

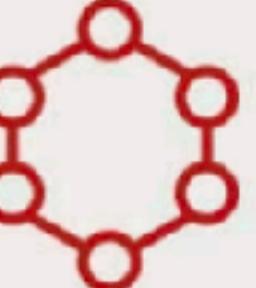
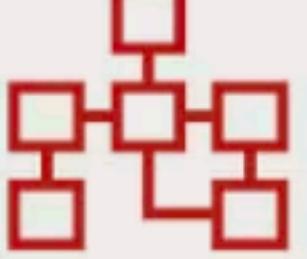
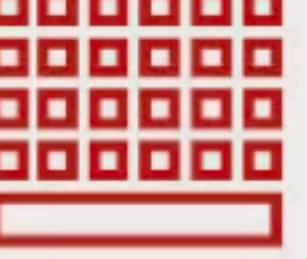
从微服务到云原生和AI原生

# Reference

- <https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture>
- <https://juejin.im/post/6844903968661831693>
- <https://www.aliyun.com/reports/2025-ai-architecture>

# Outline

- 概念
- 特征
- Docker、Kubernetes、Istio
- DevOps
- DevSecOps、BizDevOps
- Serverless
- 云原生
- 算力原生
- AI原生

	Development Process	Application Architecture	Deployment & Packaging	Application Infrastructure
~ 1980	Waterfall 	Monolithic 	Physical Server 	Datacenter 
~ 1990				
~ 2000	Agile 	N-Tie 	Virtual Servers 	Hosted 
~ 2010	DevOps 	Microservices 	Containers 	Cloud 

# 发展历史

单体架构

垂直架构

SOA 架构

微服务架构

云原生架构

AI 原生架构

实现快  
维护成本高

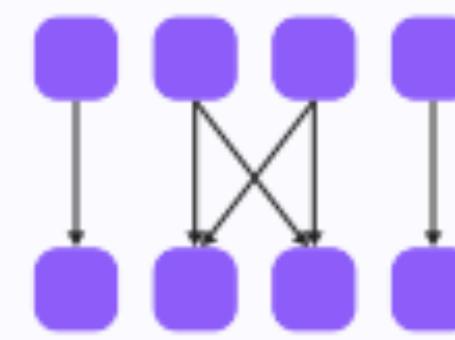
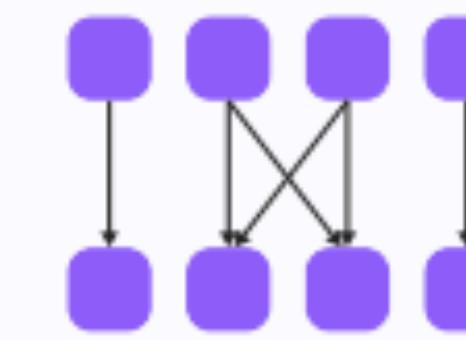
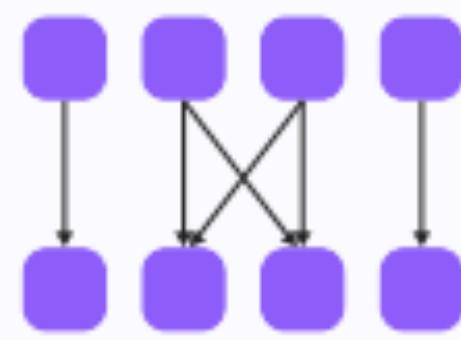
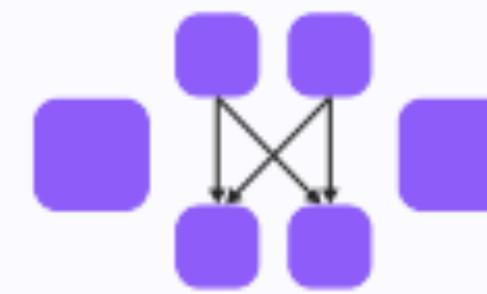
模块化  
负载均衡

服务管理  
RPC 技术

高密度部署  
原子、自治

按量使用  
极致弹性

基于模型  
Agent 驱动



# 架构演化

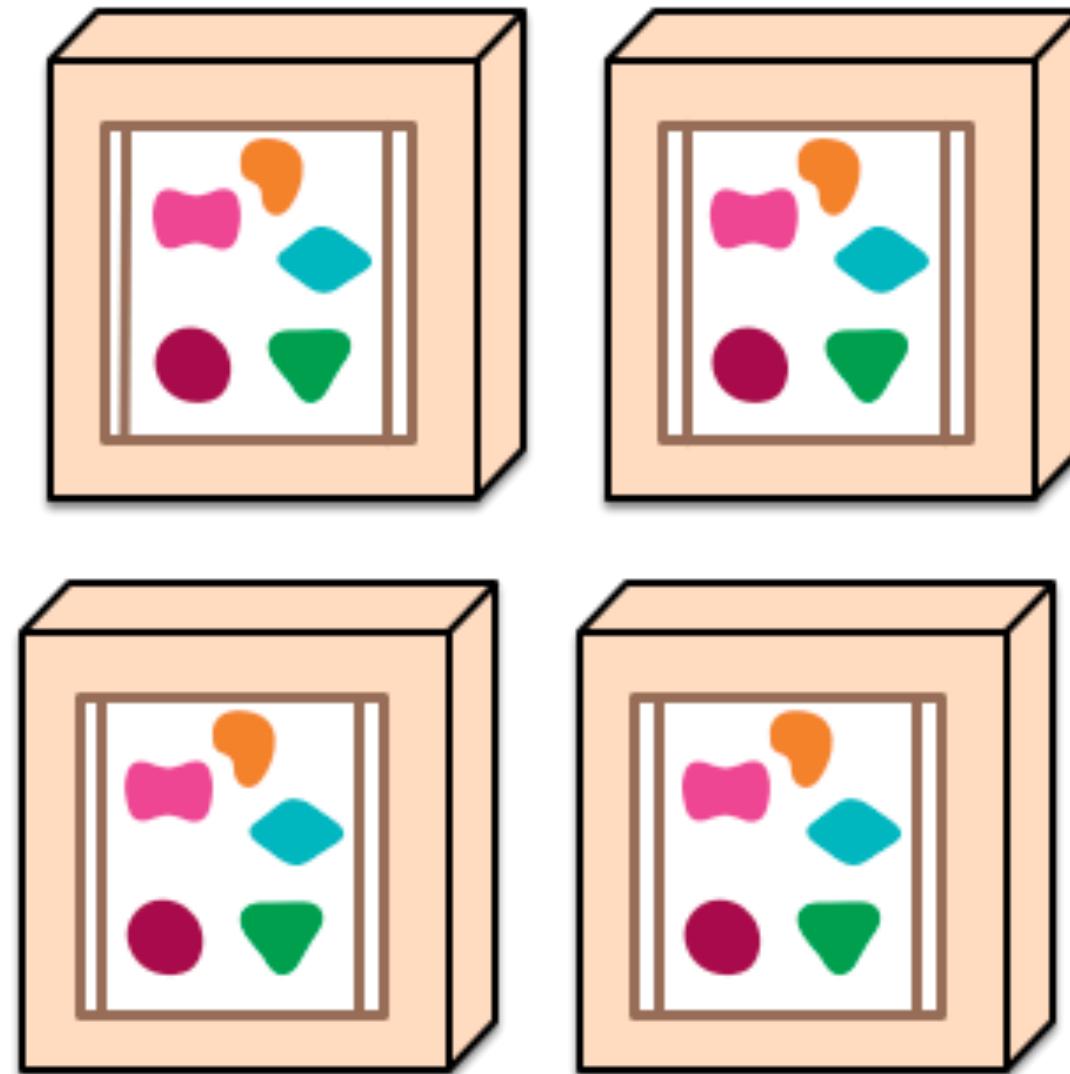
# What are microservices?

- Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are
  - Highly maintainable and testable
  - Loosely coupled
  - Independently deployable
  - Organized around business capabilities
  - Owned by a small team
- The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.

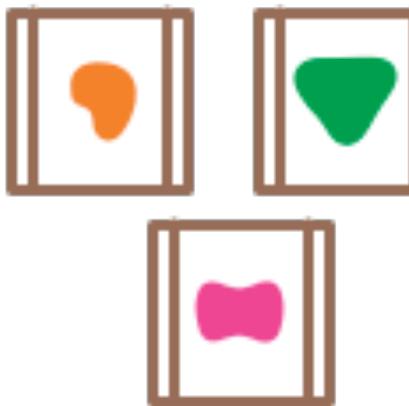
一个单体应用程序把它所有的功能放在一个单一进程中...



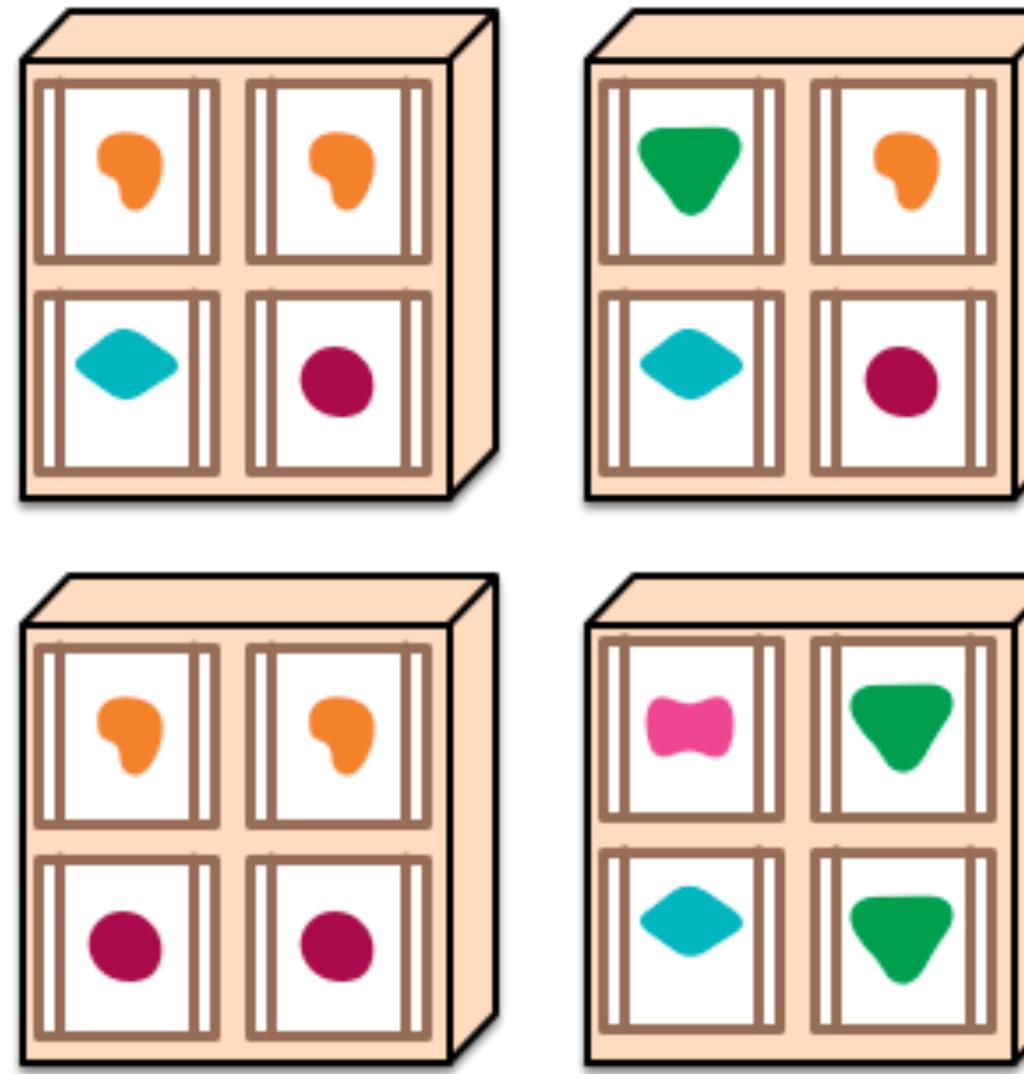
...并且通过在多个服务器上复制这个单体进行扩展



一个微服务架构把每个功能元素放进一个独立的服务中...



...并且通过跨服务器分发这些服务进行扩展，只在需要时才复制.



# 单体和微服务

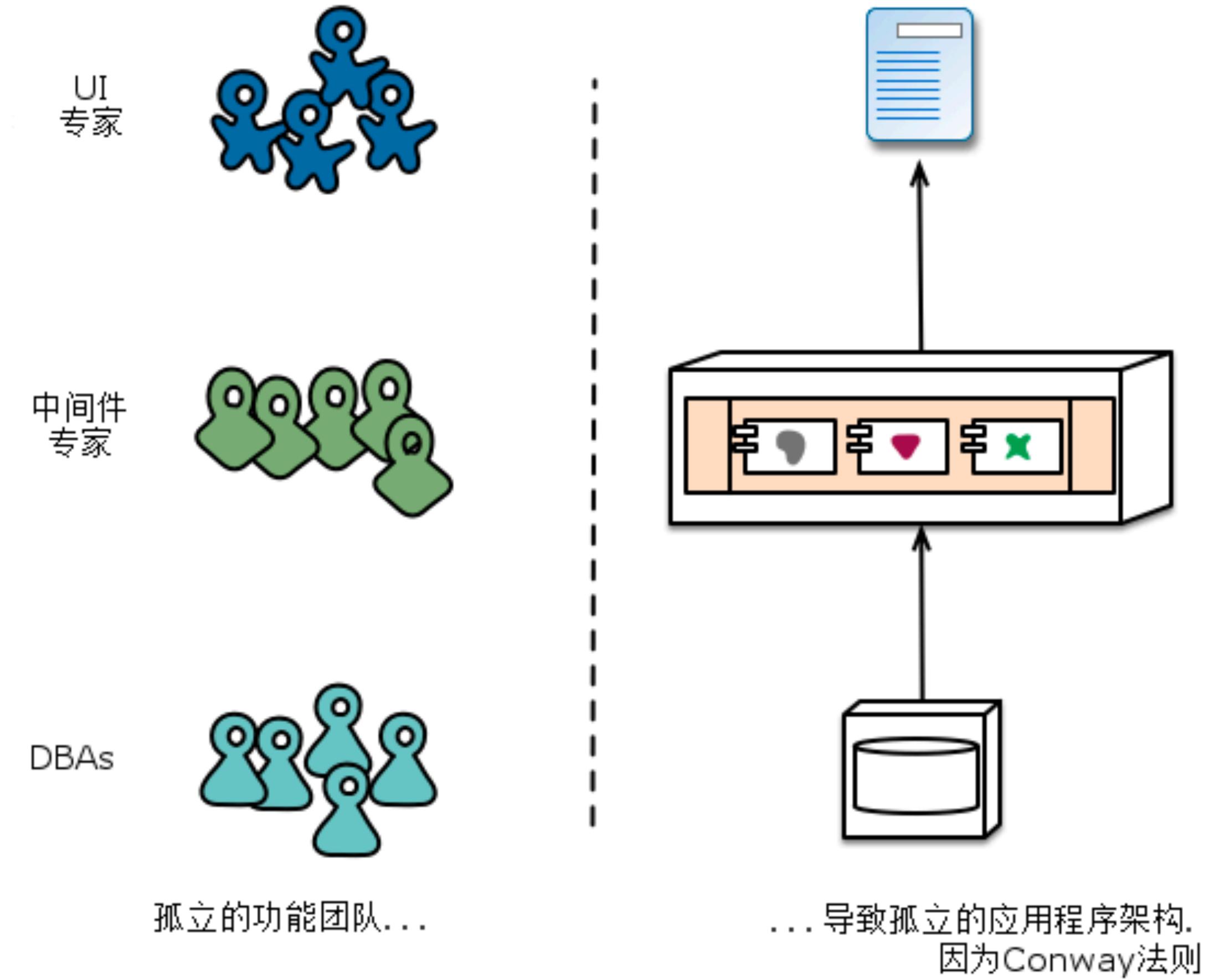
# 微服务架构的特征

# 1. 通过服务组件化

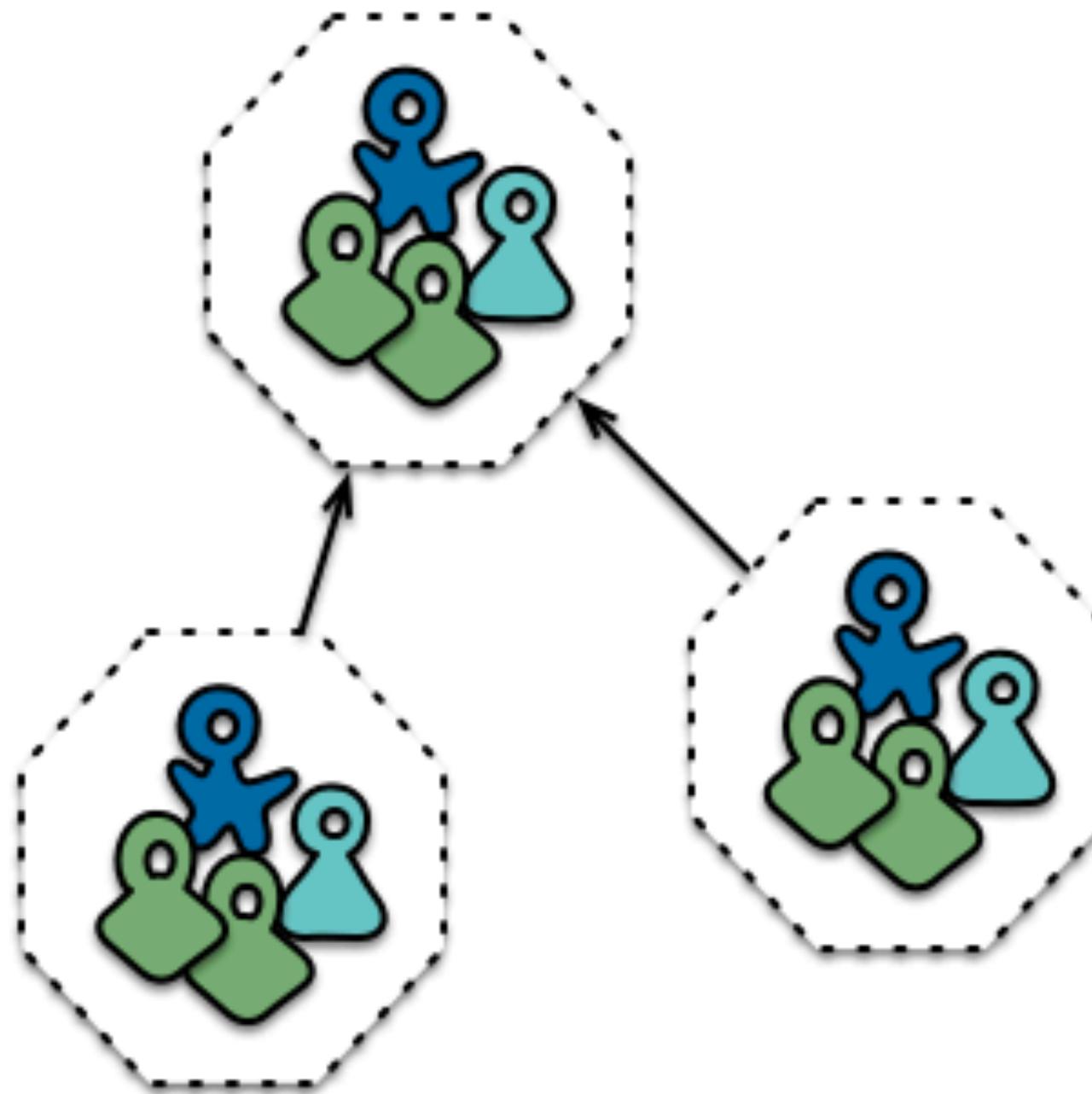
- 组件是什么?
  - 我们的定义是：组件是一个可独立替换和独立升级的软件单元。
- 我们把**库**定义为链接到程序并使用内存函数调用来调用的组件，而**服务**是一种进程外的组件，它通过web服务请求或rpc(远程过程调用)机制通信。
- 服务组件化
  - 使用服务作为组件而不是使用库的一个主要原因是服务是可独立部署的。
  - 使用服务作为组件的另一个结果是一个更加明确的组件接口。

## 2. 围绕业务能力组织

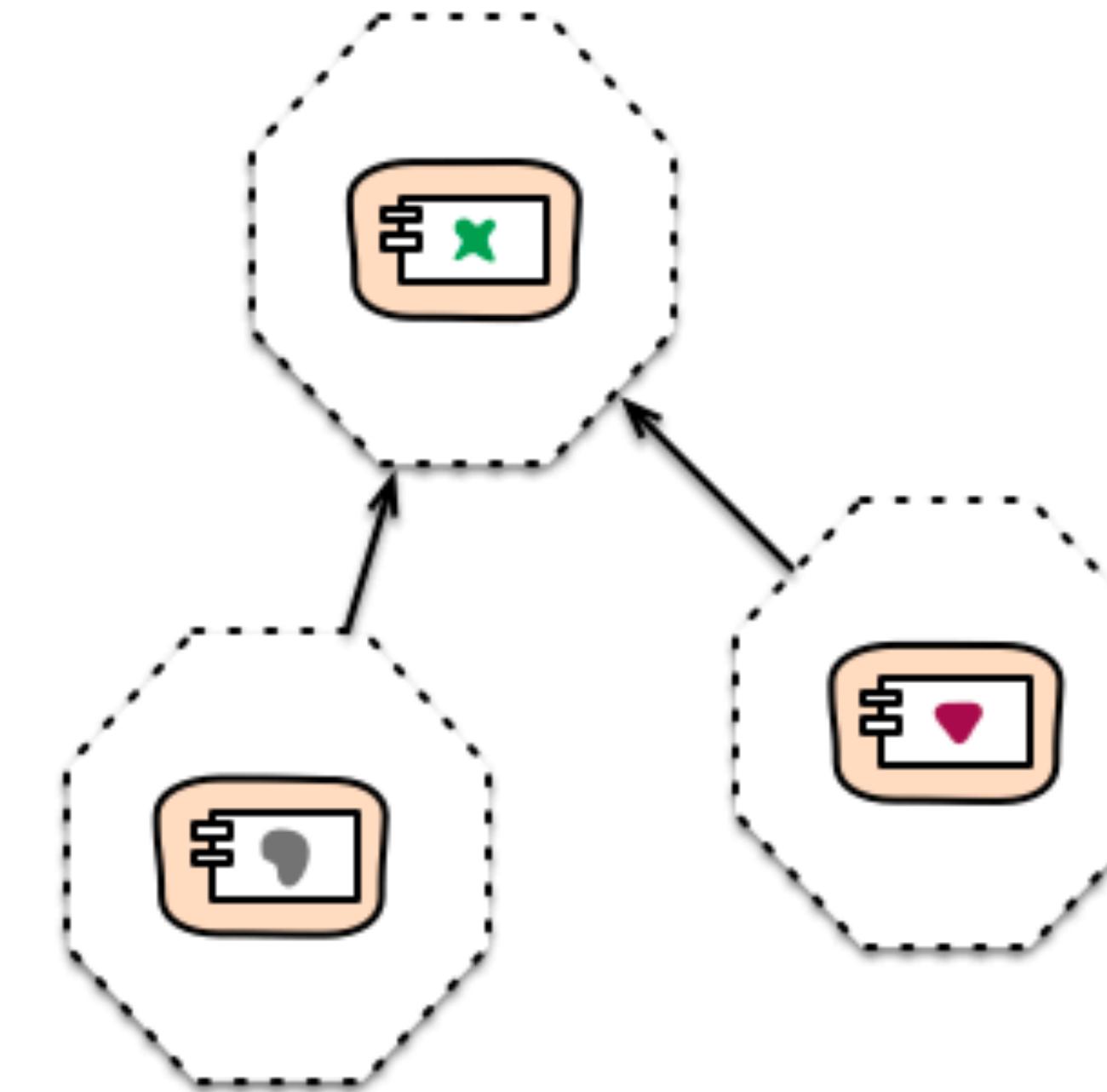
- 当想要把大型应用程序拆分成部件时，通常管理层聚焦在技术层面，导致UI团队、服务侧逻辑团队、数据库团队的划分。
- 当团队按这些技术线路划分时，即使是简单的更改也会导致跨团队的时间和预算审批。
- 一个聪明的团队将围绕这些优化，两害取其轻 - 只把业务逻辑强制放在它们会访问的应用程序中。换句话说，逻辑无处不在。



# 孤立的功能团队



跨功能团队...



