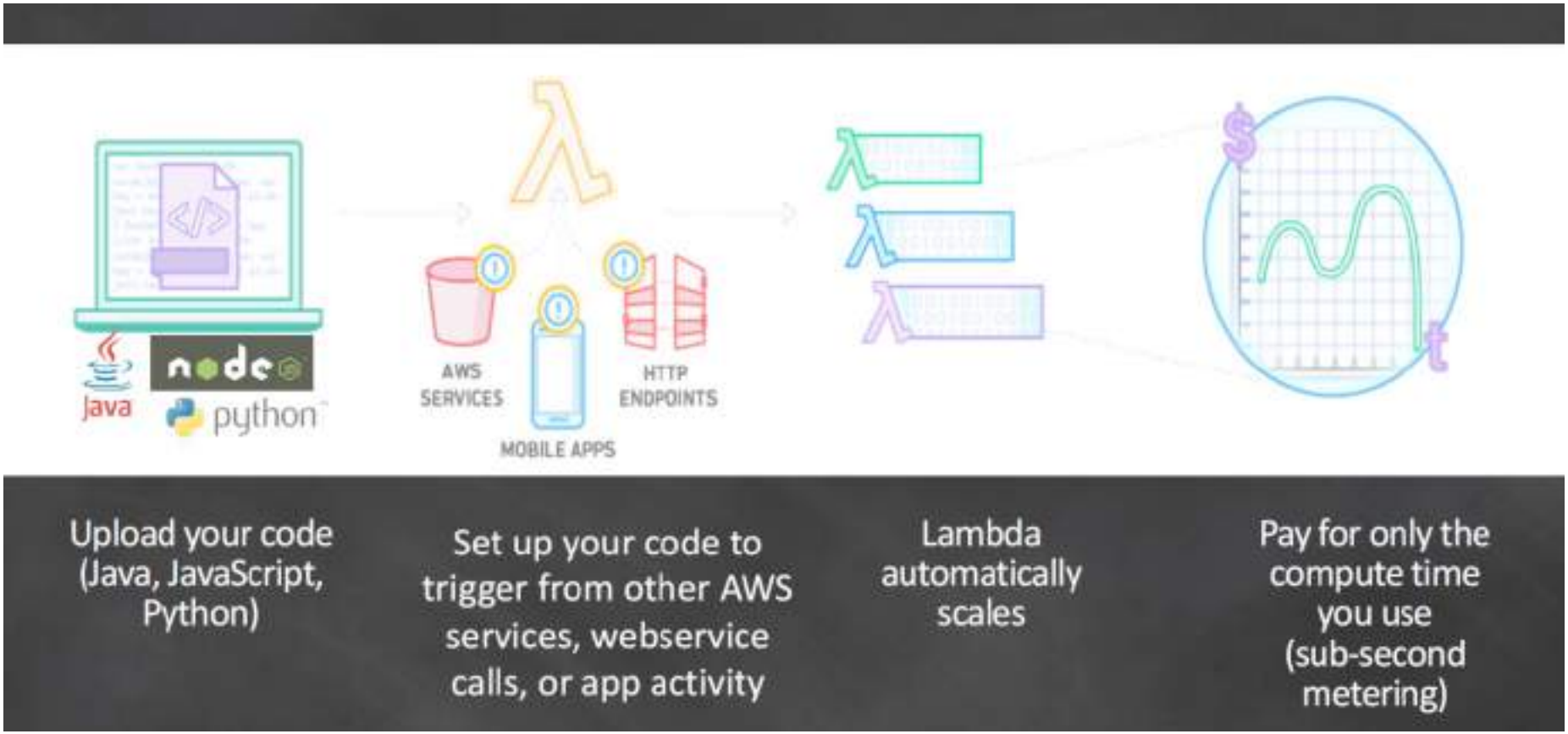
IBM MicroClimate

Netflix OSS

Amazon Lambada

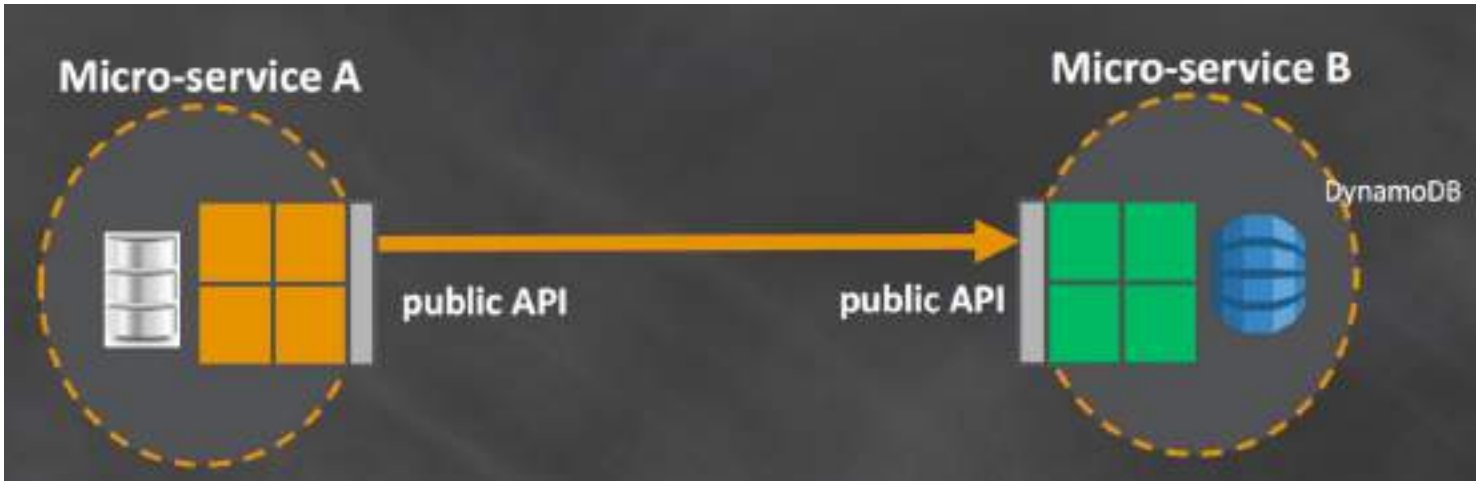
微服务系统构建

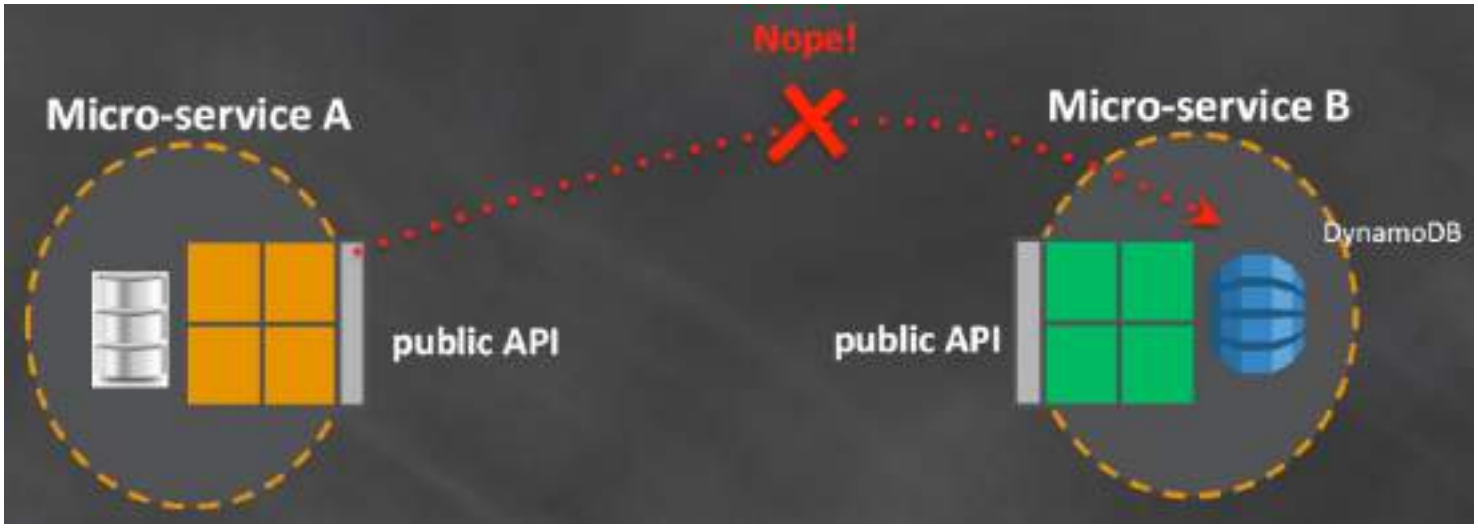
- 微服务系统部署(Lambda)