...围绕业务能力组织的团队  
根据Conway法则

# 跨功能团队

**PS：微服务到底有多大？**

# 3. 是产品不是项目

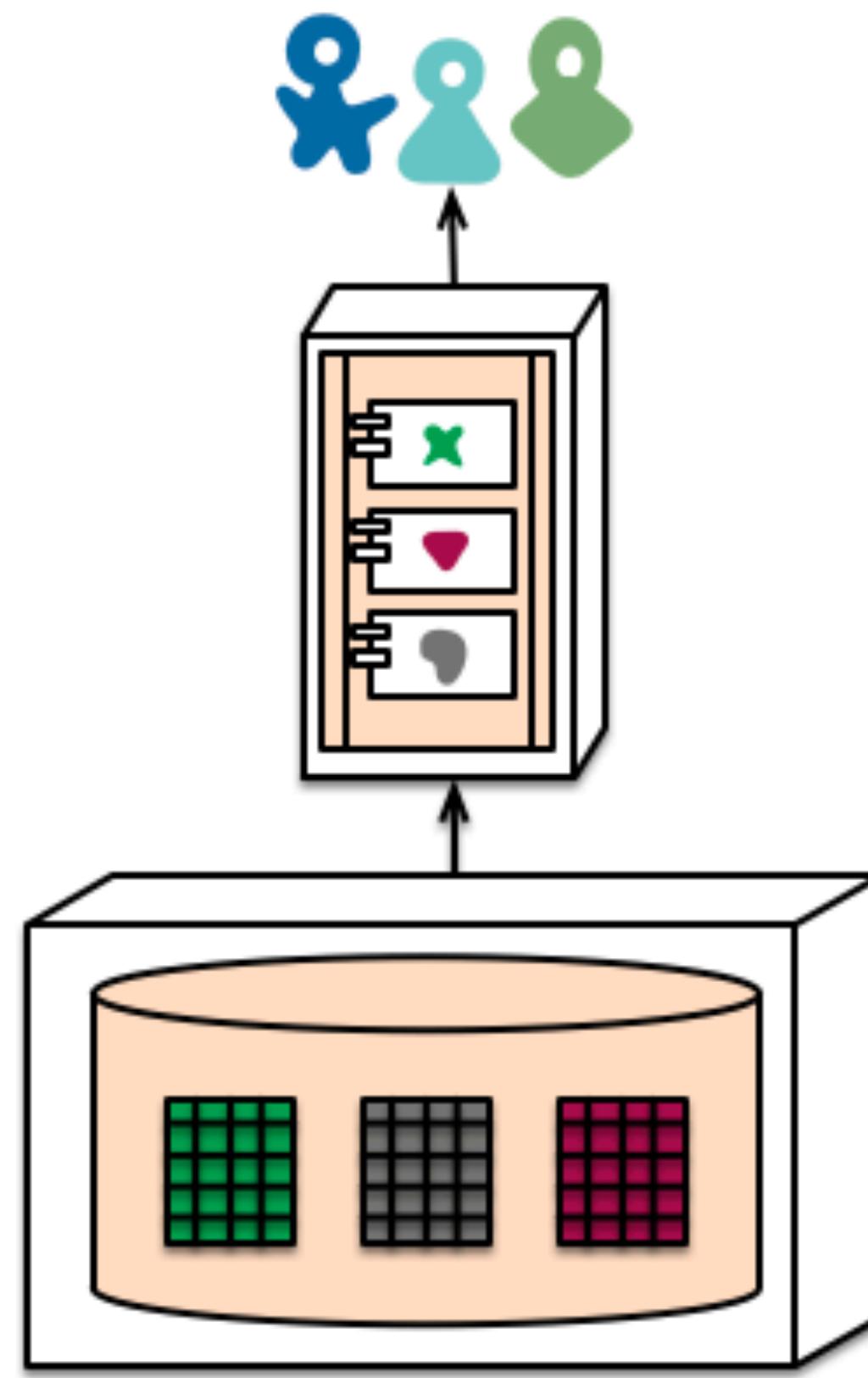
- 我们看到大多数应用程序开发工作使用一个项目模式：
  - 目标是交付将要完成的一些软件。完成后的软件被交接给维护组织，然后它的构建团队就解散了。
  - 微服务支持者倾向于避免这种模式，而是认为一个团队应该负责产品的整个生命周期。
    - 对此一个共同的启示是亚马逊的理念“you build, you run it”，开发团队负责软件的整个产品周期。这使开发者经常接触他们的软件在生产环境如何工作，并增加与他们的用户联系，因为他们必须承担至少部分的支持工作。
  - 产品思想与业务能力紧紧联系在一起。要持续关注软件如何帮助用户提升业务能力，而不是把软件看成是将要完成的一组功能。
  - 没有理由说为什么同样的方法不能用在单体应用程序上，但服务的粒度更小，使得它更容易在服务开发者和用户之间建立个人关系。

# 4 智能端点和哑管道

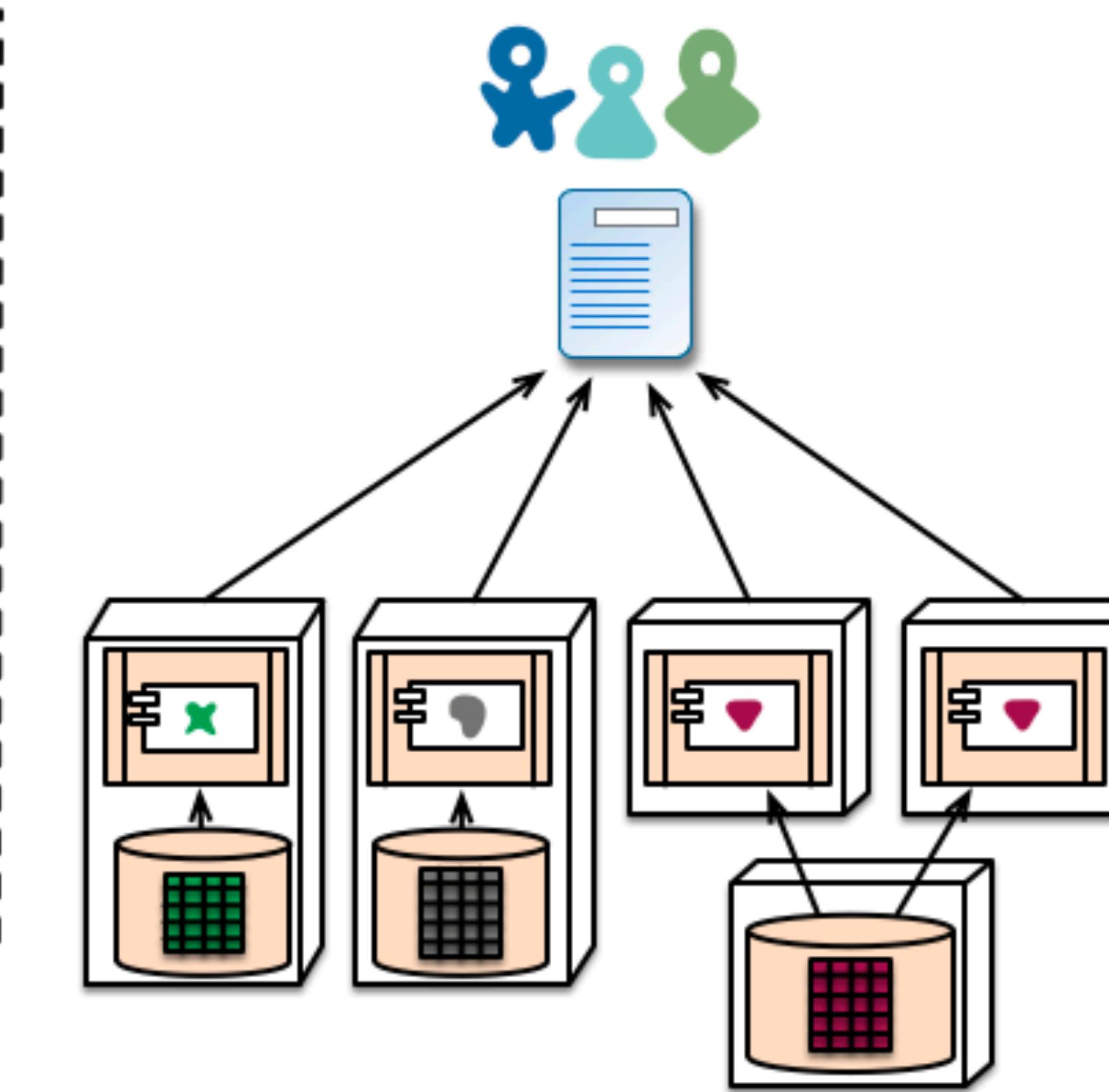
- 最常用的两种协议是
  - 1. 使用资源API的HTTP请求-响应[6]。
    - 对第一种协议最好的表述是
      - 本身就是web，而不是隐藏在web的后面。
        - --Ian Robinson
      - 微服务团队使用的规则和协议，正是构建万维网的规则和协议(在更大程度上，是UNIX的)。从开发者和运营人员的角度讲，通常使用的资源可以很容易的缓存。
    - 2. 在轻量级消息总线上传递消息。
      - 选择的基础设施是典型的哑的(哑在这里只充当消息路由器) - 像RabbitMQ或ZeroMQ这样简单的实现仅仅提供一个可靠的异步交换结构 - 在服务里，智能仍旧存活于端点中，生产和消费消息。
      - 单体应用中，组件都在同一进程中执行，它们之间通过方法调用或函数调用通信。把单体变成微服务最大的问题在于通信模式的改变。一种幼稚的转换是从内存方法调用转变成RPC，这导致频繁通信且性能不好。相反，你需要用粗粒度通信代替细粒度通信。

# 5 去中心化治理

- 集中治理的一个后果是单一技术平台的标准化发展趋势。
- 把单体的组件分裂成服务，在构建这些服务时可以有自己的选择。你想使用Node.js开发一个简单的报告页面？去吧。用C++实现一个特别粗糙的近乎实时的组件？好极了。你想换用一个更适合组件读操作数据的不同风格的数据库？我们有技术来重建它。
- 当然，仅仅因为你可以做些什么，而不意味着你应该这样做 - 但用这种方式划分系统意味着你可以选择。
- 团队在构建微服务时也更喜欢用不同的方法来达标。他们更喜欢生产有用的工具这种想法，而不是写在纸上的标准，这样其他开发者可以用这些工具解决他们所面临的相似的问题。有时，这些工具通常在实施中收获并与更广泛的群体共享，但不完全使用一个内部开源模型。现在git和github已经成为事实上的版本控制系统的标准，在内部开放源代码的实践也正变得越来越常见。
- 或许去中心化地治理技术的极盛时期，就是亚马逊的“谁构建，谁运行”的风气开始普及的时候。各个团队负责其所构建的软件的所有方面的工作，其中包括7 x 24地对软件进行运维。将运维这一级别的职责下放到团队这种做法，目前绝对不是主流。但是我们确实看到越来越多的公司，[将运维的职责交给各个开发团队](#)。Netflix就是已经形成这种风气的另一个组织[11]。避免每天凌晨3点被枕边的寻呼机叫醒，无疑是在程序员编写代码时令其专注质量的强大动力。而这些想法，与那些传统的中心化技术治理的模式具有天壤之别。

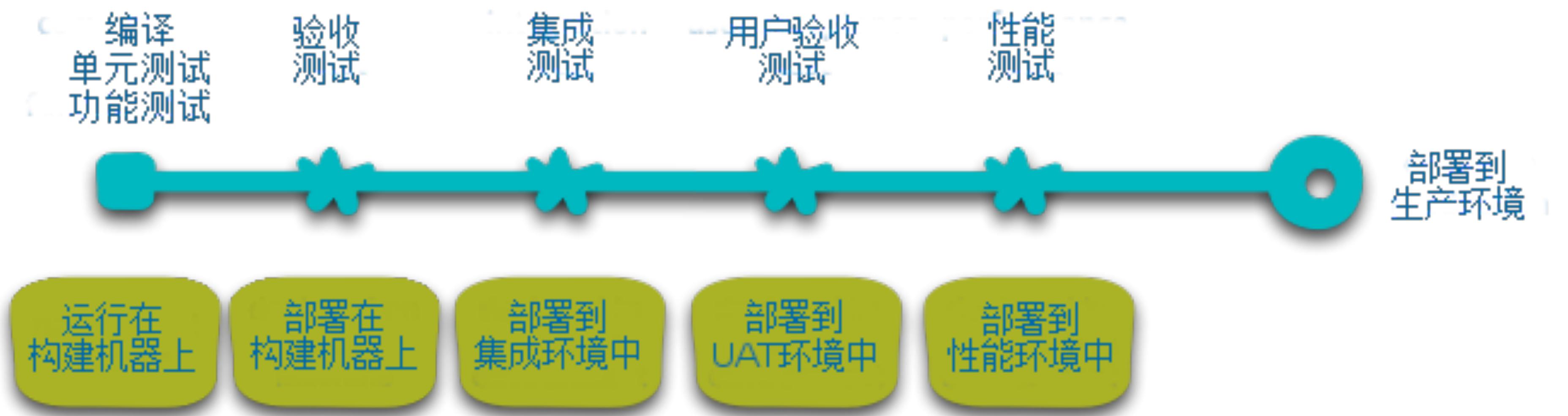


单体 - 单一数据库

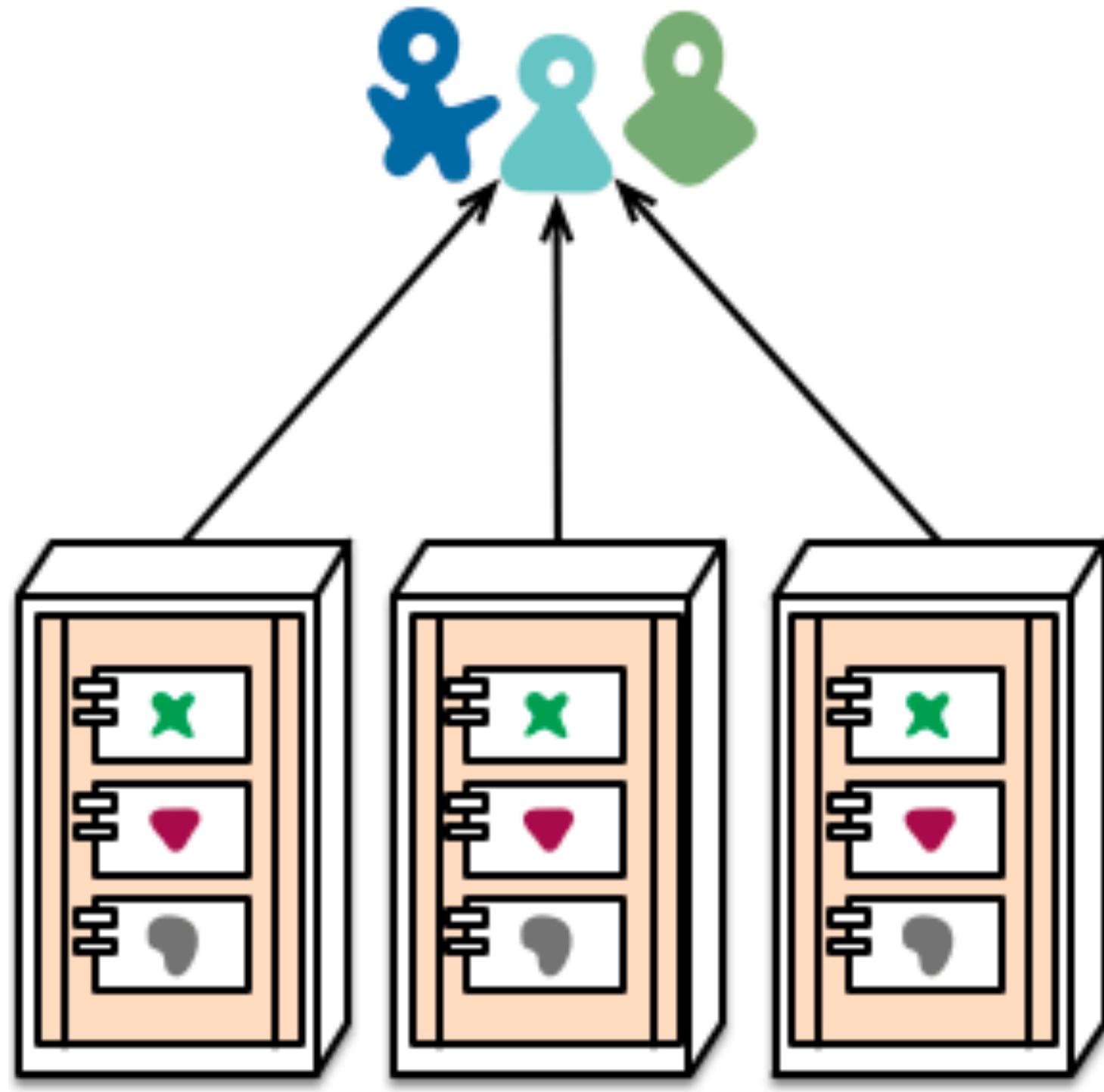


微服务 - 应用程序数据库

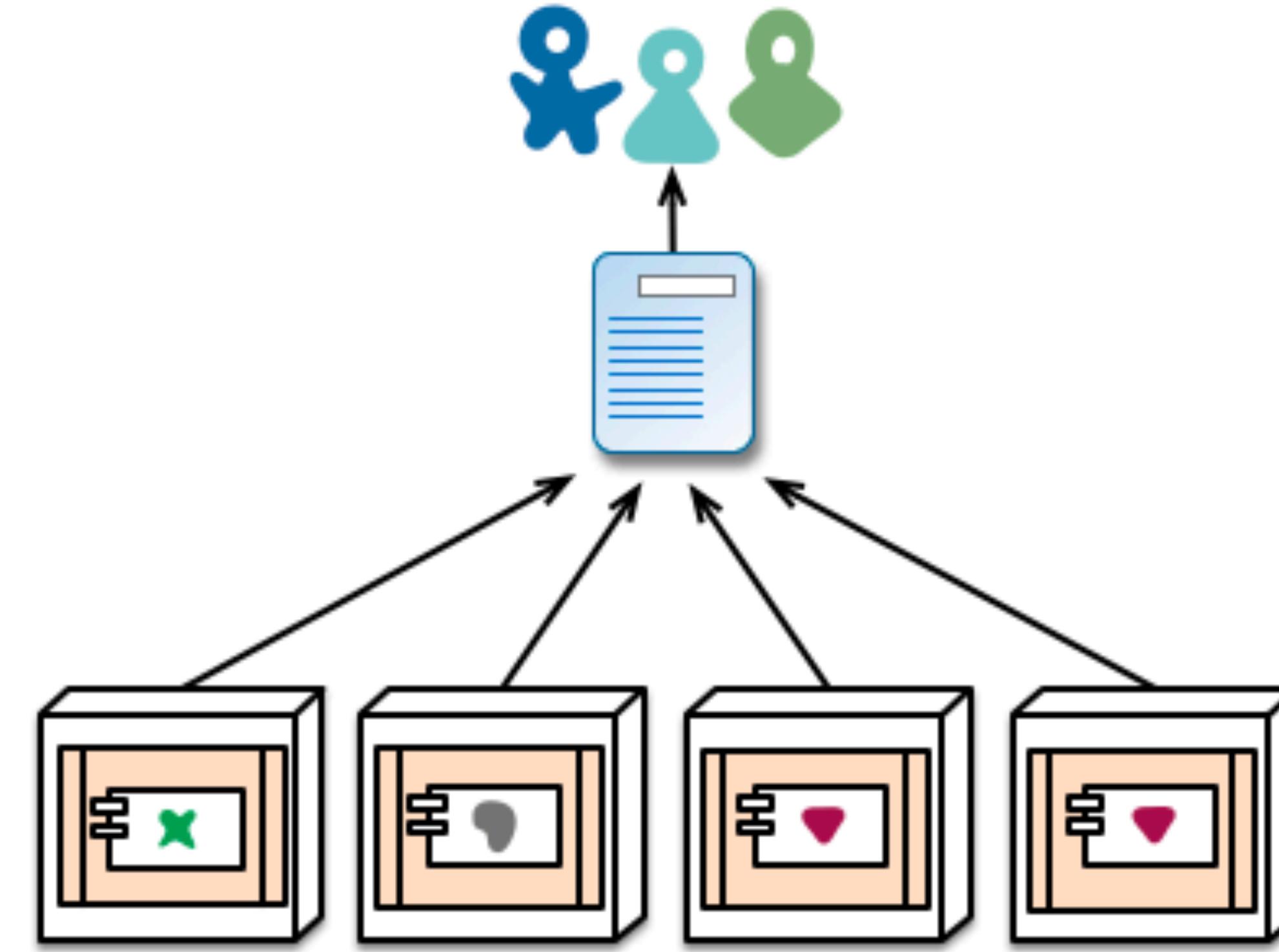
# 6 去中心化数据管理



# 7 基础构建管道



单体 - 多个模块在同一个进程中



微服务 - 每个模块运行在不同的进程中

# 模块部署常常不同

# 8 为失效设计

- 使用服务作为组件的一个结果是，应用程序需要被设计成能够容忍服务失效。任何服务调用都可能因为供应者不可用而失败，客户端必须尽可能优雅的应对这种失败。
- 与单体应用设计相比这是一个劣势，因为它引入额外的复杂性来处理它。结果是，微服务团队不断反思服务失效如何影响用户体验。Netflix的Simian Army在工作日诱导服务甚至是数据中心故障来测试应用程序的弹性和监测。

- 既然服务随时都可能失败，那么能够**快速检测故障**，如果可能的话，能**自动恢复**服务是很重要的。微服务应用程序投入大量比重来进行应用程序的实时监测，既检查构形要素(每秒多少次数据请求)，又检查业务相关指标(例如每分钟收到多少订单)。语义监测可以提供一套早期预警系统，触发开发团队跟进和调查。
- 这对微服务架构特别重要，因为**微服务偏好编排和事件协作**，这会带来突发行为。虽然很多专家称赞偶然涌现的价值，事实的真相是，突发行为有时可能是一件坏事情。监测对于快速发现不良突发行为是至关重要的，所以它可以被修复。
- 单体可以被构建成和微服务一样透明 - 事实上，它们应该是透明的。不同的是，你绝对需要知道在不同进程中运行的服务是否断开。对同一进程中的库来说，这种透明性是不大可能有用的。

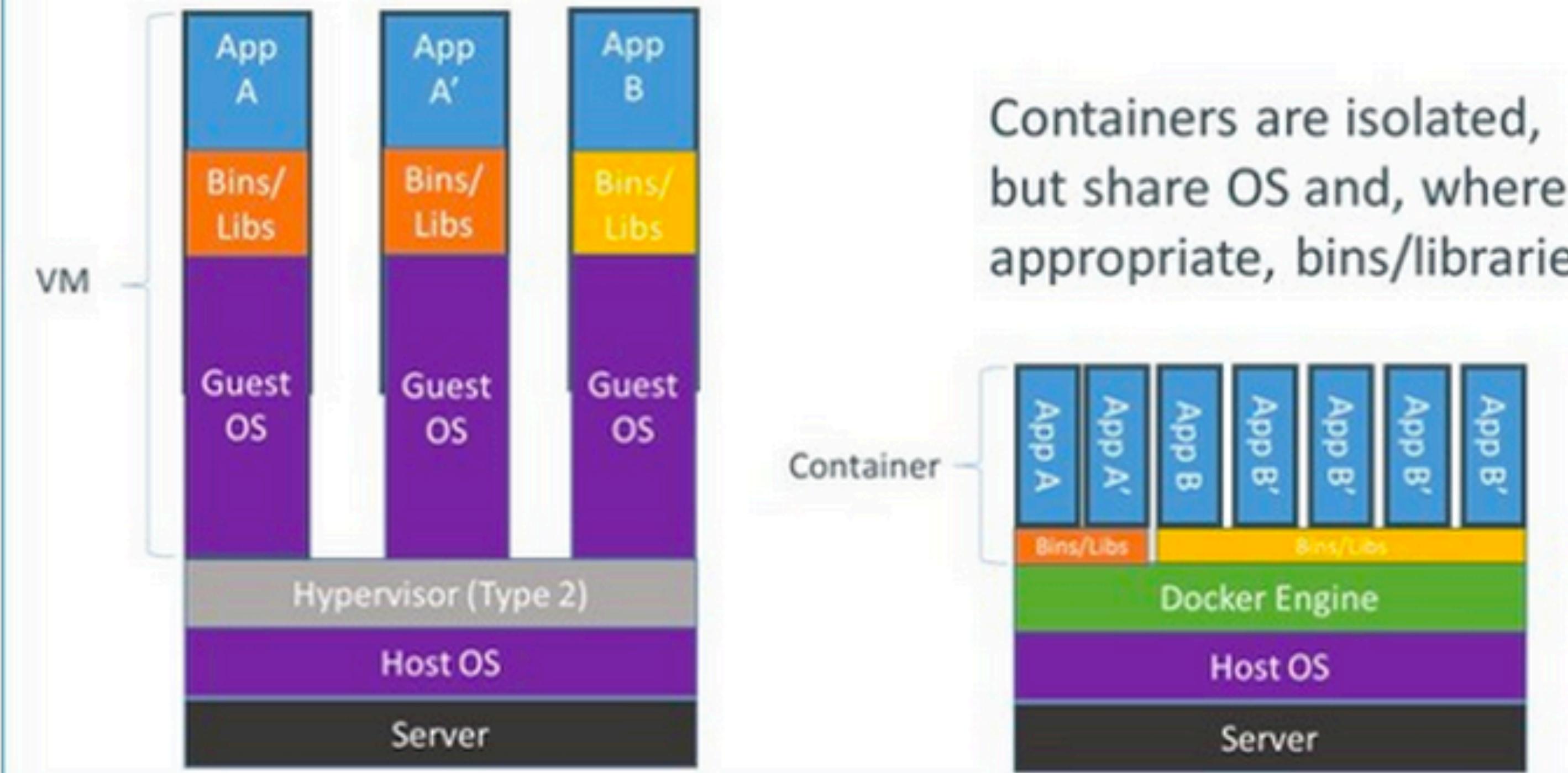
- 微服务团队希望看到为每个单独的服务设置的完善的监控和日志记录，比如控制面板上显示启动/关闭状态和各种各样的运营和业务相关指标。断路器状态、当前吞吐量和时延的详细信息是我们经常遇到的其他例子。

# 进化式设计

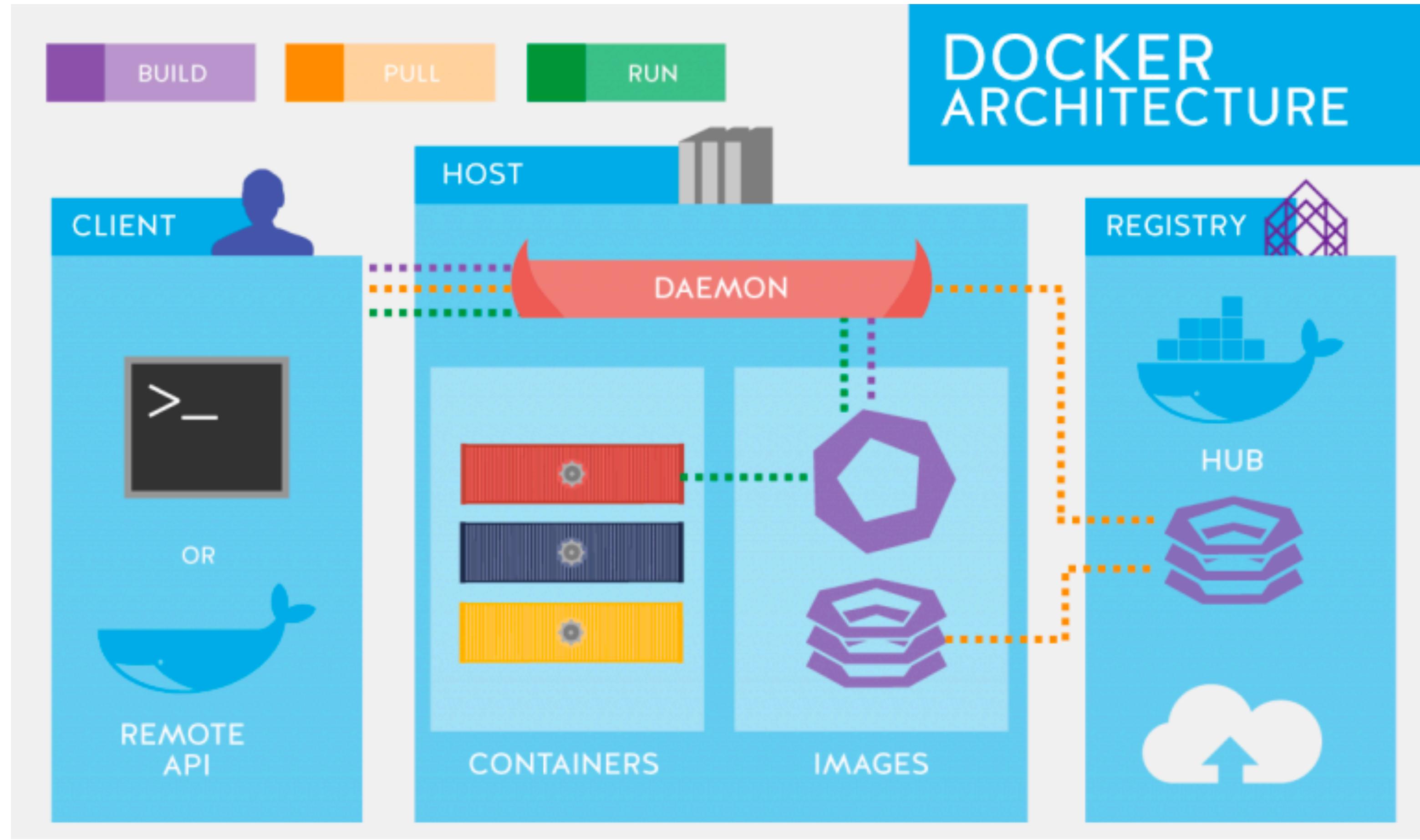
- 微服务从业者，通常有进化式设计背景并且把服务分解看做是进一步的工具，使应用程序开发者能够控制他们应用程序中的变更而不减缓变更。变更控制并不一定意味着变更的减少 - 用正确的态度和工具，你可以频繁、快速且控制良好的改变软件。
- 当你试图把软件系统组件化时，你就面临着如何划分成块的决策 - 我们决定分割我们的应用的原则是什么？组件的关键特性是独立的更换和升级的理念[13] - 这意味着我们要找到这样的点，我们可以想象重写组件而不影响其合作者。事实上很多微服务群组通过明确地预期许多服务将被废弃而不是长期演进来进一步找到这些点。
- 强调**可替代性**是模块设计更一般原则的一个特例，它是通过变更模式来驱动模块化的[14]。你想保持在同一模块中相同时间改变的事情。系统中很少变更的部分应该和正在经历大量扰动的部分放在不同的服务里。如果你发现自己不断地一起改变两个服务，这是它们应该被合并的一个标志。
- 把组件放在服务中，为**更细粒度**的发布计划增加了一个机会。对单体来说，任何变更都需要完整构建和部署整个应用程序。而对微服务来说，你只需要重新部署你修改的服务。这可以简化和加速发布过程。坏处是，你必须担心一个服务的变化会阻断其消费者。传统的集成方法试图使用版本管理解决这个问题，但是微服务世界的偏好是只把版本管理作为最后的手段。我们可以避免大量的版本管理，通过把服务设计成对他们的提供者的变化尽可能的宽容。

# Docker

## Containers vs. VMs



# Containers vs VMs



# Docker Architecture

虚机应用容器化改造时，一般需要执行如下6个流程：



# 容器化改造的流程

# 镜像库

- Registry      Key Features   Pricing   Deployment Options
- Amazon ECR      Private registry, image scanning, vulnerability detection   Paid On-premises, Cloud, Hybrid
- Azure Container Registry      Private registry, image scanning, vulnerability detection   Paid On-premises, Cloud, Hybrid
- Docker Hub   Public registry, image scanning, vulnerability detection   Free and Paid   Cloud
- GitHub Package Registry      Private registry, image scanning, vulnerability detection   Paid Cloud
- GitLab Container Registry      Private registry, image scanning, vulnerability detection   Paid On-premises, Cloud, Hybrid
- Google Artifact Registry      Private registry, image scanning, vulnerability detection   Paid Cloud
- Harbor Container Registry      Private registry, image scanning, vulnerability detection   Free and Paid   On-premises, Cloud, Hybrid
- Red Hat Quay      Private registry, image scanning, vulnerability detection   Paid On-premises, Cloud, Hybrid
- Sonatype Nexus Repository OSS   Private registry, image scanning, vulnerability detection   Free and Paid   On-premises, Cloud, Hybrid

容器镜像服务 CIS (Container Image Service) 是一种简单易用、安全可靠的云原生资产管理服务，提供容器镜像、Helm Chart等符合OCI标准的云原生制品的全生命周期管理，方便用户对云原生资产的拉取、推送、删除，并且提供移动云镜像、DockerHub镜像和用户公开镜像，方便用户使用。

## 个人版

- **我的镜像**: 提供镜像的全生命周期管理，支持推送、拉取、删除、公开镜像，查看镜像层信息，自定义镜像描述和摘要等操作。
- **命名空间**: 方便用户对镜像进行隔离、管理，例如不同的开发小组可使用不同的镜像空间。
- **镜像资源**: 提供镜像资源搜索功能，有超过170个基础镜像，包括移动云基础镜像、DockerHub镜像、用户公开镜像三大类。
- **镜像制作**: 支持从源代码到容器镜像的持续集成能力，具备安全、稳定、高效等特性。

## 企业版

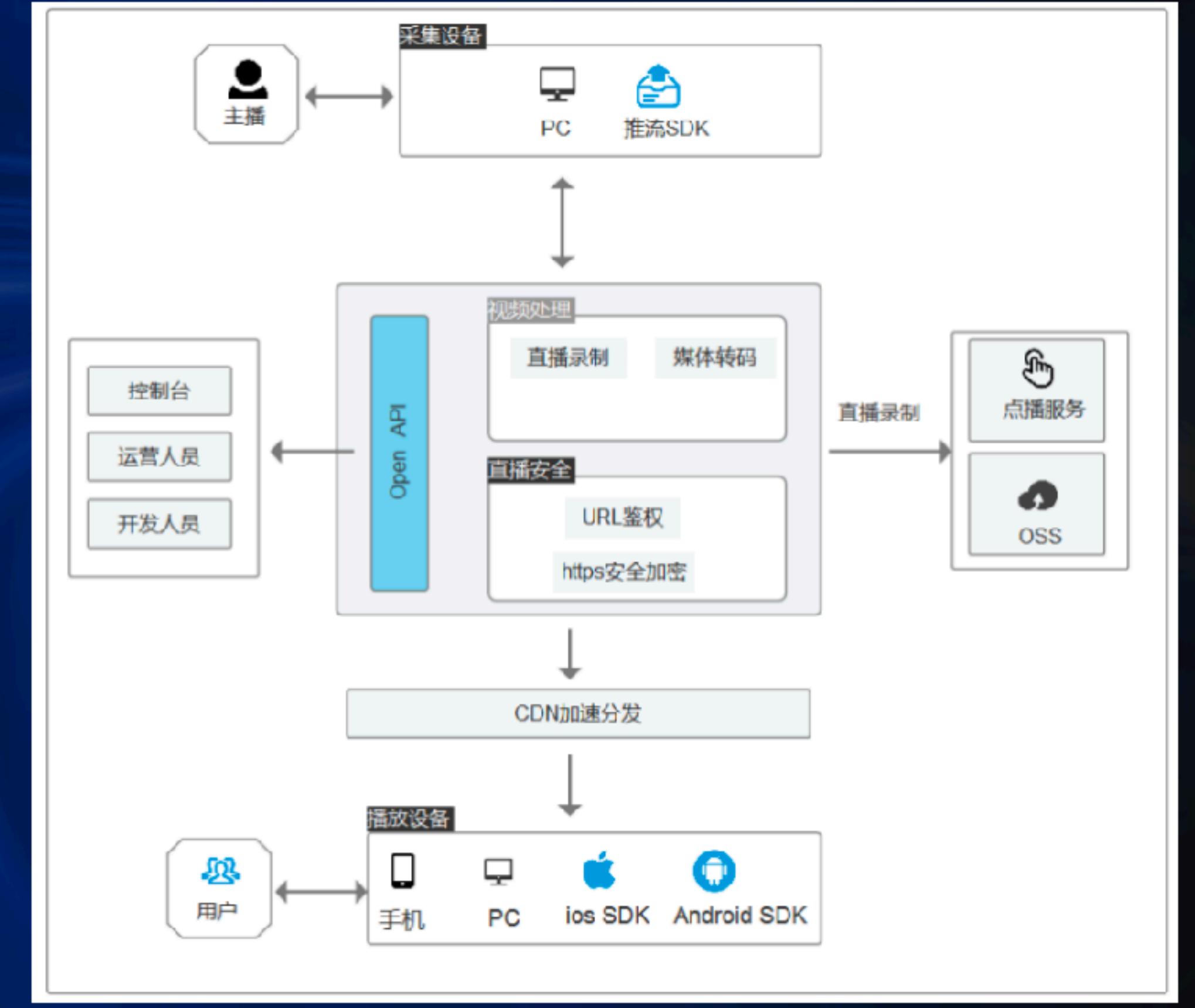
- **独享实例**: 每个实例均具有独立的服务后端及后端存储，相较于个人版共享镜像托管，实例完全由单个用户独享，无需担心其他用户的影响及使用限制。
- **实例同步**: 通过设定规则，实现容器镜像从源实例自动同步至目标实例。若目标实例与源实例位于不同地域，可以实现跨地域的同步。
- **Helm Chart**: 支持Helm Chart制品的全生命周期管理，拓展了OCI制品全生命周期管理的类型。
- **P2P分发**: 基于P2P方式支持千节点下镜像的大规模分发，分发效率提升数倍。
- **云原生交付链**: 用户可自由组合镜像构建、制品安扫、制品同步和应用部署等任务，提供全链路可观测、可追踪、安全防护能力。

# 移动云容器镜像服务CIS

## □ 视频直播解决方案

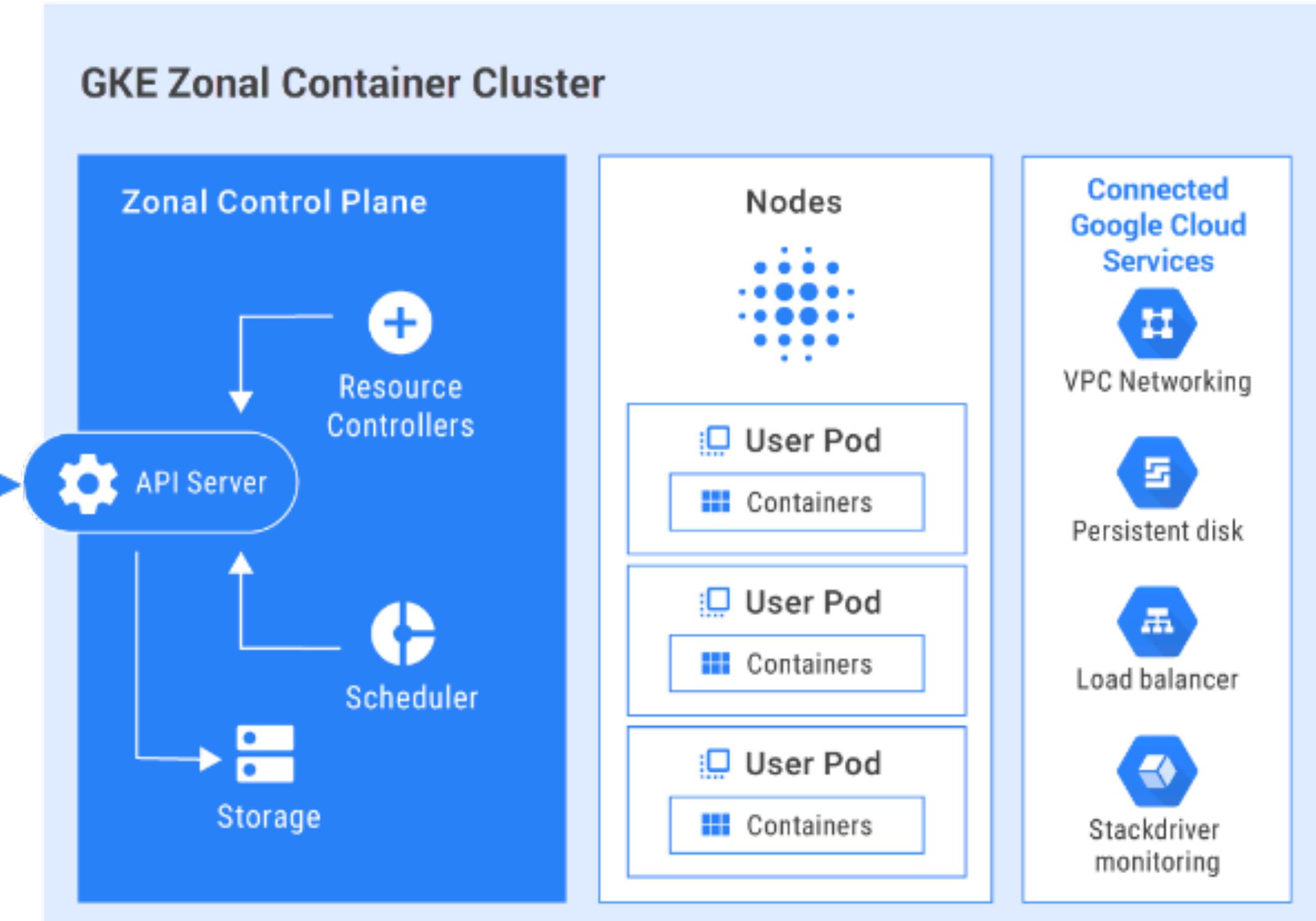
某科技公司在其开发的旅游直播平台中，借助移动云容器服务、容器镜像服务等产品，采用容器方式部署，以实现弹性伸缩，满足客户业务突发情况。采用负载均衡方式增加系统并发量，同时借助移动云视频直播产品全国加速分发的能力较好满足了客户需求。