微服务系统构建核心原则

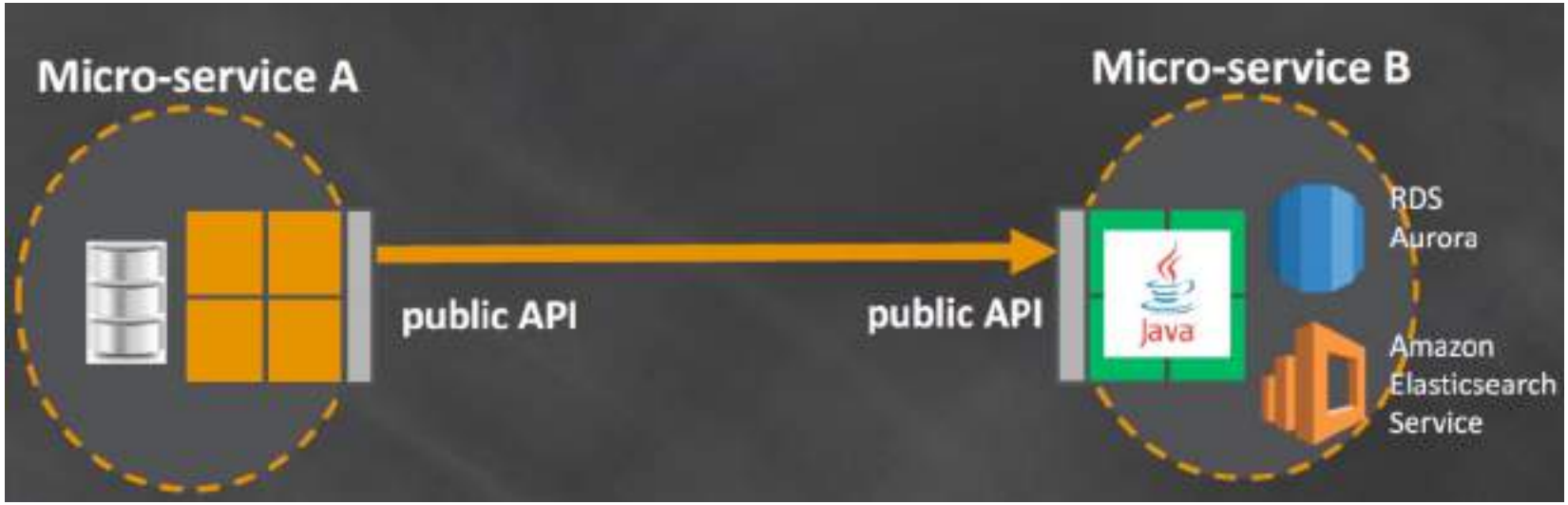
- 服务之间只通过公共API访问

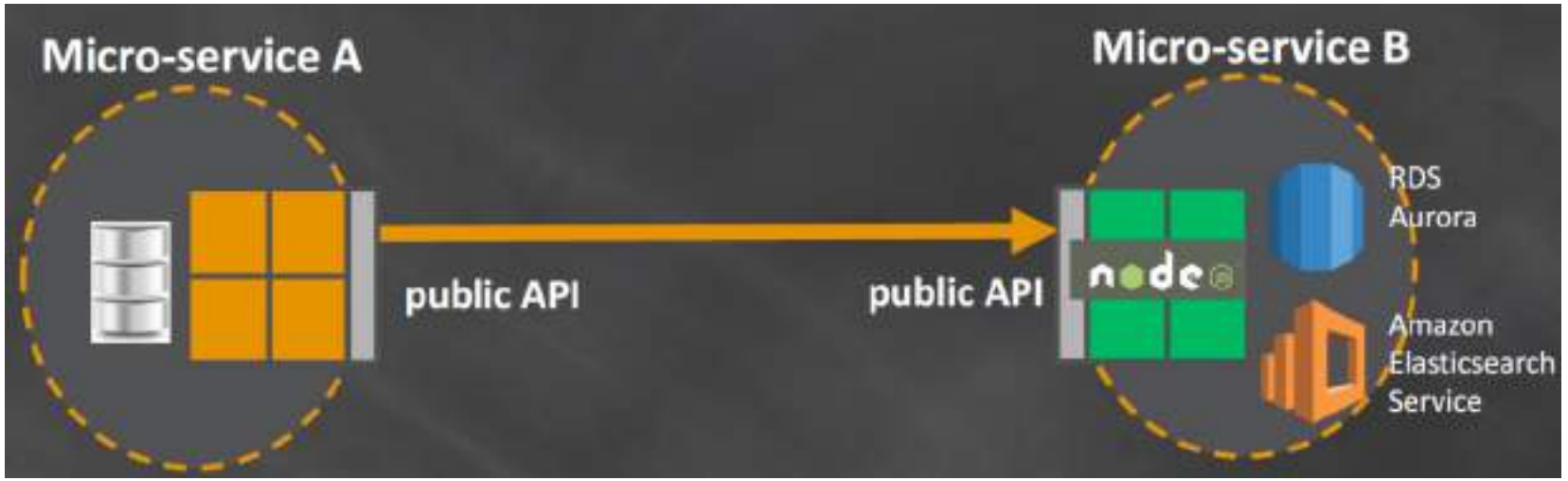




微服务系统构建核心原则

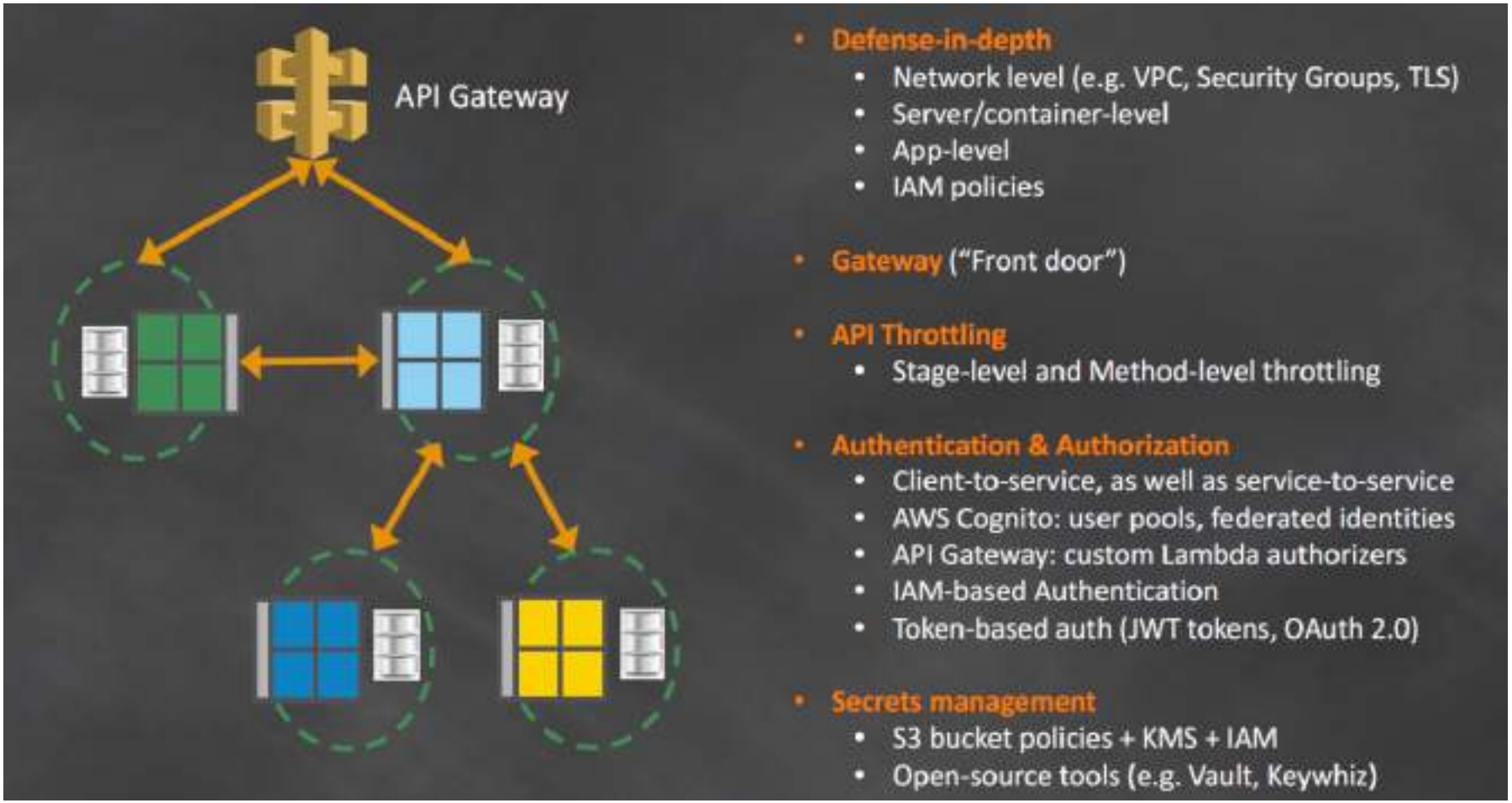
- 利用合适的工具（拥抱多语言编程）





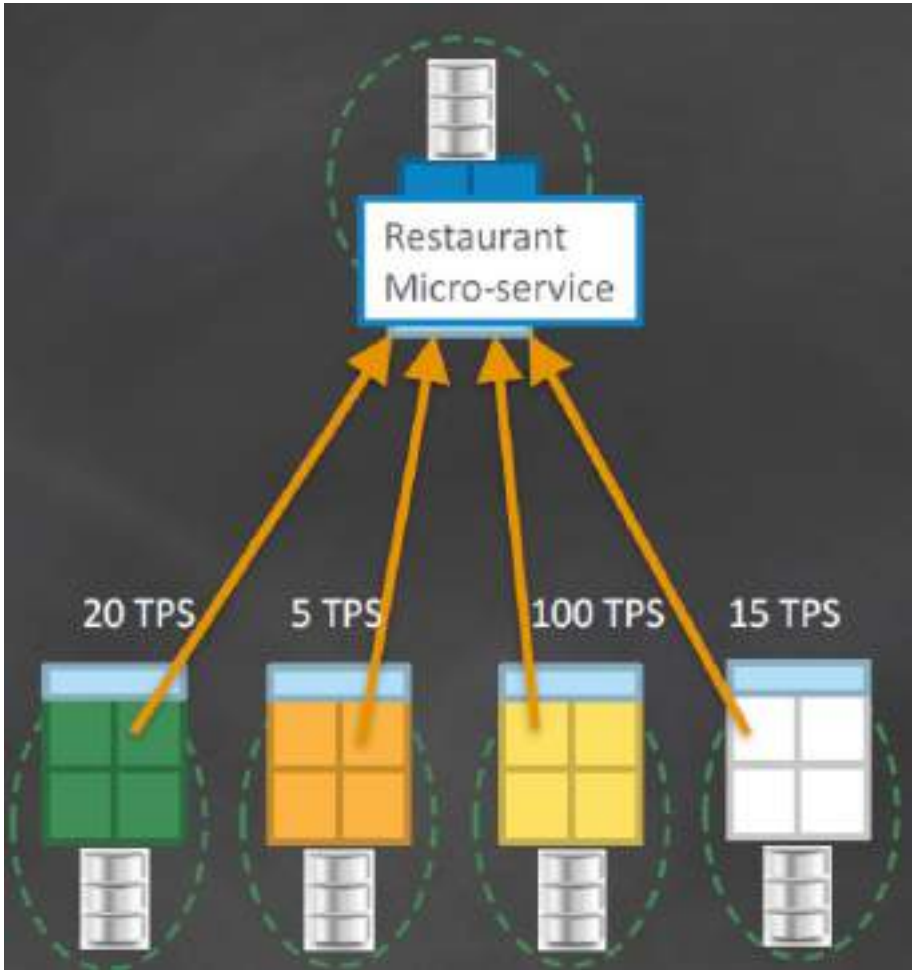
微服务系统构建核心原则

- 注重安全性



微服务系统构建核心原则