容器镜像服务在整个系统中担任了非常重要的角色，通过使用容器镜像服务，保障了容器业务的快速扩展和极速部署。充分发挥了容器镜像服务安全可靠、灵活易用的特性。



# CIS实践案例

# Kubernetes



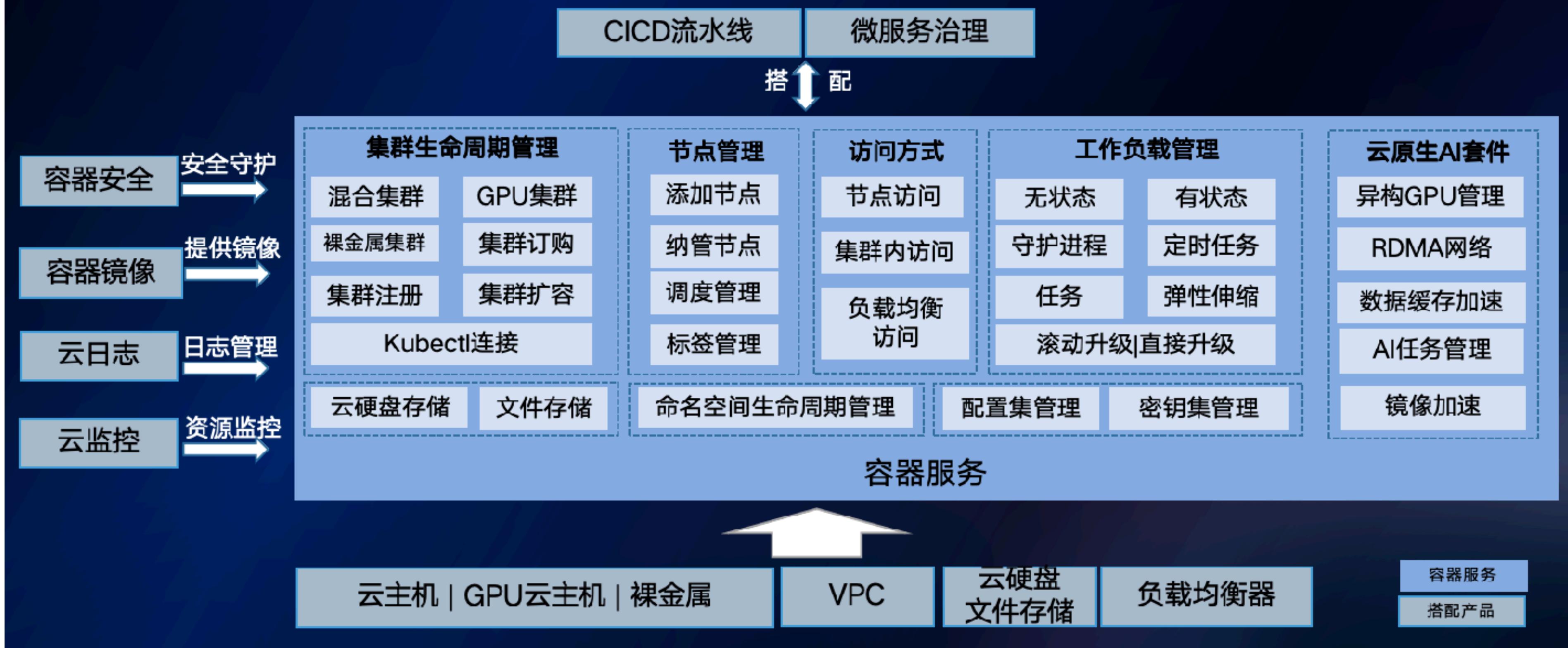
Color  
Legend:

GKE provisions, maintains, and  
operates.

GKE provisions. User optionally  
maintains and operates.

# Kubernetes Architecture

容器服务基于Kubernetes技术，依托移动云基础资源，构建高性能、可扩展的Kubernetes集群，为用户容器应用提供一站式生命周期管理，助力用户应用轻松上云。通过云原生AI套件，为客户提供强大的AI任务管理及加速能力。



# 移动云-KCS平台

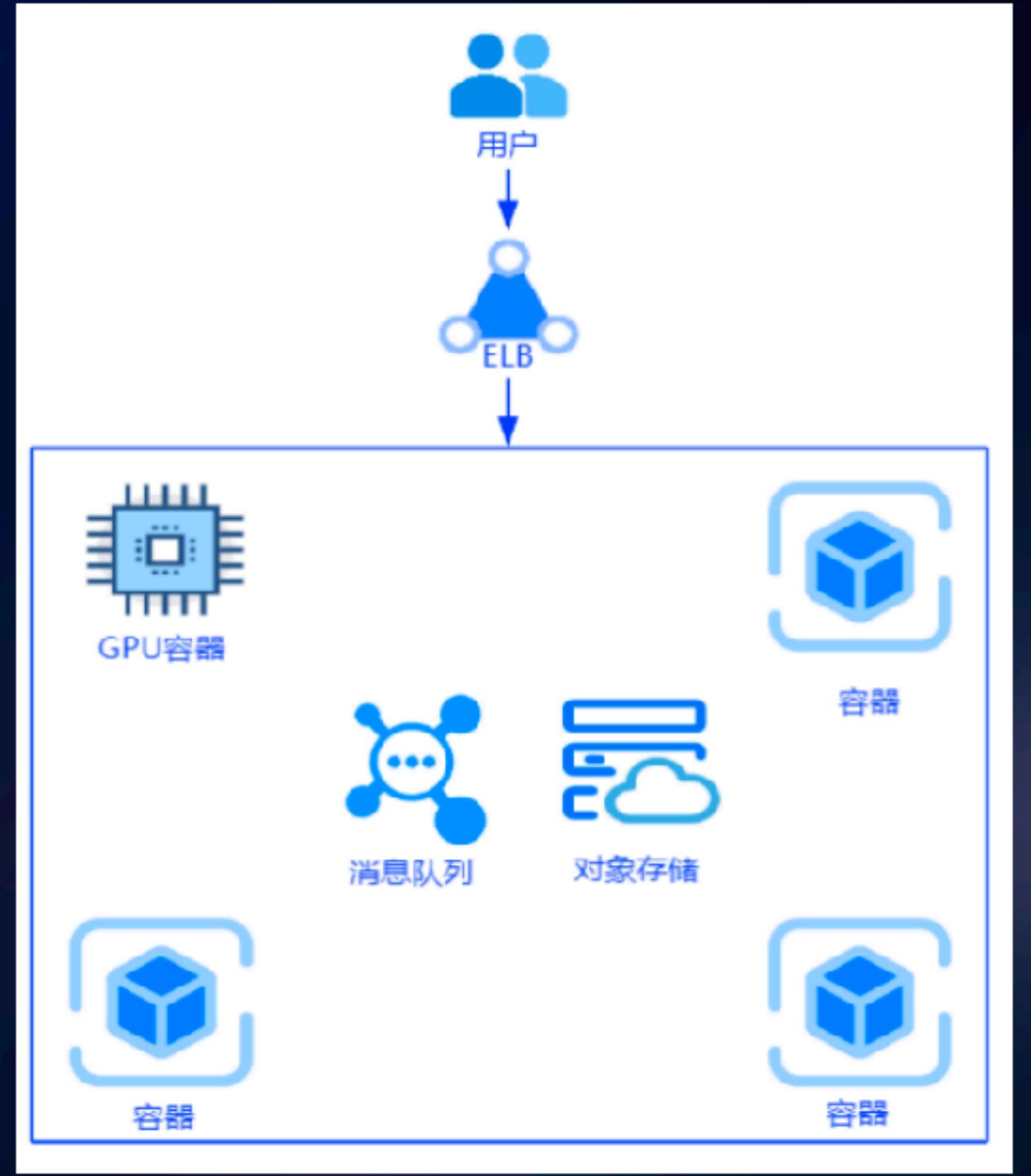
## 口 某科技公司推理业务

该公司运营着多个业务系统，涵盖语音识别、图像分析、智能推荐等领域，这些系统不仅需要处理大量的实时数据，还要求**快速响应和高精度的推理能力**。

### KCS对场景的支持解析：

- ① 所有业务系统均以容器化部署在KCS中，确保系统在面对流量峰值时能迅速调整资源，维持稳定的服务质量。
- ② 针对图像分析等高计算需求的推理业务，部署GPU容器加速了深度学习模型，推理任务的处理速度获得了显著提升。
- ③ 支持不同业务模块之间的异步消息传递，如Kafka或RocketMQ。

推荐搭配：容器镜像+ 容器服务+云负载均衡+消息队列+对象存储



# KCS实践案例—推理业务

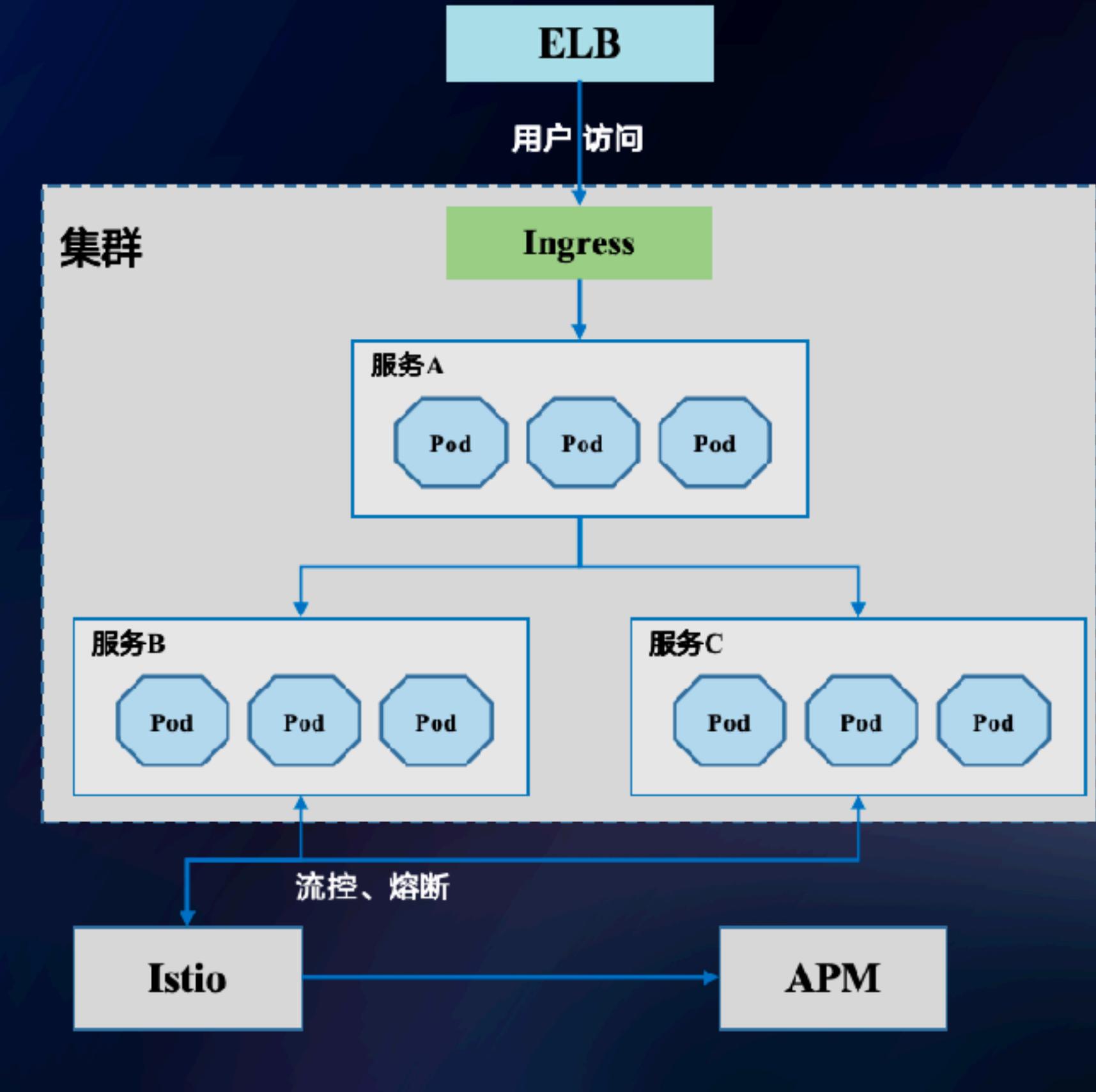
## 口 微服务应用管理

微服务架构应用能方便的实现产品敏捷开发、快速迭代的需求。

KCS对场景的支持解析：

- ① 将合理拆分的微服务模块镜像存储在移动云容器镜像仓库，进行全生命周期管理；
- ② 移动云容器服务产品提供各个微服务的部署和全生命周期管理；
- ③ 搭配微服务治理产品，支持微服务治理，可提供应用进行流控、灰度、熔断等治理能力
- ④ 支持应用的灰度发布；
- ⑤ 支持Helm的应用部署

推荐搭配：容器镜像+ 容器安全+容器服务+云负载均衡



# KCS实践案例—微服务应用

## 口 电商/直播业务

针对电商业务、视频直播业务等具有**明显业务峰谷值波动**的行业，传统应用部署和运行方式不能灵活的进行业务扩缩容的动态调整适应流量变化。容器服务则可以根据业务流量进行自动扩容/缩容，无需人工干预，能够有效避免流量激增扩容不及时导致的系统崩溃，以及平时大量资源闲置造成的浪费。

KCS对场景的支持解析：

- ① 容器业务Pod实例的手动扩缩容
- ② 根据资源利用率（CPU/内存）、时间定义自动弹性伸缩策略，秒级触发容器的伸缩操作。
- ③ 集群的自动化弹性扩容

推荐搭配：云主机+云硬盘+负载均衡器+容器镜像+容器服务

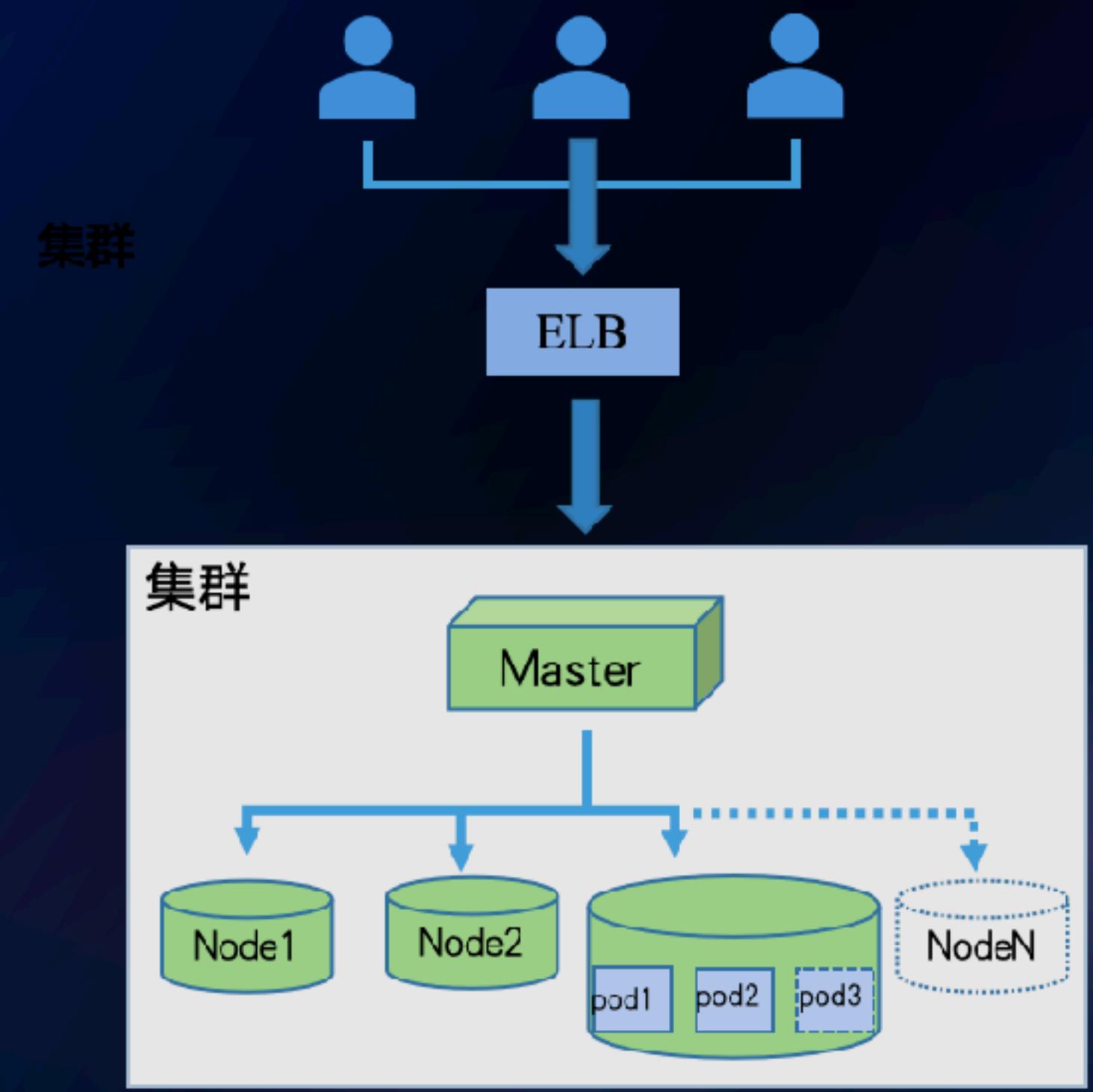


图2-弹性扩容

# KCS实践案例—电商/直播业务

### 某医疗行业客户



某医疗企业在国内主要业务为采集人类遗传资源，组建基因库，该企业处于快速发展阶段，**产品迭代频率需要小步快跑，敏捷发版优化体验，业务量快速大幅增长，资源扩容需求较为紧迫。**针对客户需求，为客户打造了以移动云容器服务、数据库为业务承载核心产品，搭配容器镜像、消息队列以及研发效能工具构建DevOps敏捷交付链，并配合安全、监控等产品的一整套解决方案。

容器服务提供专有集群用于业务部署，3台 master 节点保障管理面高可用；业务节点跨多可用区，集群核心组件监控和健康巡检 7×24 小时监测集群健康状态；集群、节点、应用、容器多层次级资源监控随时掌握业务负载情况；弹性伸缩实现按需调度资源，全面提升资源使用率；

# KCS实践案例 – 医疗行业客户

## 某旅游行业客户



某旅游局自2020年12月订购使用容器服务集群，已持续平稳运行18个月。

承载了景区运营分析、预约系统以及景区直播等多个核心业务。其中景区运营分析、预约系统均包括大量重要数据与客户信息，对于平台数据存储和容灾能力都提出了较高要求，直播业务更是对产品时延、负载能力和稳定性发起挑战。

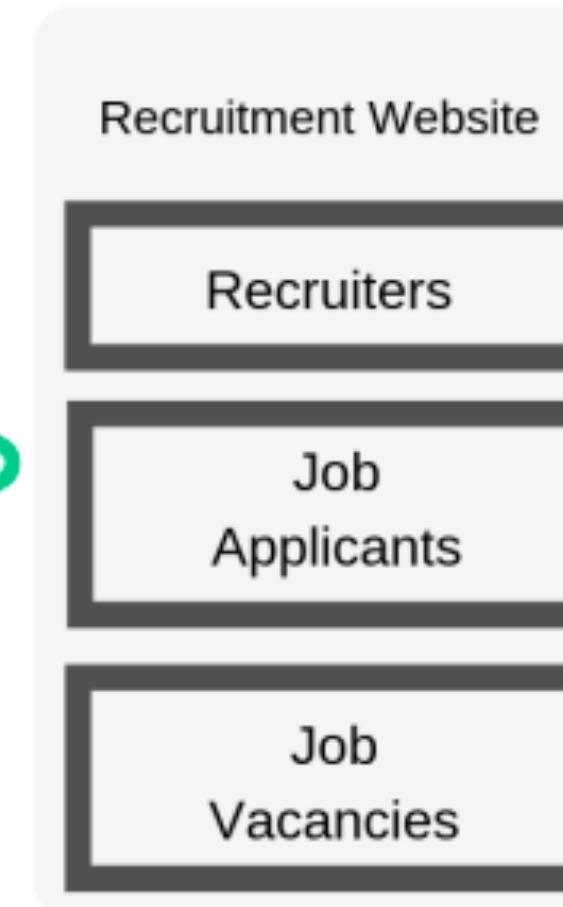
容器服务搭配容器镜像、云主机、文件存储等基础产品构建基础业务承载平台，使用容器安全等产品提供安全保障，形成一套完全匹配客户业务需求的标准解决方案。容器化运行业务，实现了应用发布、回滚的自动化，大幅减少运维工作；业务Pod的弹性伸缩将业务扩展的时间从小时级缩短为秒级。

# KCS实践案例—旅游行业客户

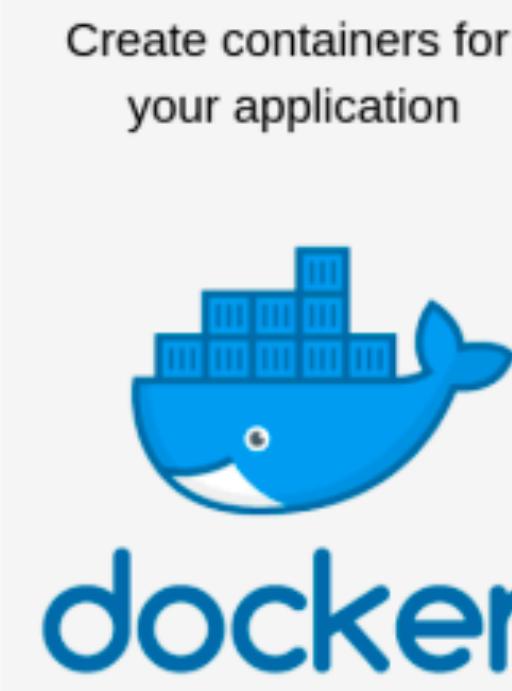
## Monolithic Application



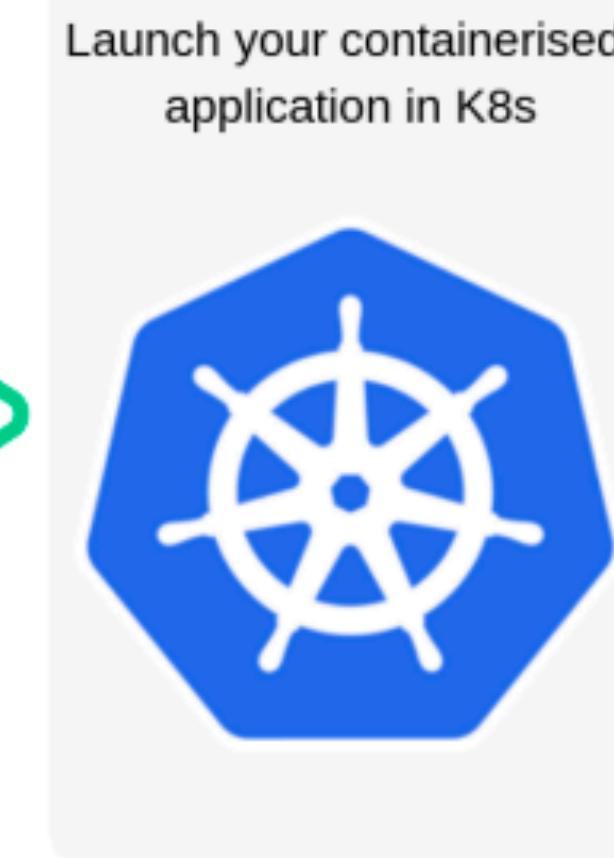
## Transition to Microservices



## Docker



## Kubernetes



# 一个微服务案例

# Istio

# 微服务框架现状

## 无服务治理类

专注于通信框架，RPC或消息队列模式，部分框架支持多语言开发

Apache Thrift™



## 单语言带服务治理类

在通信框架的基础上支持服务治理能力，单一编程语言实现，JAVA语言为主流



DUBBO

## ServiceMesh

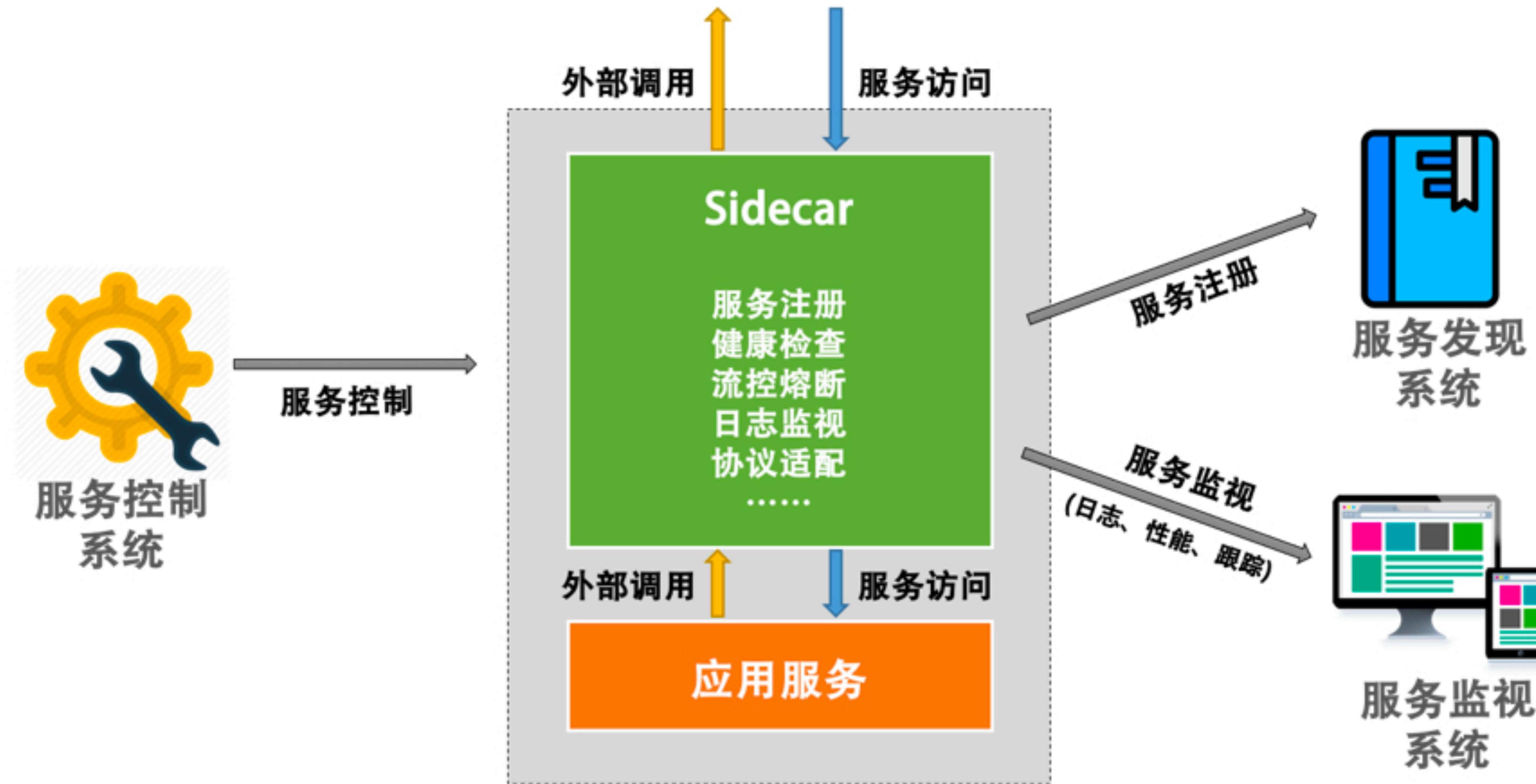
支持服务治理，通过SideCar模式解决多语言问题，目前处于发展成熟期



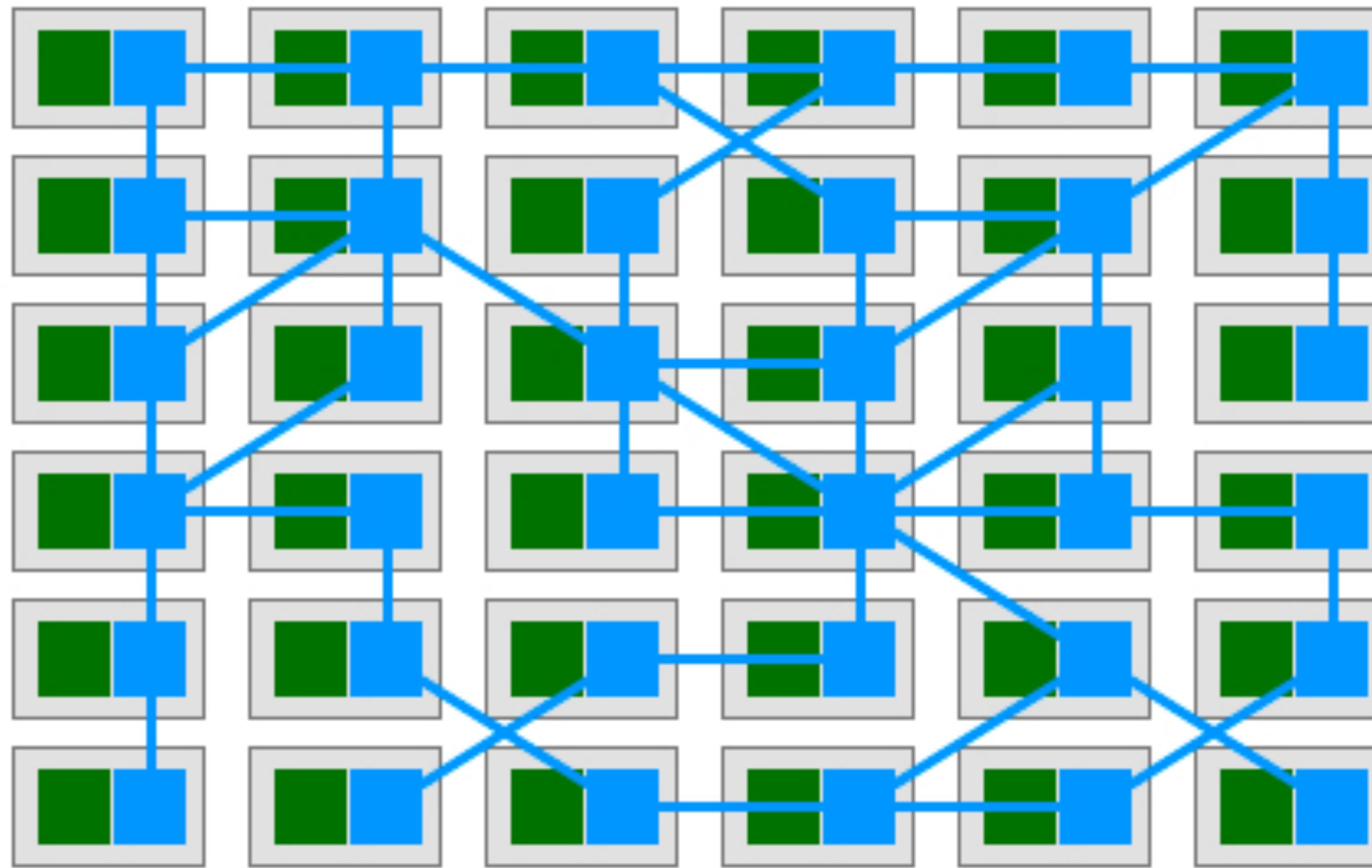
## 多语言带服务治理类

在通信框架的基础上支持服务治理能力，多种编程语言实现

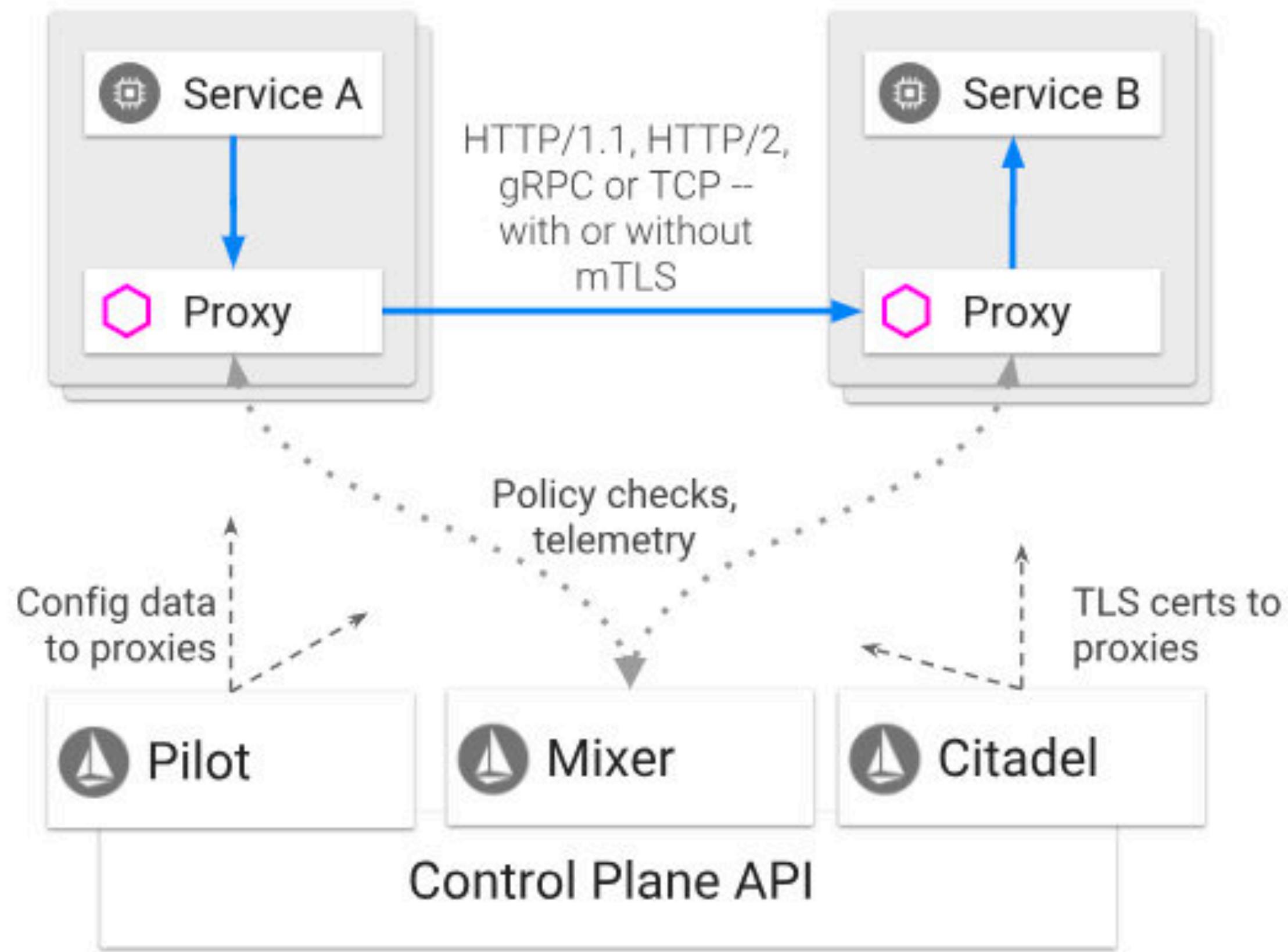
**TARS**  
Github.com/TarsCloud



# Sidecar



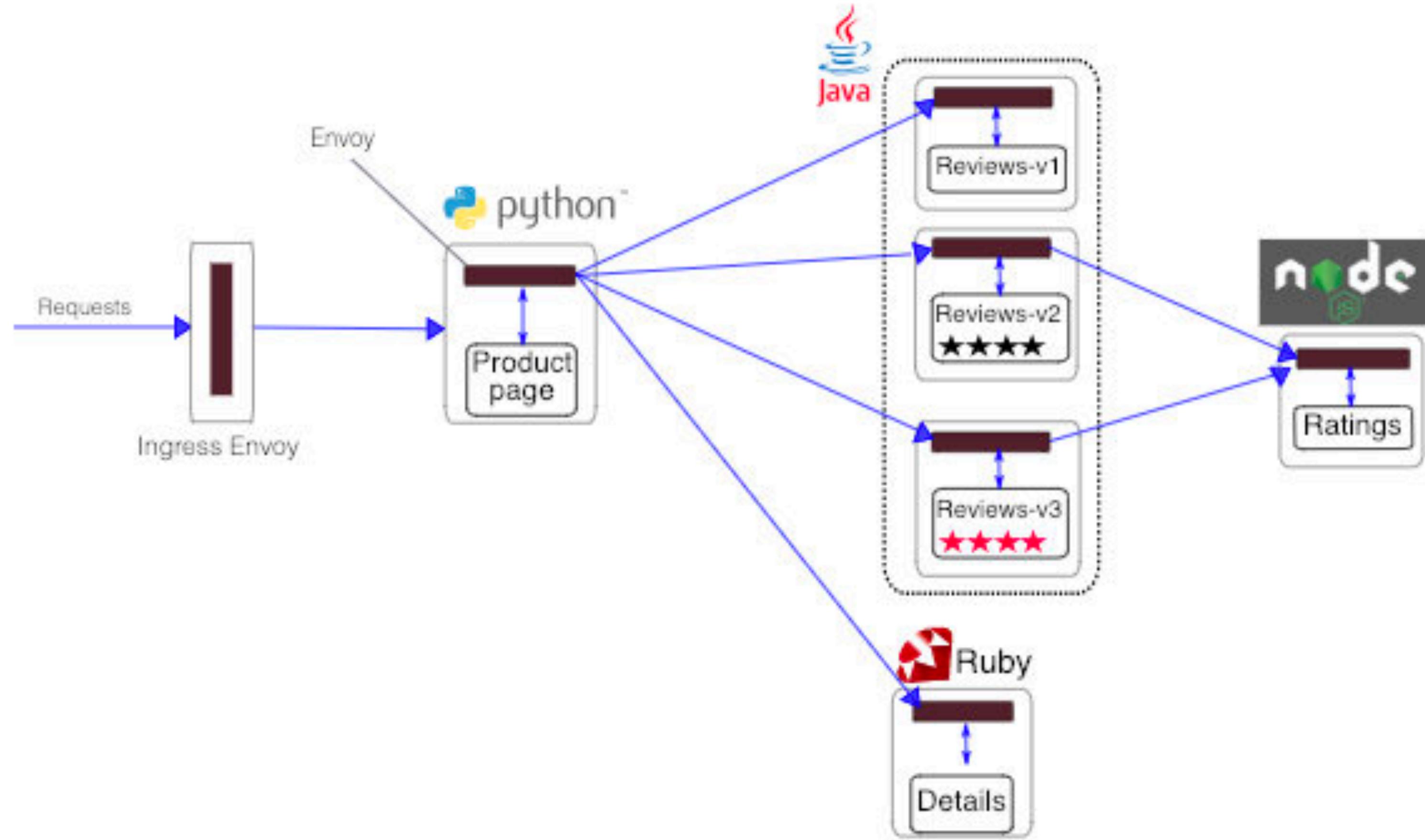
# Service Mesh



#### Istio Architecture

- 数据面
  - Sidecar
- 控制面
  - Pilot: 服务发现、流量管理
  - Mixer: 访问控制、遥测
  - Citadel: 终端用户认证、流量加密

# Istio Architecture



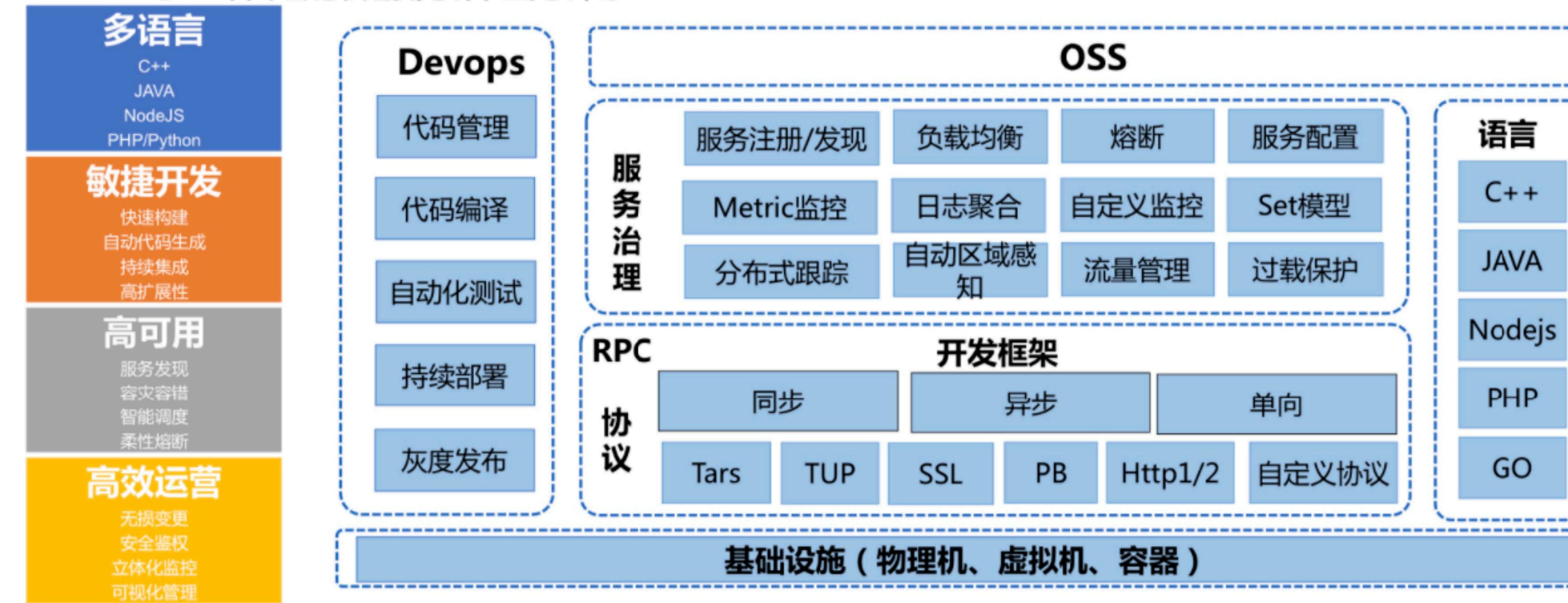
# 一个istio案例

# Sidecar 还是 Ambient?

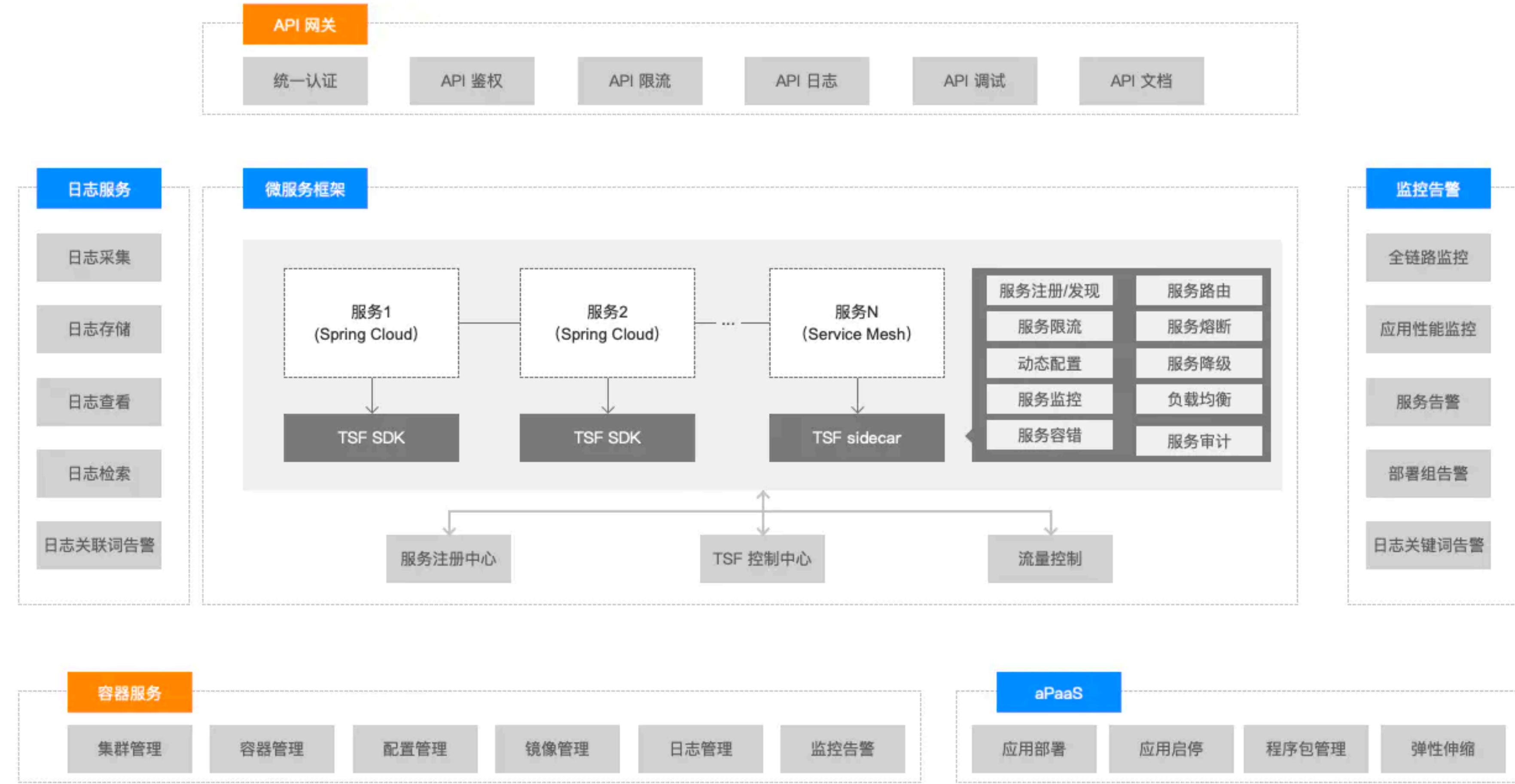
- Istio 服务网格在逻辑上分为数据平面和控制平面。
- 数据平面是一组代理，用于调解和控制微服务之间的所有网络通信。它们还收集和报告所有网格流量的可观测数据。
- 控制平面管理和配置数据平面中的代理。
- Istio 支持两种主要的数据平面模式：
  - Sidecar 模式，它会与您在集群中启动的每个 Pod 一起部署一个 Envoy 代理，或者与在虚拟机上运行的服务一同运行。
  - Ambient 模式，使用每个节点的四层代理，并且可选地使用每个命名空间的 Envoy 代理来实现七层功能。