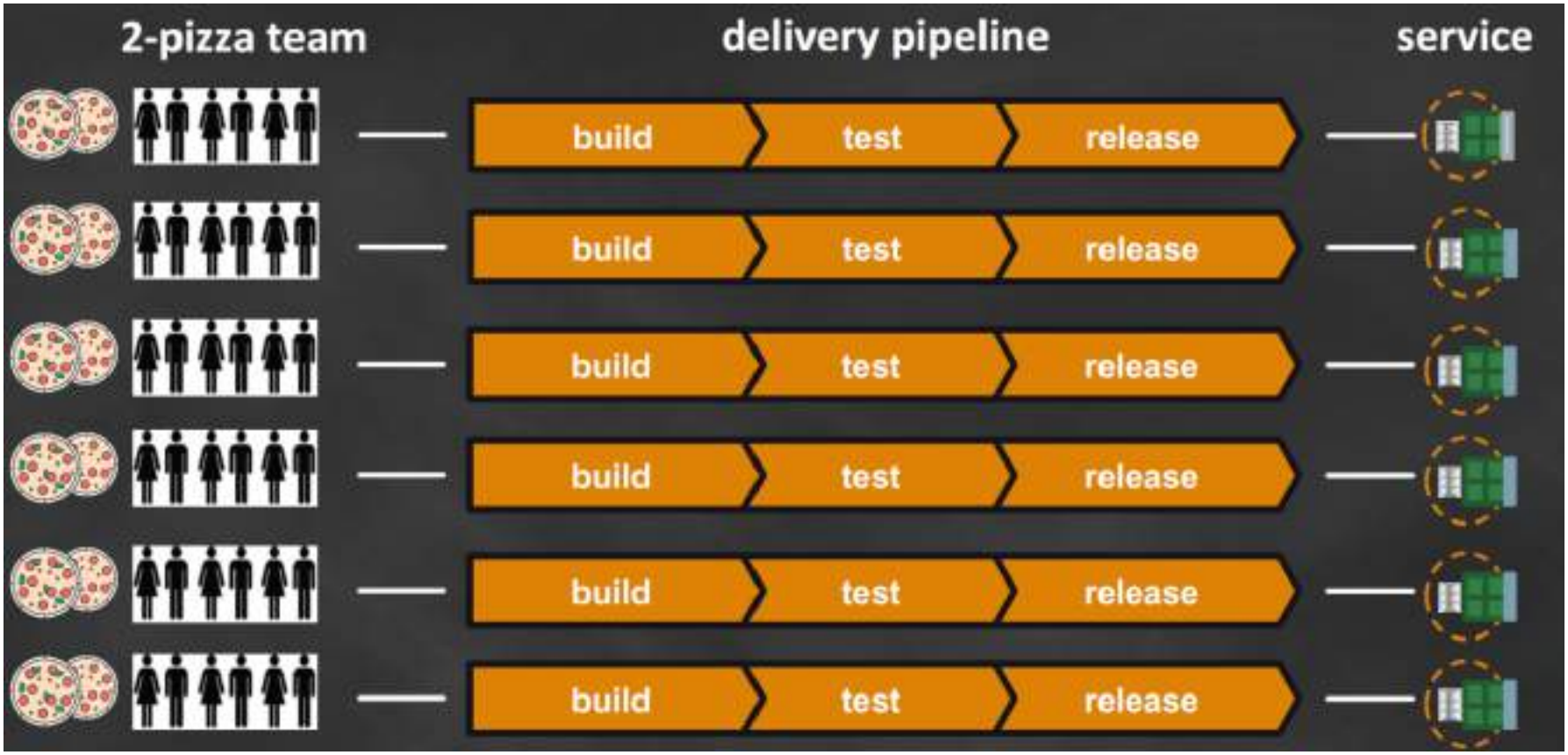
- 保障服务SLA



运维软件

微服务系统构建核心原则

- 高度自动化



微服务系统构建核心原则

- 原则总结

1. Rely only on the public API
   - Hide your data
   - Document your APIs
   - Define a versioning strategy

2. Use the right tool for the job
   - Container journey? (use ECS)