# 腾讯自研微服务框架

“ TARS 是腾讯开源、基于 TARS 协议的高性能 RPC 框架，为开发和运维提供了一体化的微服务治理方案。 ”

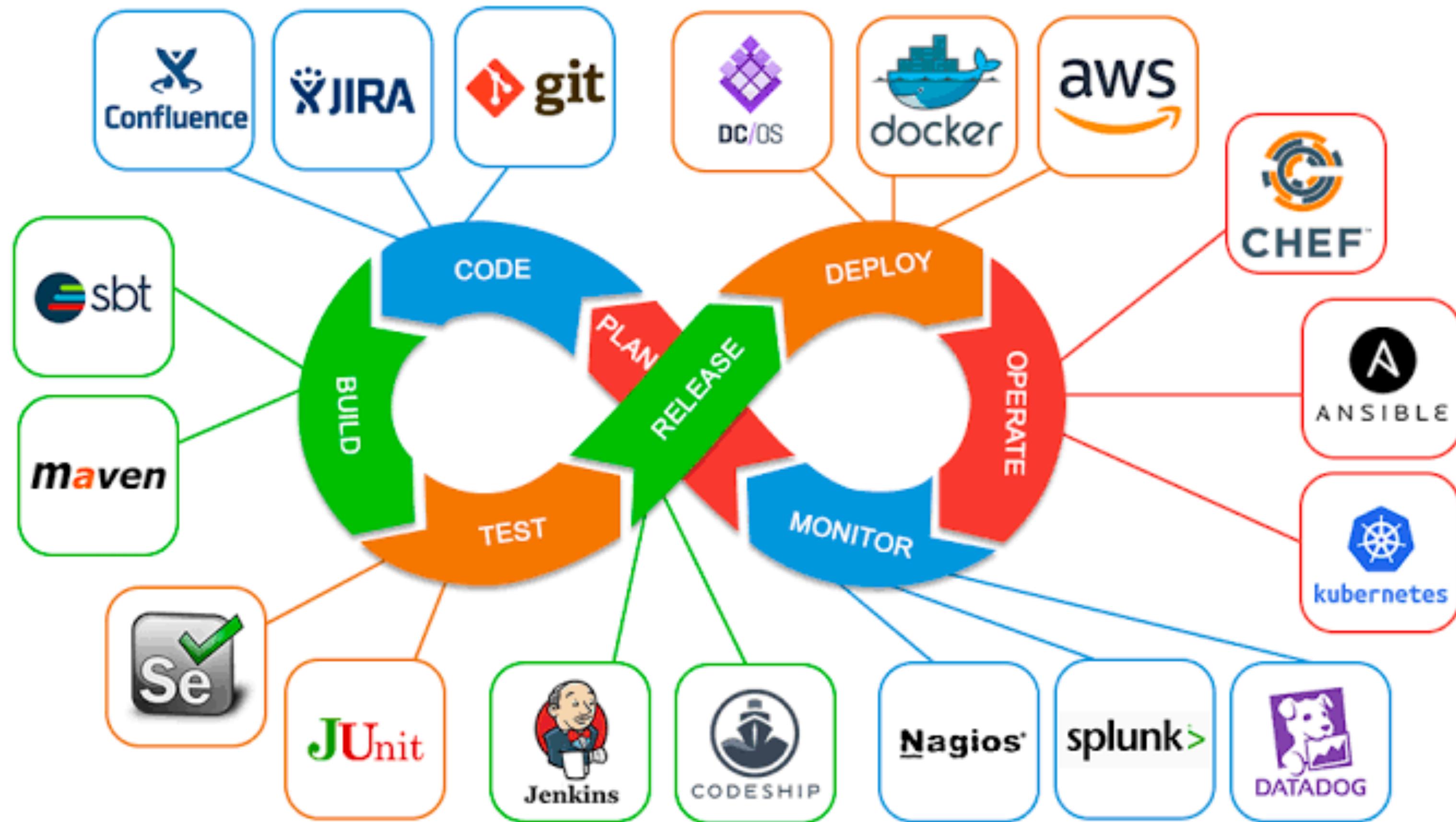


# TARS



# TSF 公有云平台

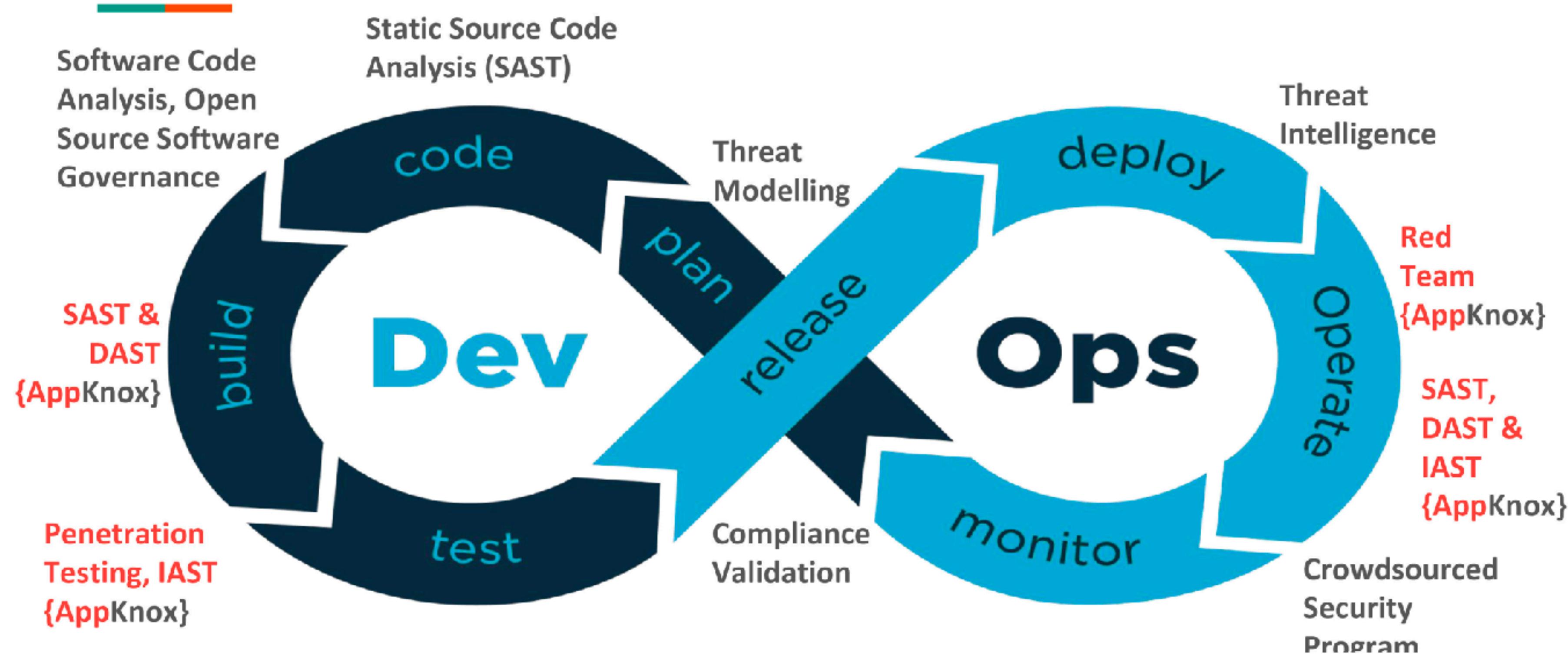
# DevOps



DevOps is as much about culture, as it is about the toolchain!

# Mindmap for Devsecops

appknox



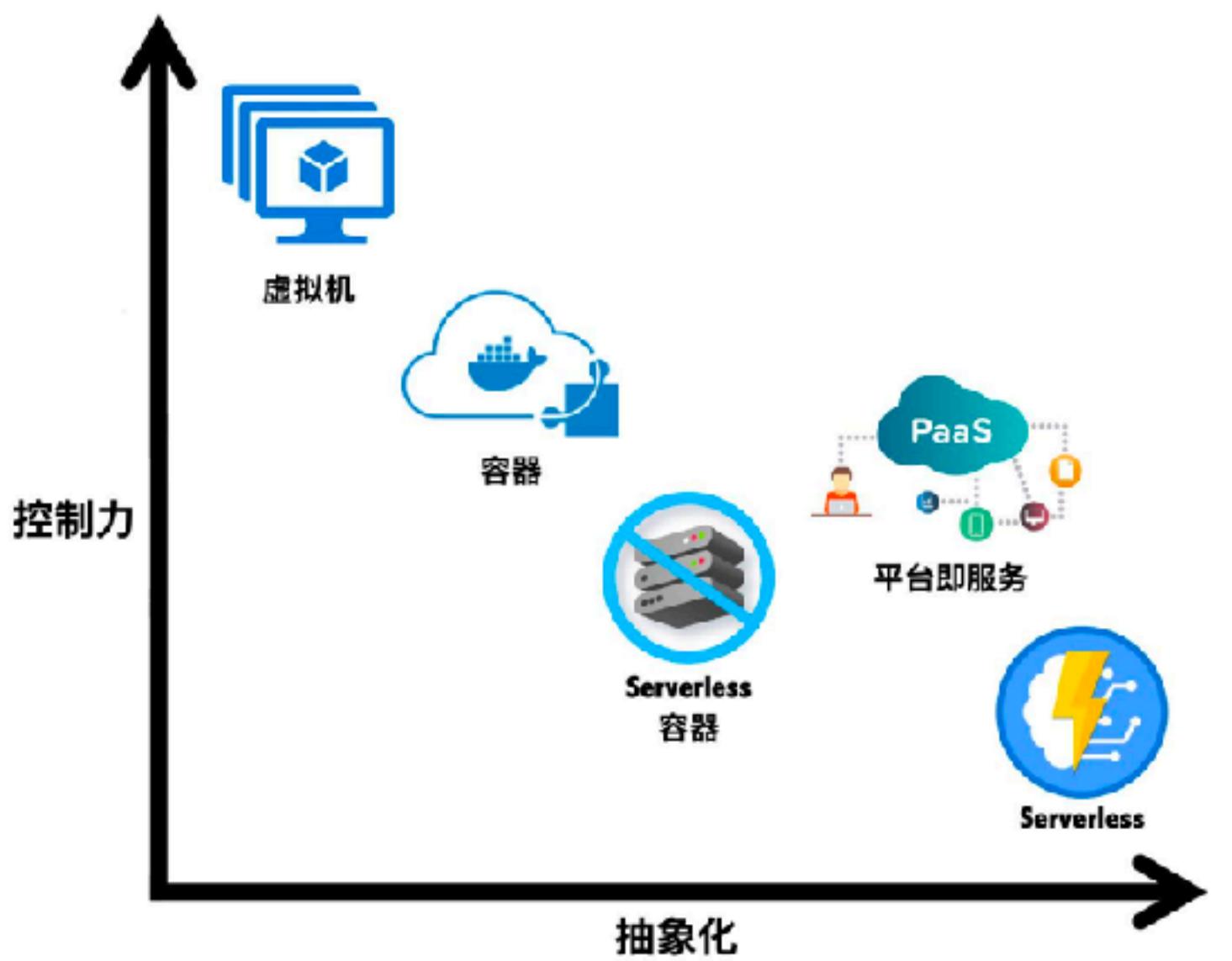
# DevSecOps



# BizDevOps

# Serverless

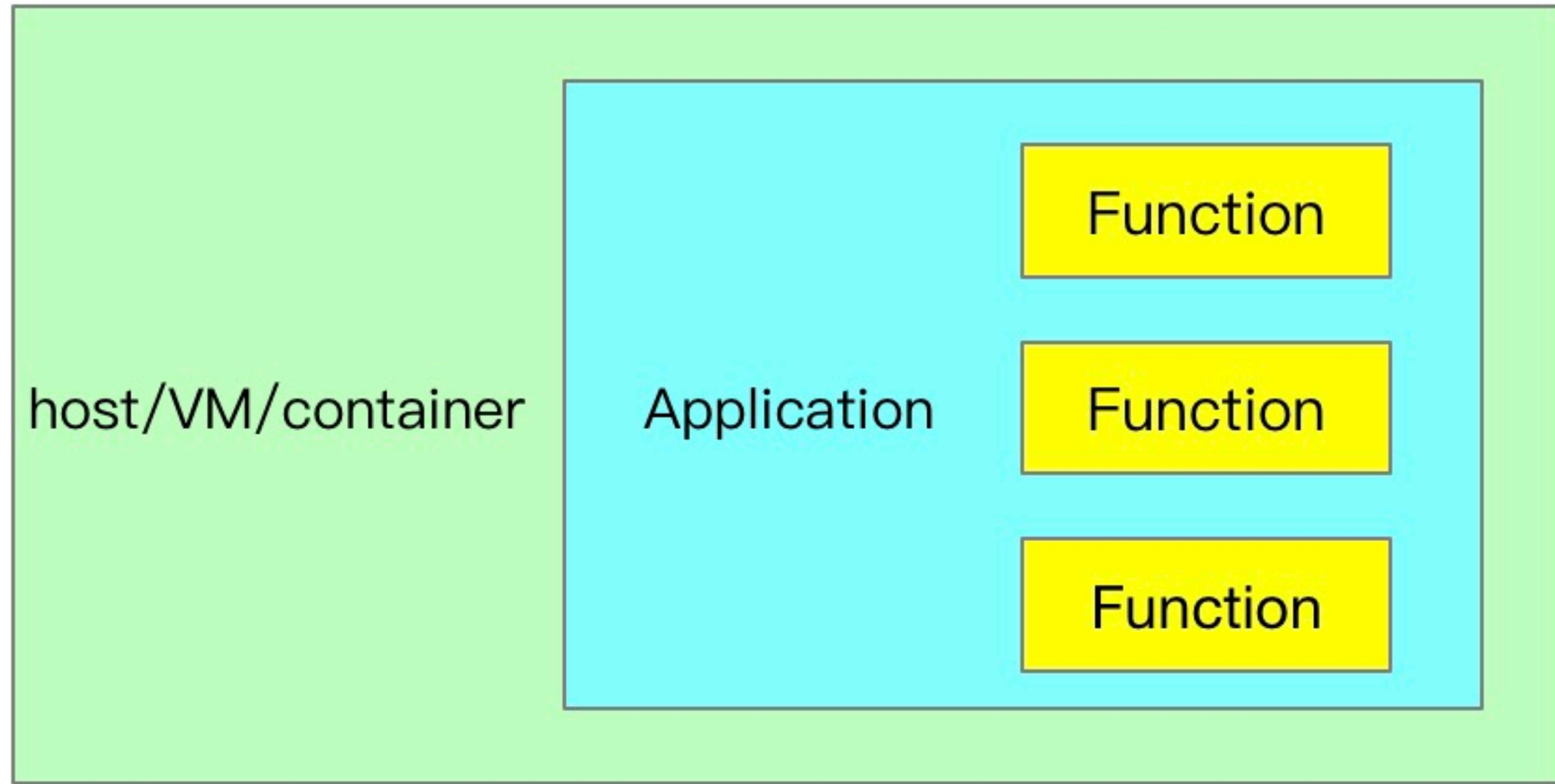
## 云计算全景图



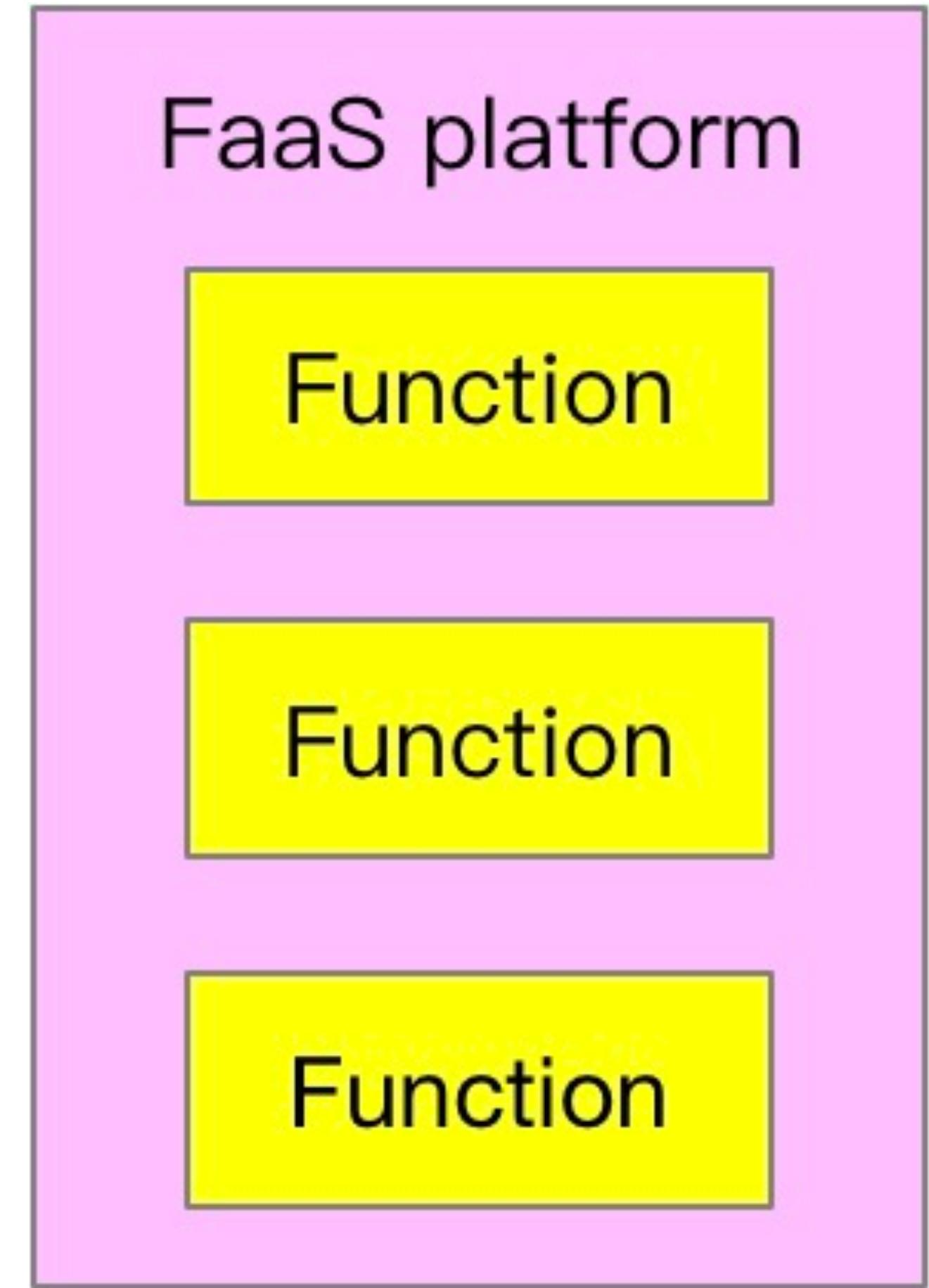
# 云计算全景

# Servless

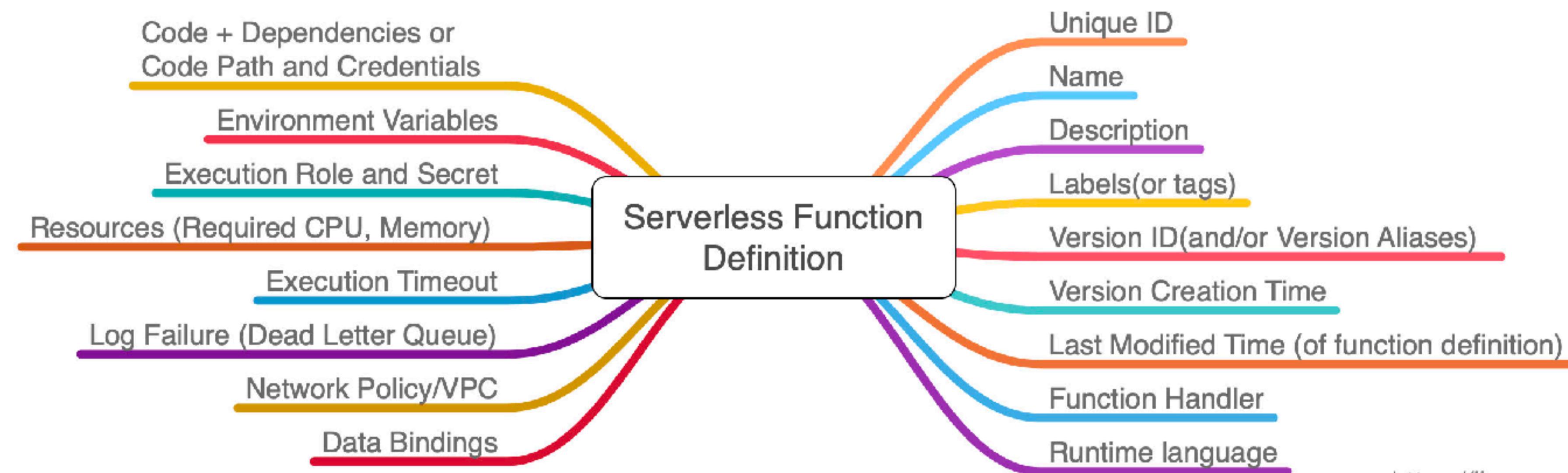
- 包含了两个领域BaaS (Backend as a Service) 和FaaS (Function as a Service) 。
- BaaS
  - BaaS (Backend as a Service) 后端即服务，一般是一个个的API调用后端或别人已经实现好的程序逻辑，比如身份验证服务Auth0，这些BaaS通常会用来管理数据，还有很多公有云上提供的我们常用的开源软件的商用服务，比如亚马逊的RDS可以替代我们自己部署的MySQL，还有各种其它数据库和存储服务。
- FaaS
  - FaaS (Functions as a Service) 函数即服务，FaaS是无服务器计算的一种形式，当前使用最广泛的是AWS的Lambada。



# 传统服务端软件的运行环境

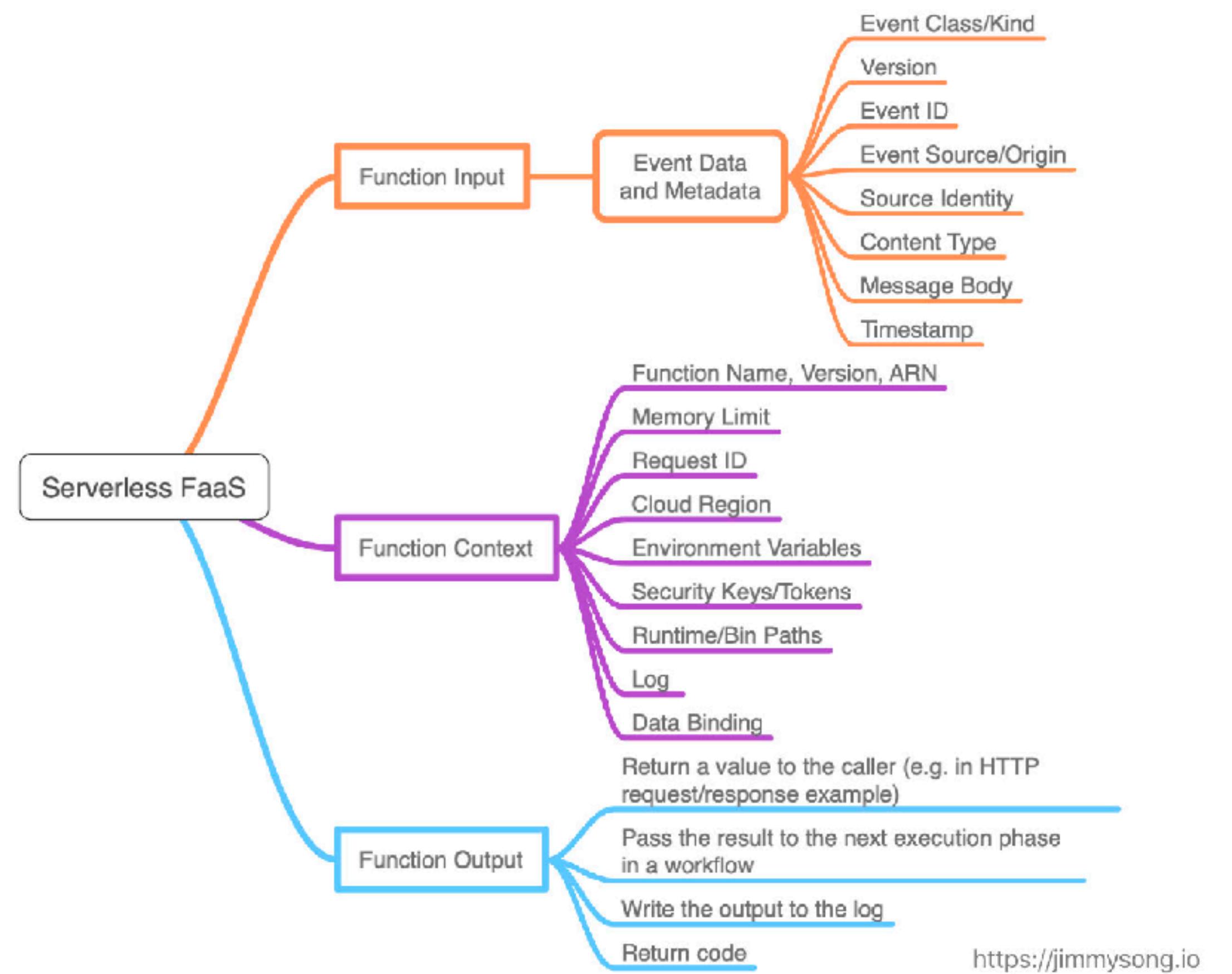


# FaaS应用架构



<https://jimmysong.io>

# 函数定义



<https://jimmysong.io>

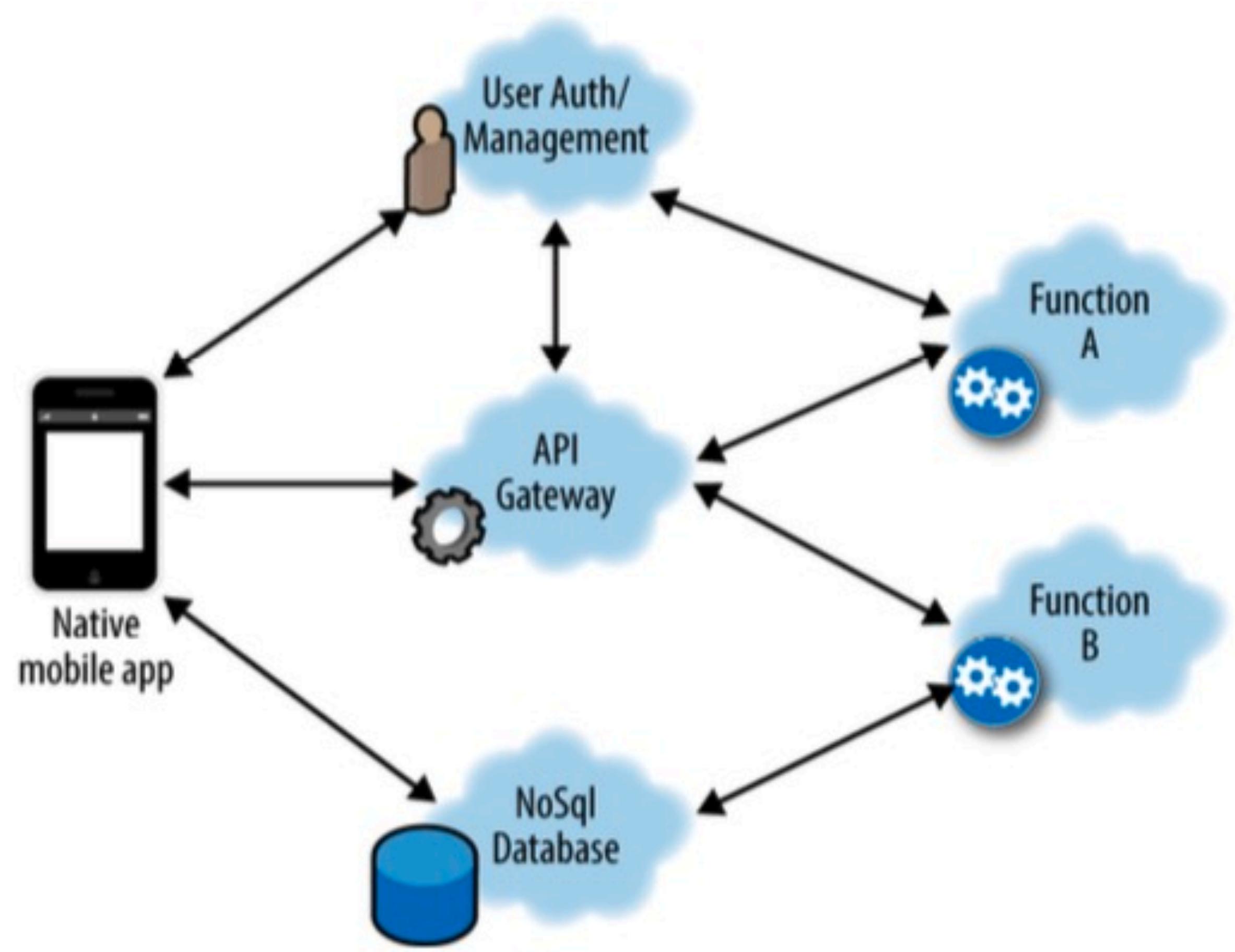
# FaaS 中的函数

# 应用场景

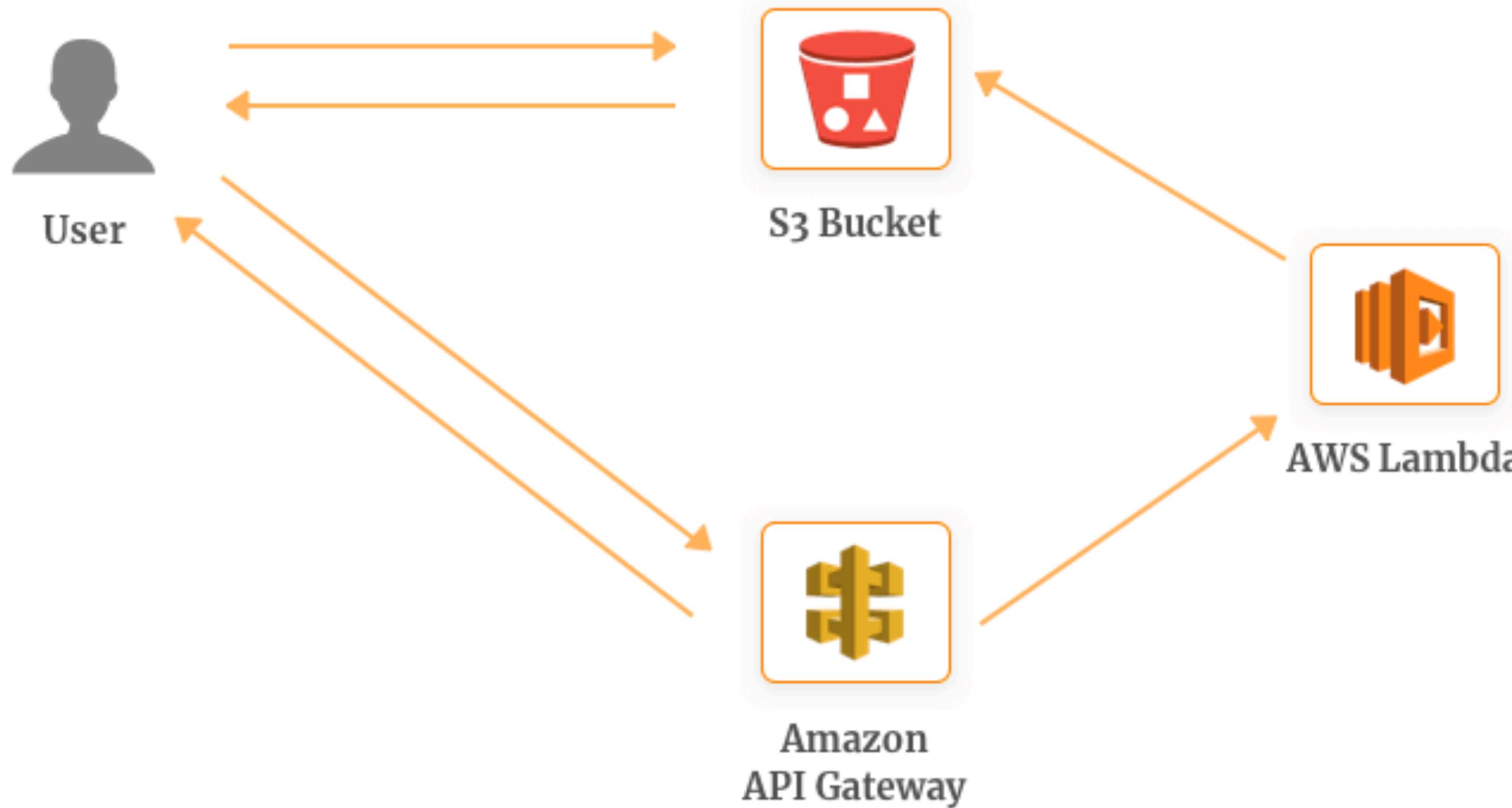
- 虽然 Serverless 的应用很广泛，但是其也有局限性，Serverless 比较适合以下场景：
  - 异步的并发，组件可独立部署和扩展
  - 应对突发或服务使用量不可预测（主要是为了节约成本，因为 Serverless 应用在不运行时不收费）
  - 短暂、无状态的应用，对冷启动时间不敏感
  - 需要快速开发迭代的业务（因为无需提前申请资源，因此可以加快业务上线速度）
- Serverless 的使用场景示例如：
  - ETL
  - 机器学习及 AI 模型处理
  - 图片处理
  - IoT 传感器数据分析
  - 流处理
  - 聊天机器人



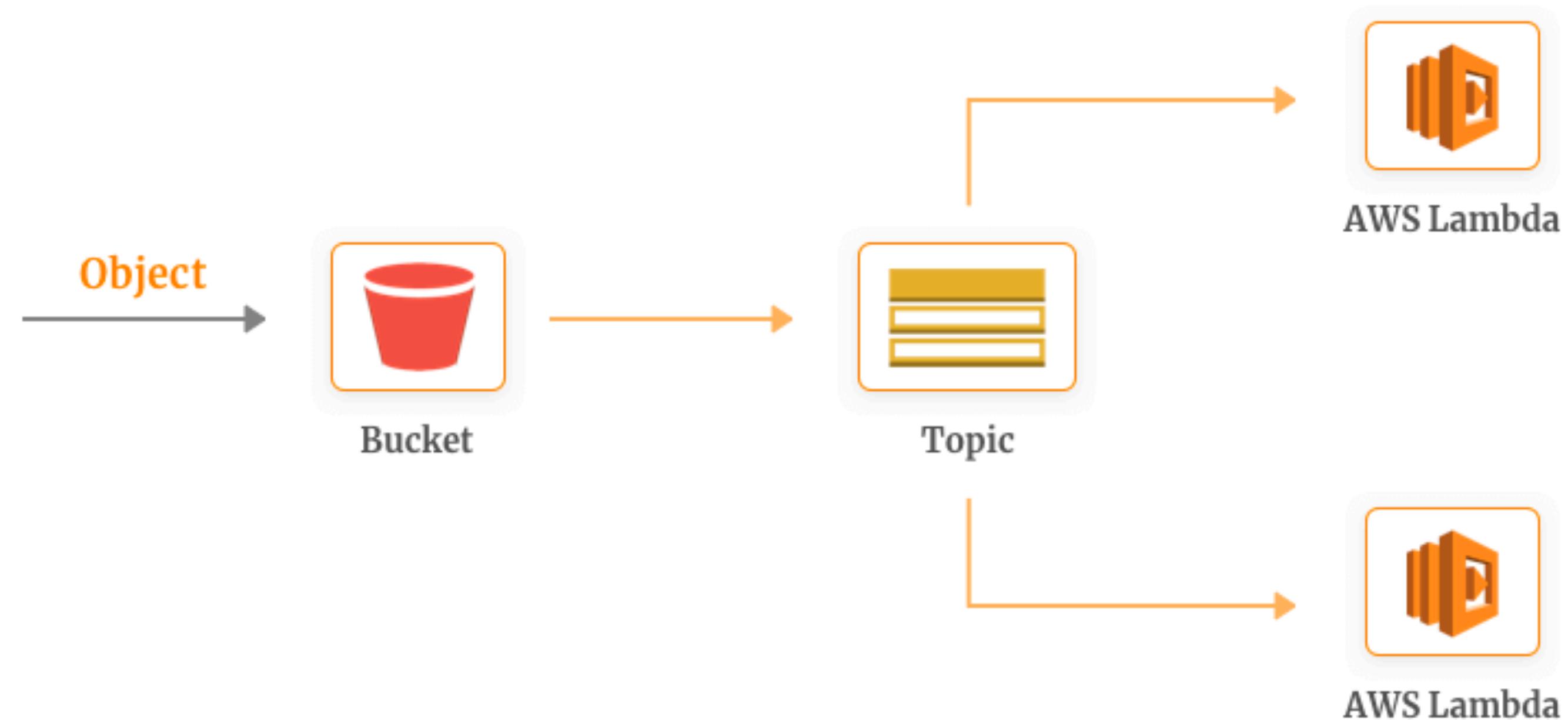
传统架构



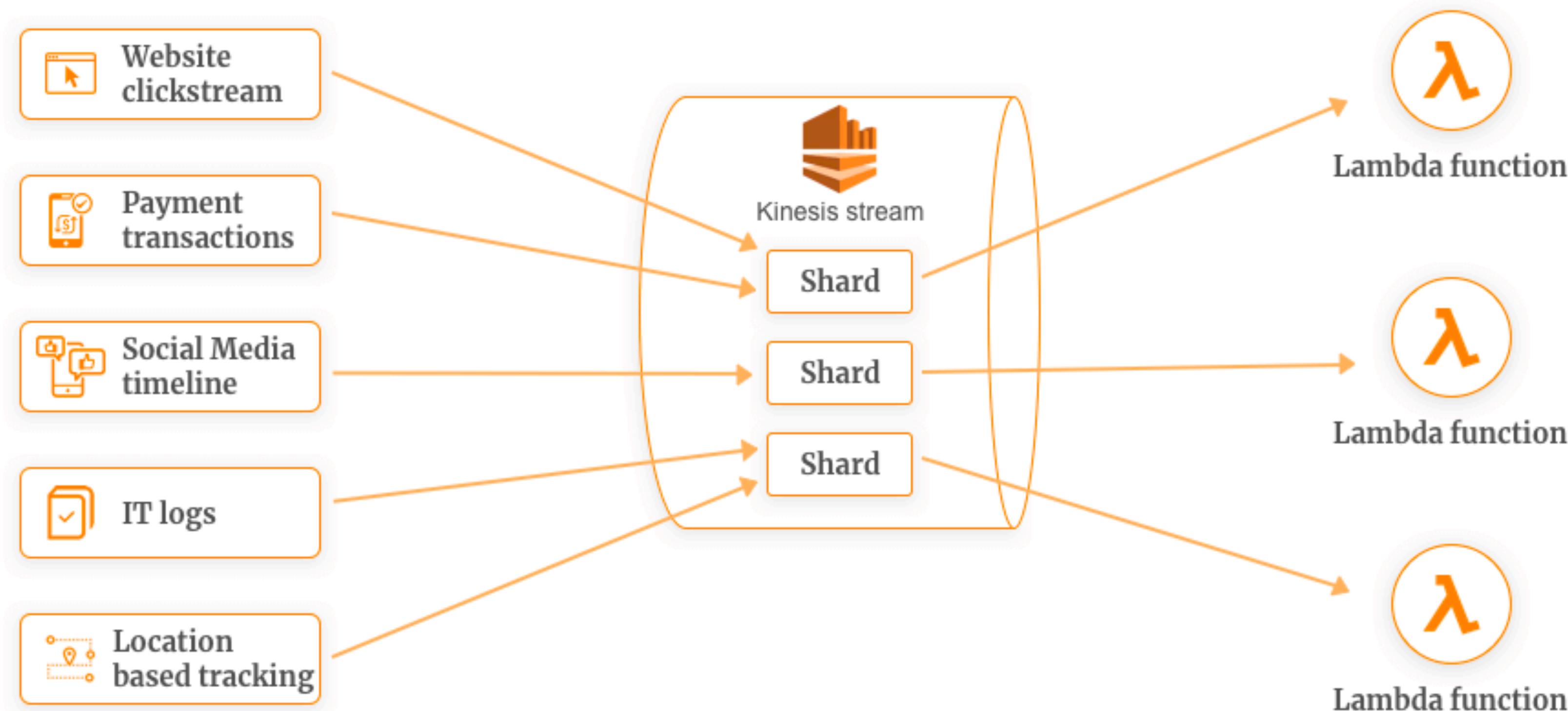
# Serverless 架构



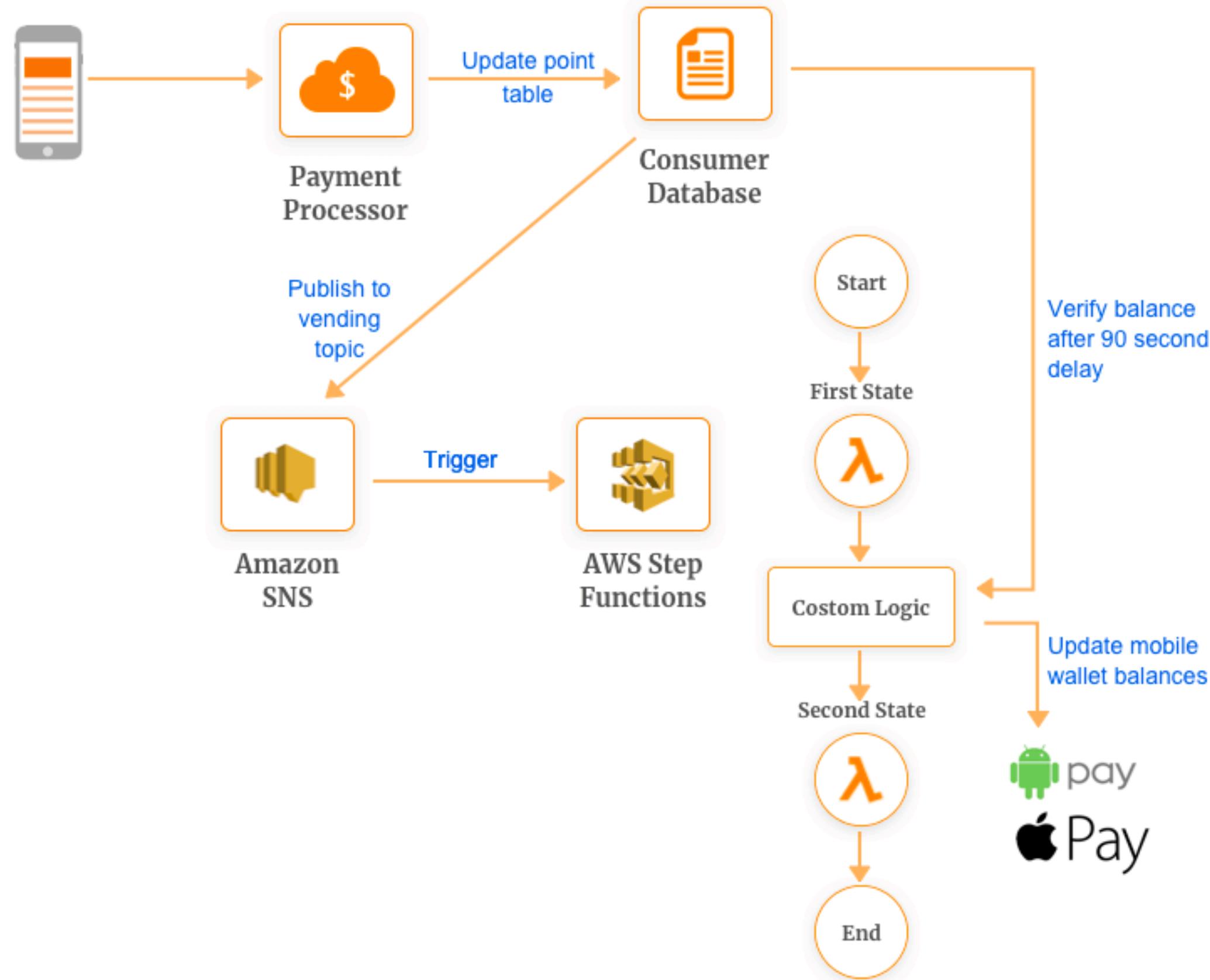
# #1. AWS Lambda Example showing Media Transformation