   - Polyglot persistence (data layer)
   - Polyglot frameworks (app layer)

3. Secure your services
   - Defense-in-depth
   - Authentication/authorization

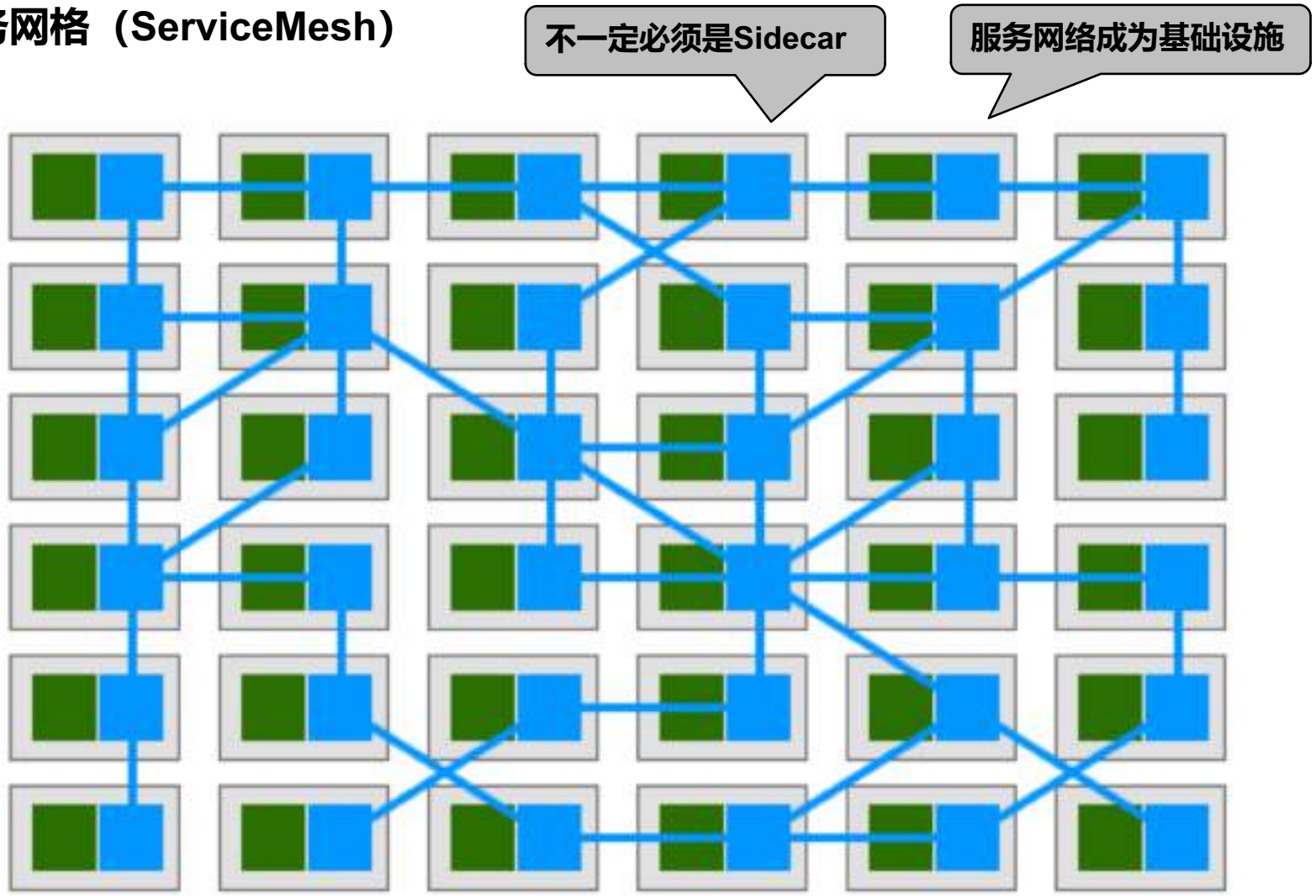
4. Be a good citizen within the ecosystem
   - Have SLAs
   - Distributed monitoring, logging, tracing

5. More than just technology transformation
   - Embrace organizational change
   - Favor small focused dev teams

6. Automate everything
   - Adopt DevOps

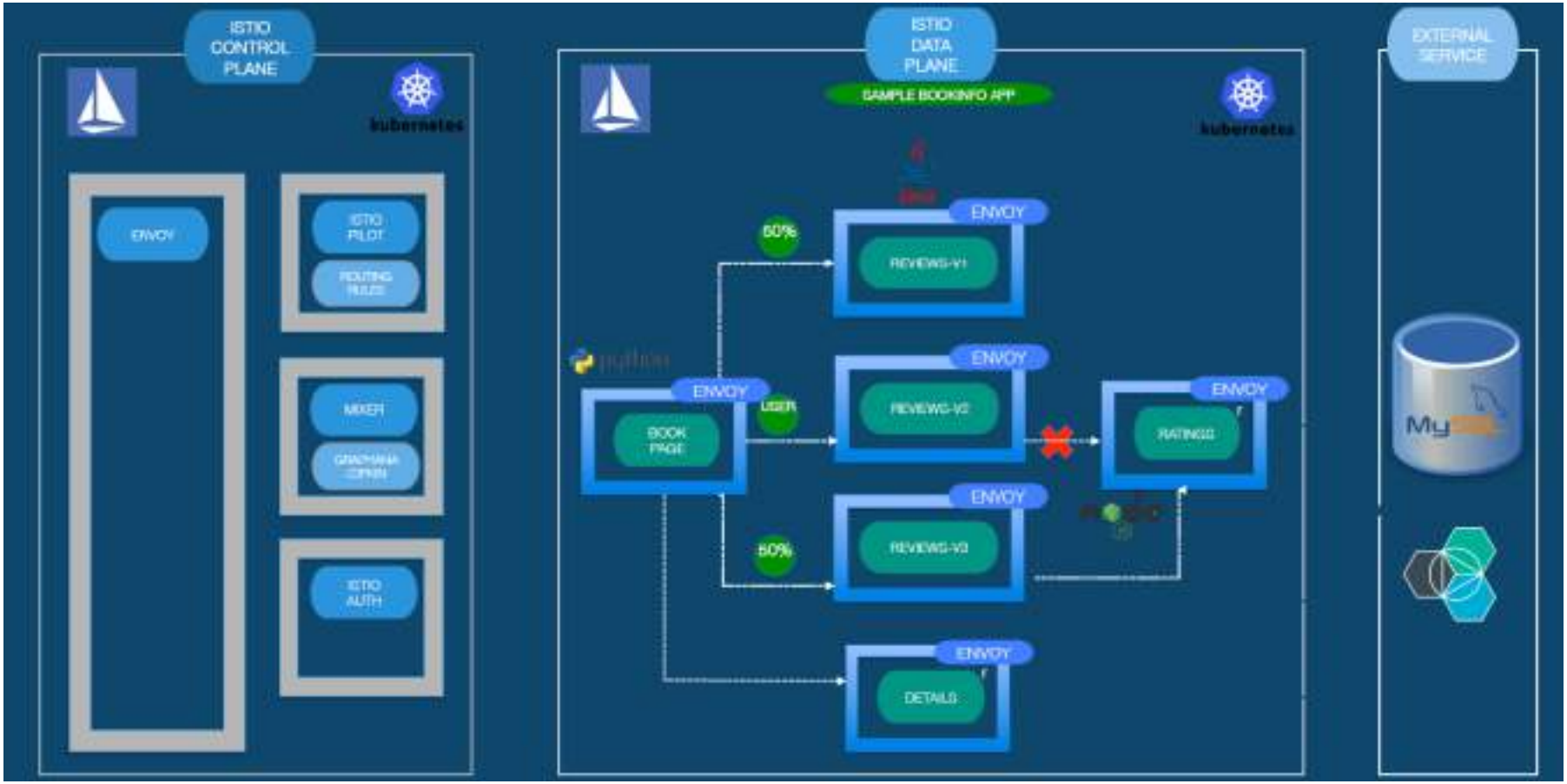
微服务系统扩展

服务网格（ServiceMesh）



微服务系统扩展

服务网格（ServiceMesh）



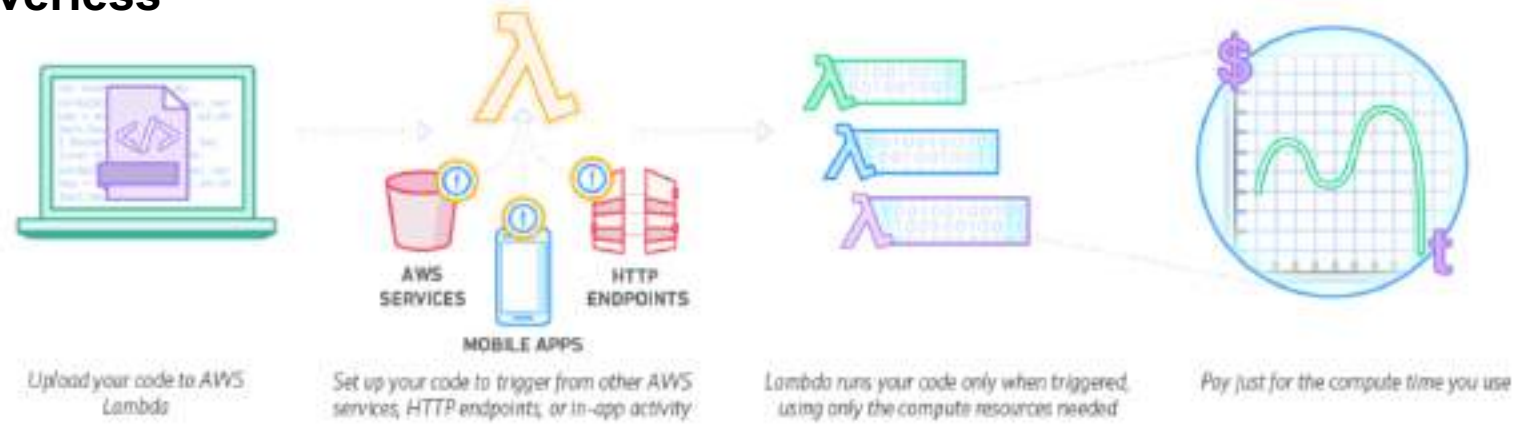
Linkerd

CONDUIT

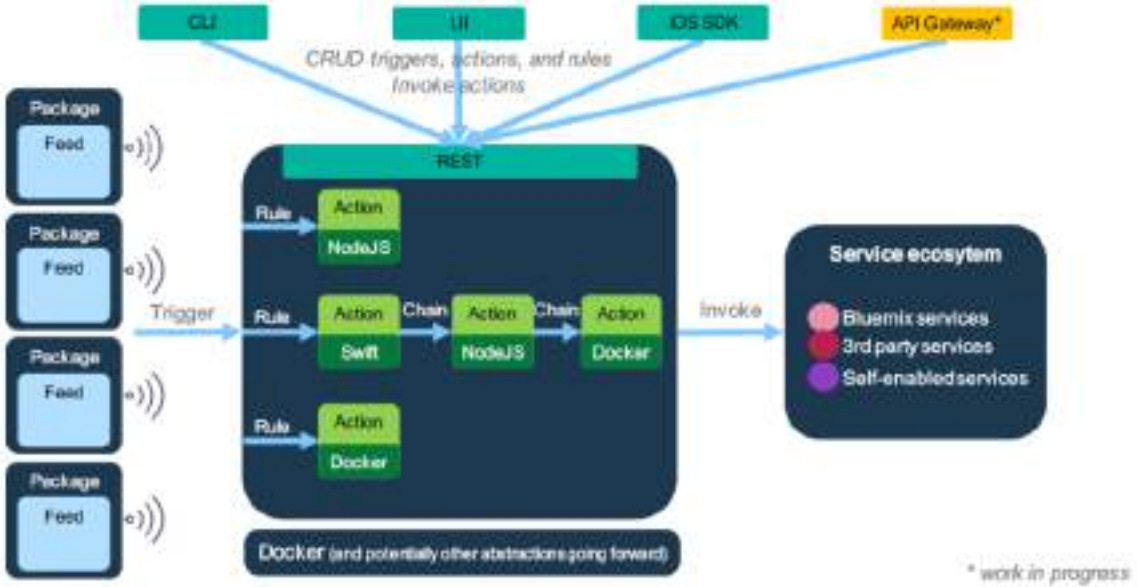
Istio

微服务系统扩展

Serverless

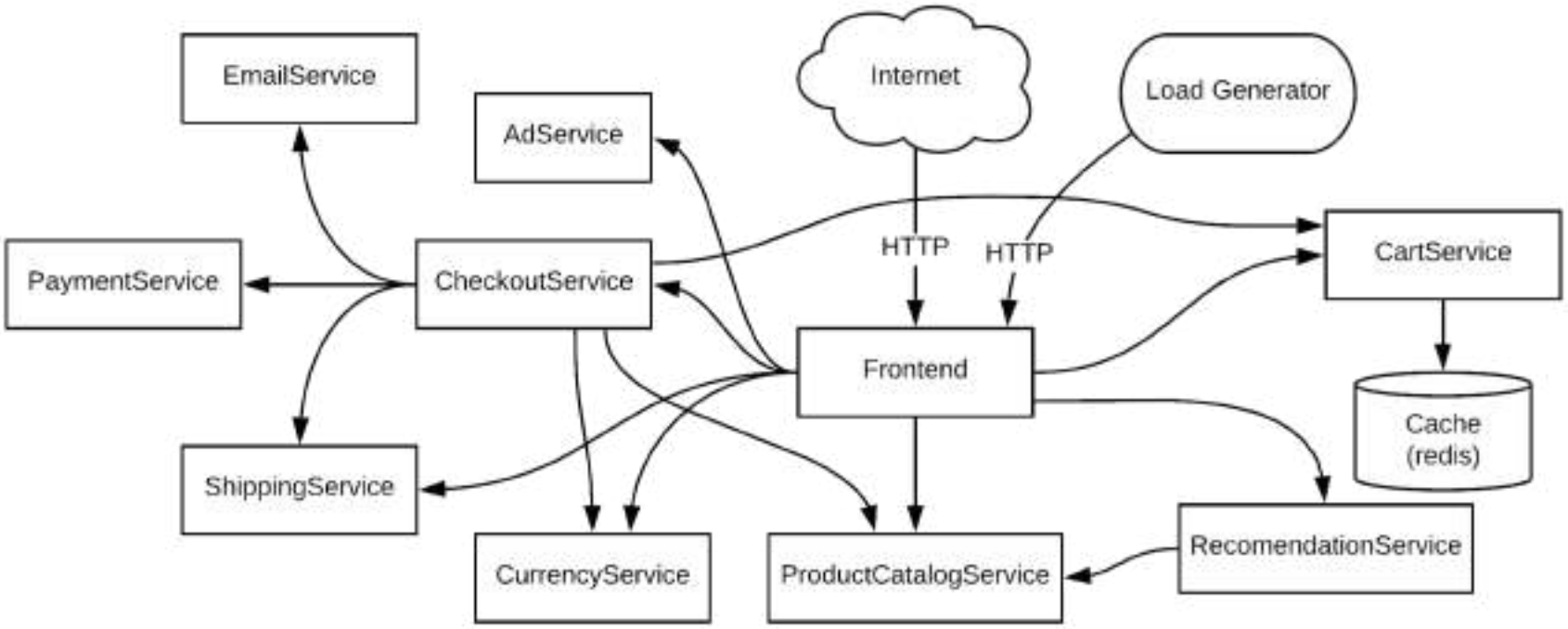


Amazon Lambda



OpenWhisk

Demo

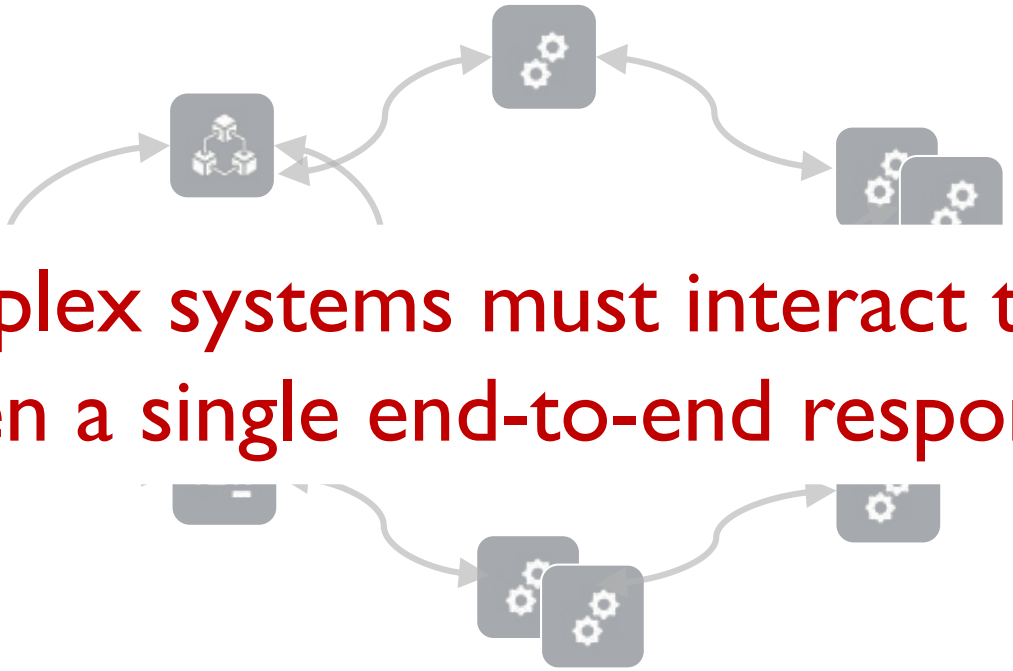


Hipster-shop Benchmark

http://33.33.33.233:32689/jaeger/dependencies;
http://33.33.33.233:24934/kiali/console/overview?duration=60&refresh=15000;
http://33.33.33.233:30138/?orgId=1;

Tracing

Debugging microservices is hard!



Many complex systems must interact to produce even a single end-to-end response

Tracing

End-to-end distributed tracing

Can't be answered by out-of-the-box Span-level tracing

End-to-end traces provide visibility

Which service contributes most to the delay experienced by the slowest 1-2% of the requests?

How much time did high priority requests take at a particular service?

Tracing

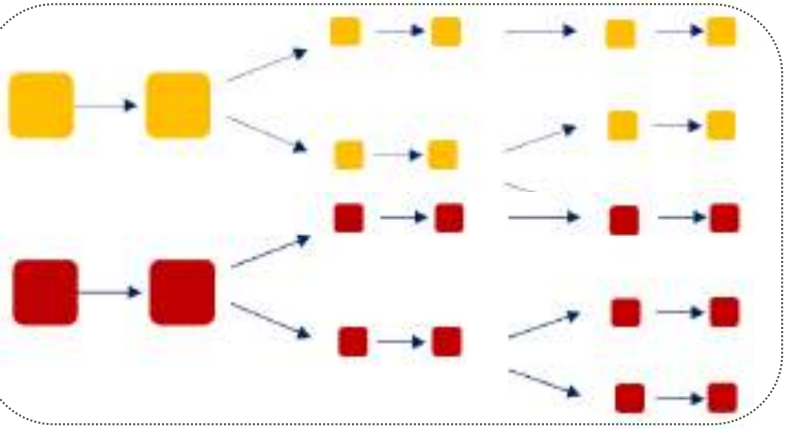
Span-level vs End-to-end tracing

Which service contributes most to the delay experienced by the slowest 1-2% of the requests?

Can't be answered by out-of-the-box Span-level tracing

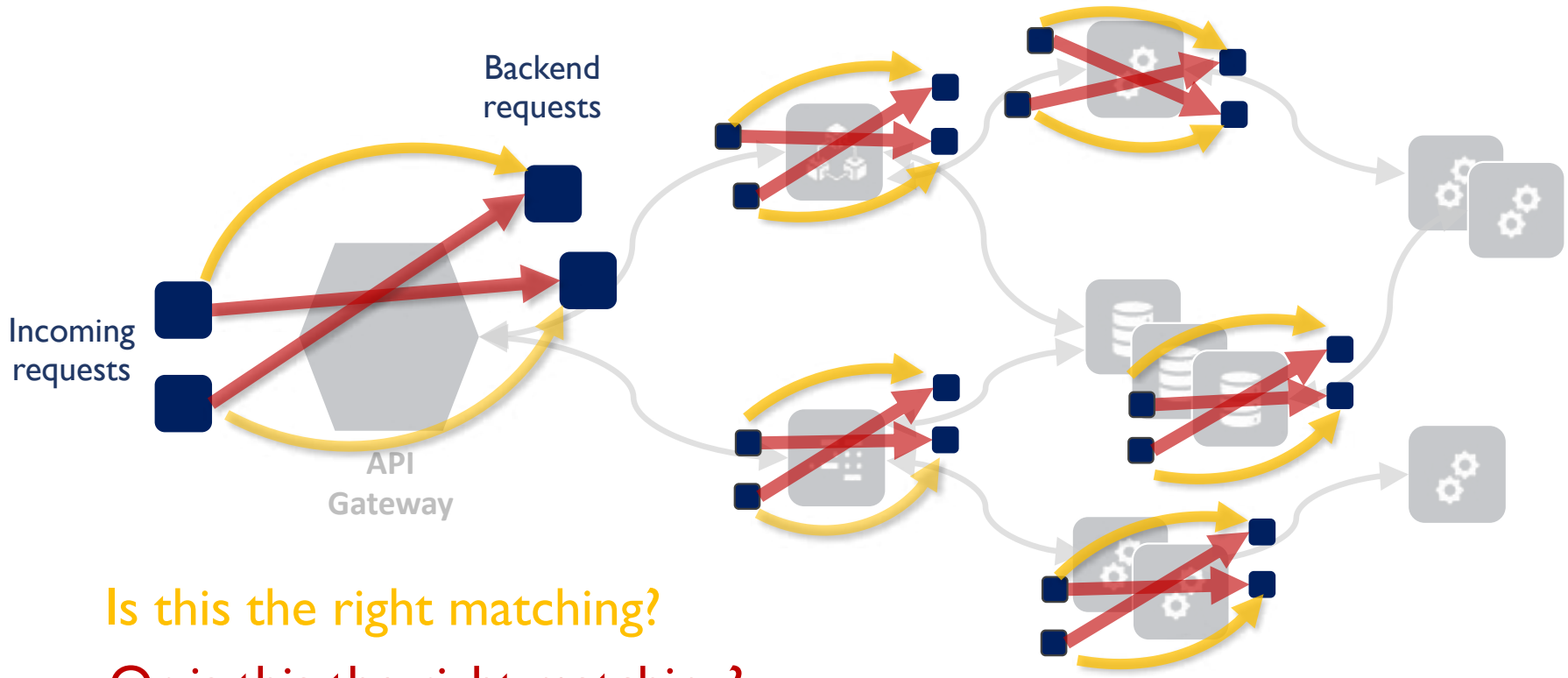
Span-level tracing

End-to-end tracing



Tracing

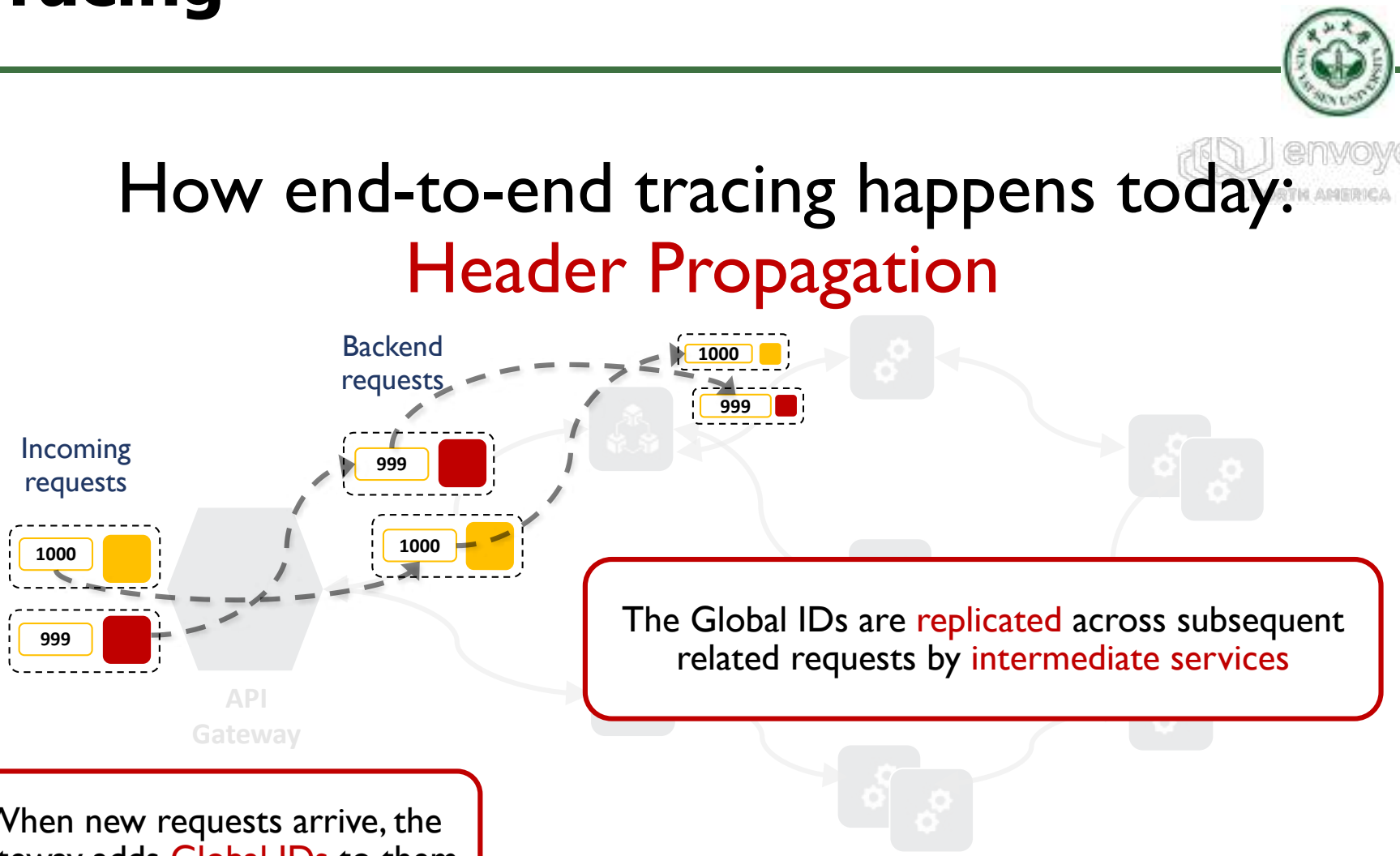
Main challenge of end-to-end tracing



Is this the right matching?

Or is this the right matching?

Tracing



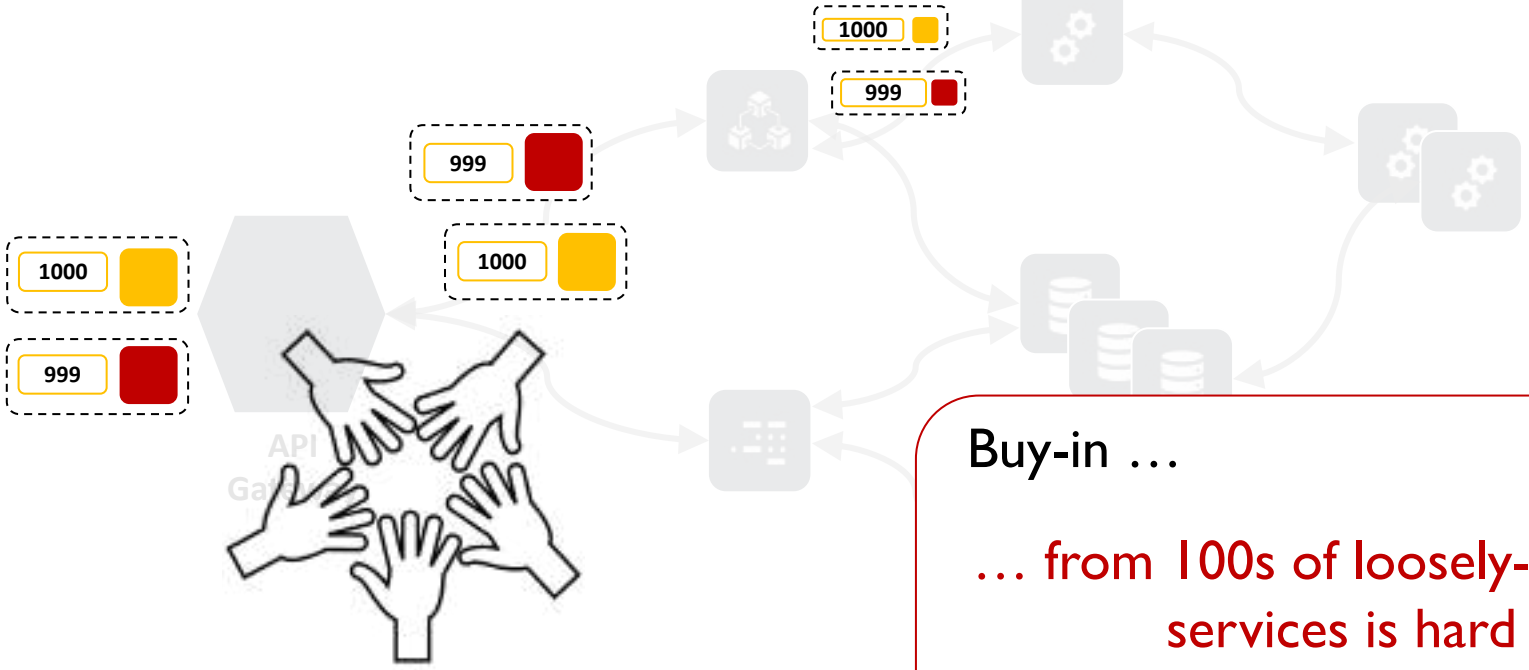
How end-to-end tracing happens today:
Header Propagation

The Global IDs are replicated across subsequent related requests by intermediate services

When new requests arrive, the gateway adds Global IDs to them

Tracing

How end-to-end tracing happens today:
Header Propagation



Buy-in …

… from 100s of loosely-coupled services is hard

☹

谢谢！