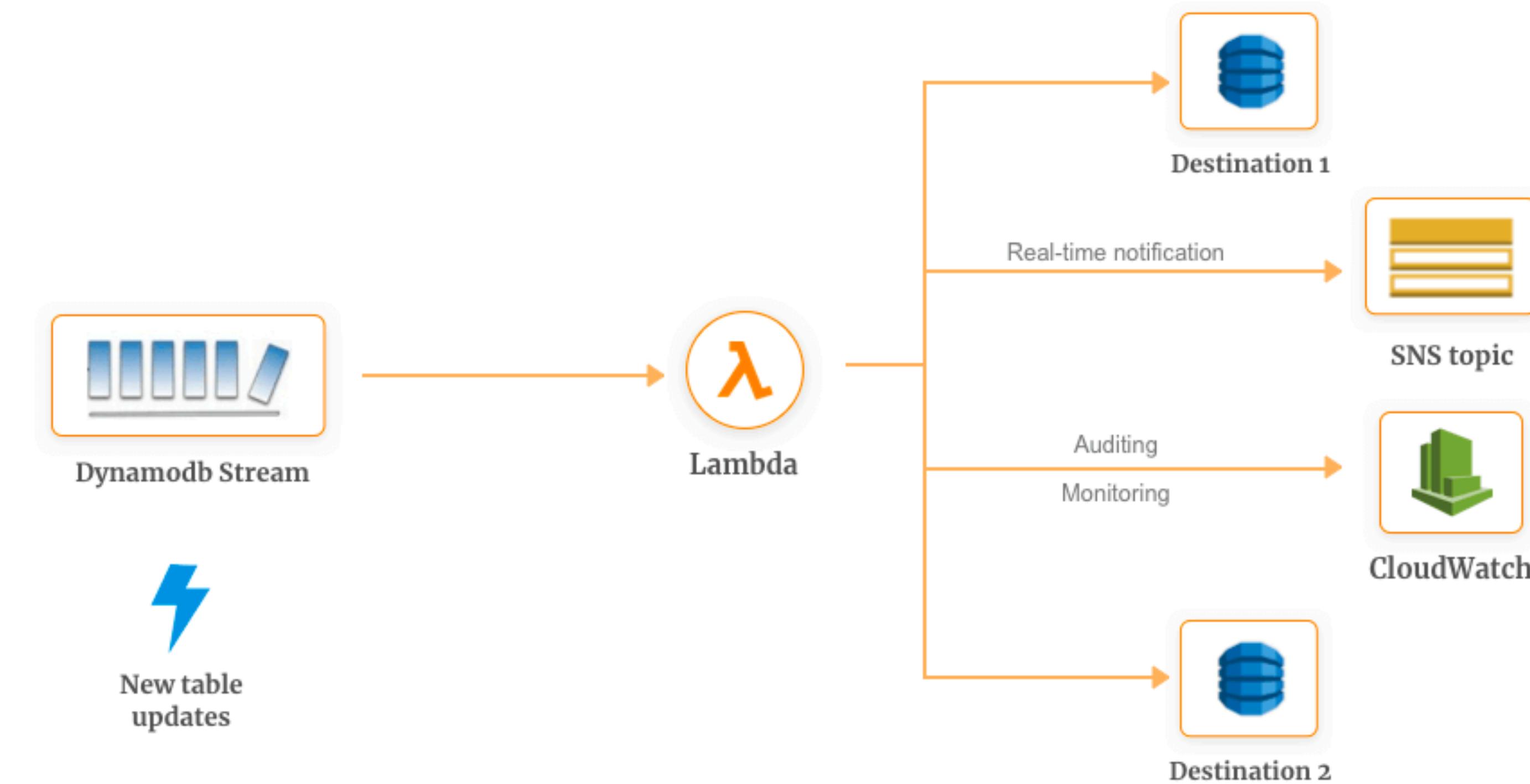
## #2. Deriving Multiple Data Format from Single Source



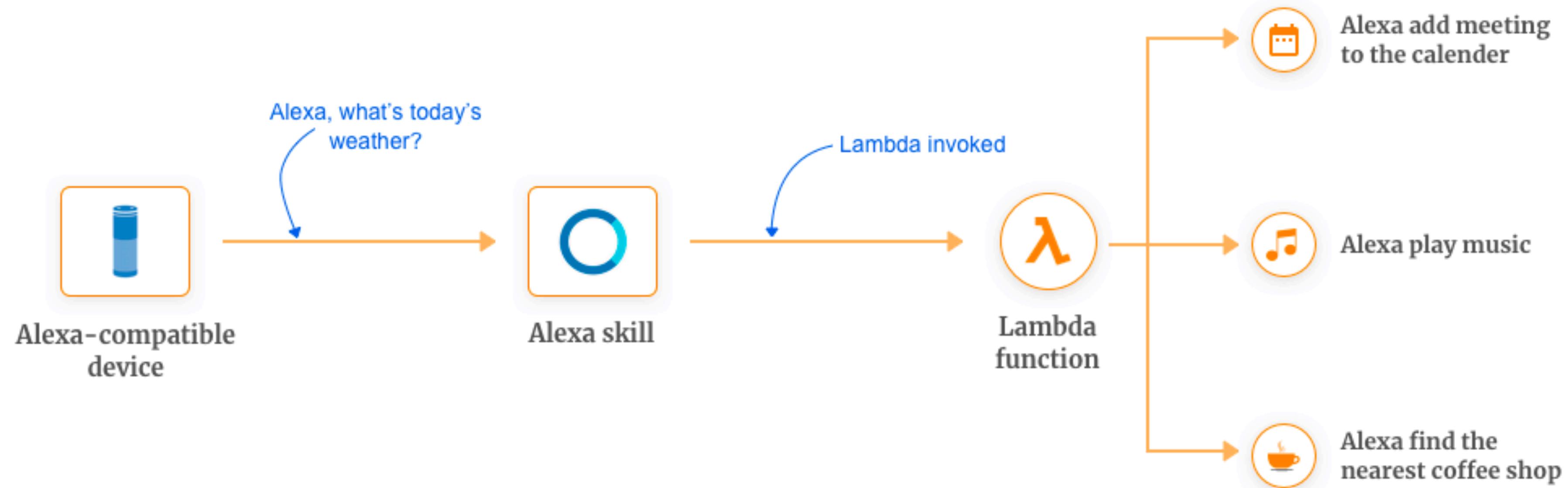
## #3. Real-Time Data Processing Example using AWS Lambda



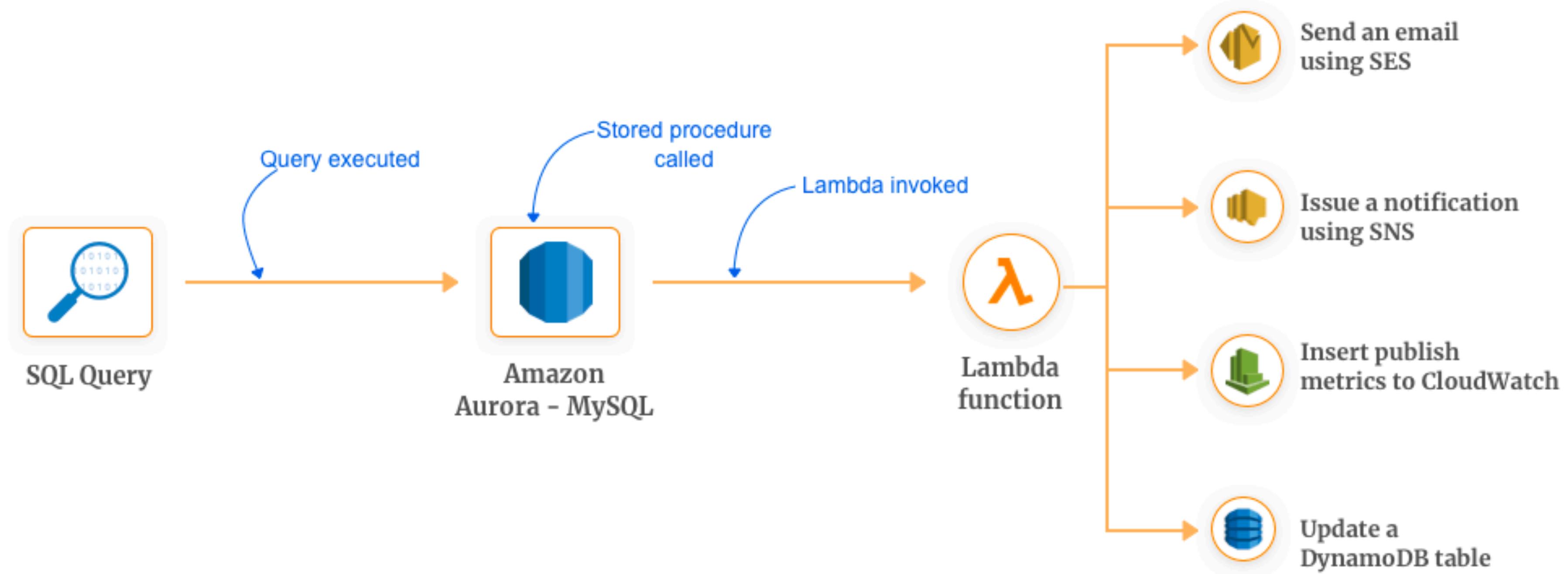
## #4. Custom Logic Workflows using AWS Lambda



## #5. Change Data Capture with AWS Lambda

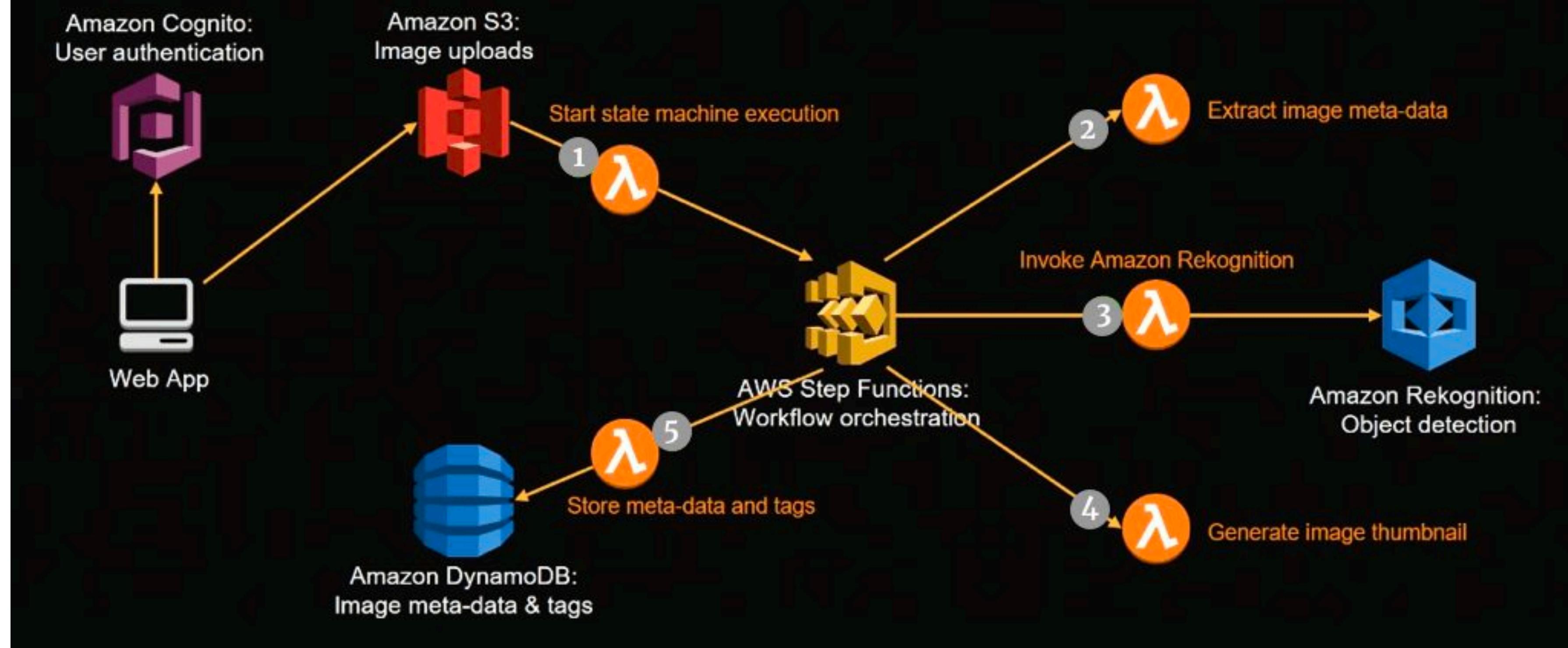


## #6. AWS Lambda Example showing Custom Alexa Skills

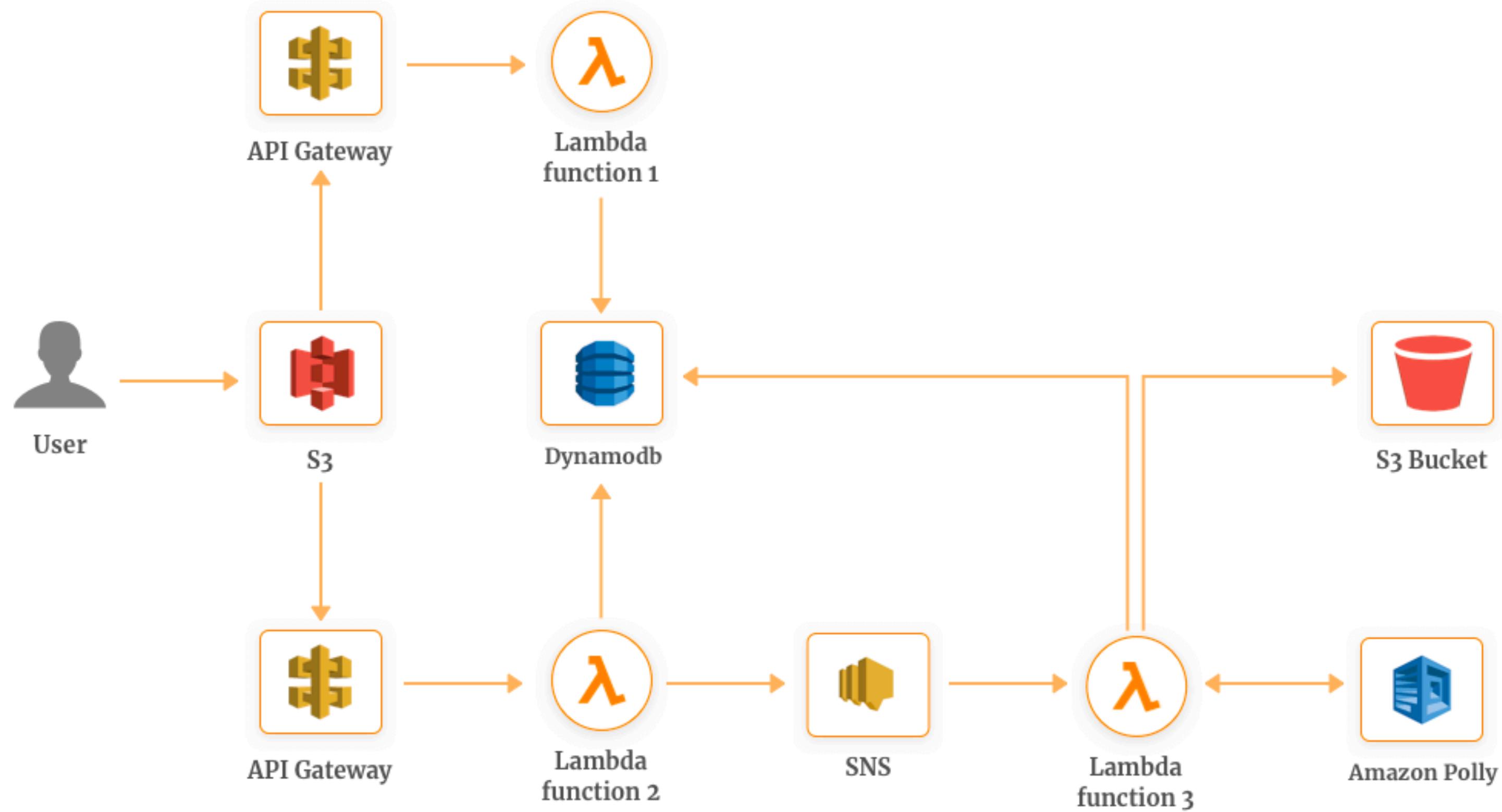


## #7. Automated Stored Procedures using AWS Lambda

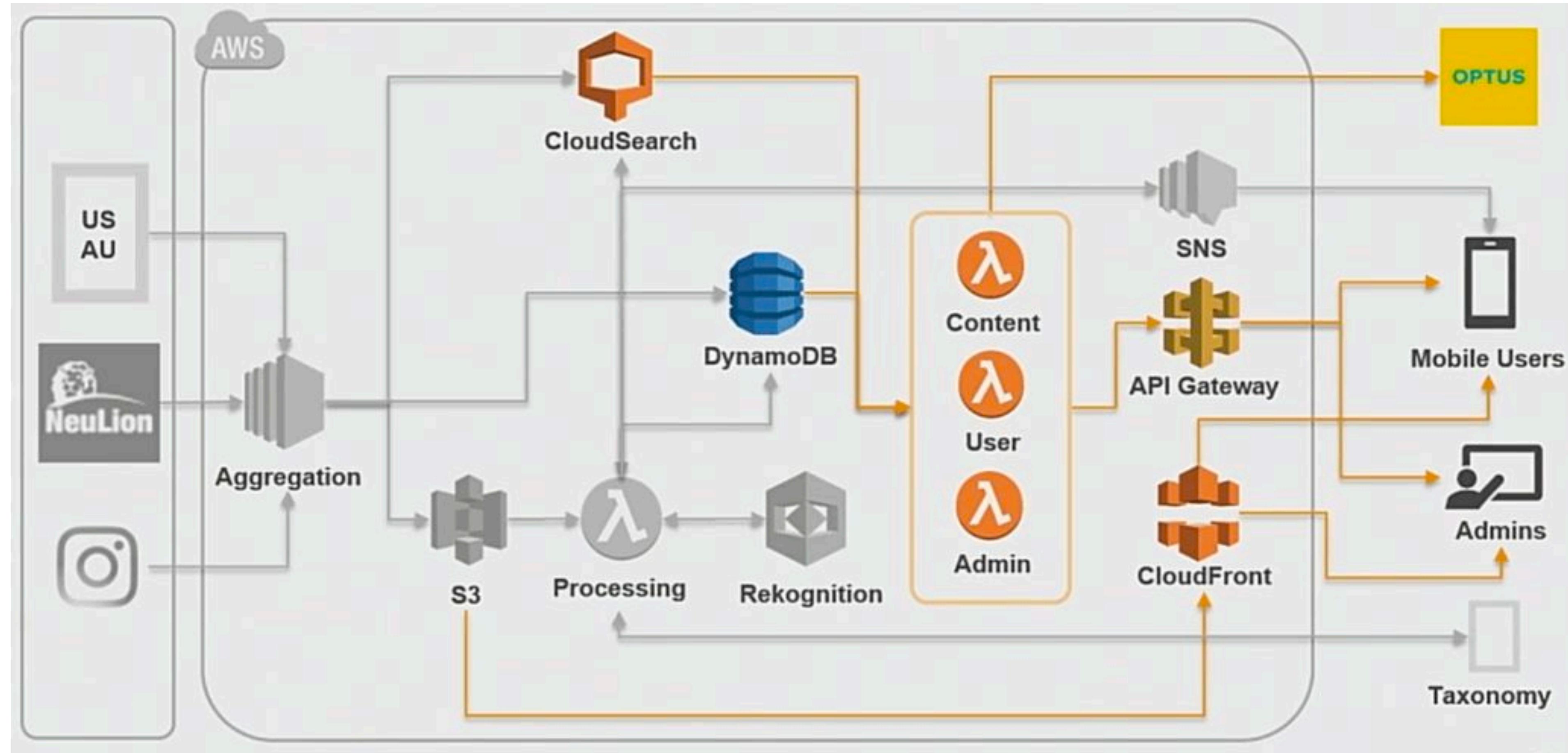
# Image recognition and processing



# #8. Serverless Image Recognition Engine



# #9 Serverless Text-to-Speech Example



# #10 Personalized Content Delivery through AWS Lambda

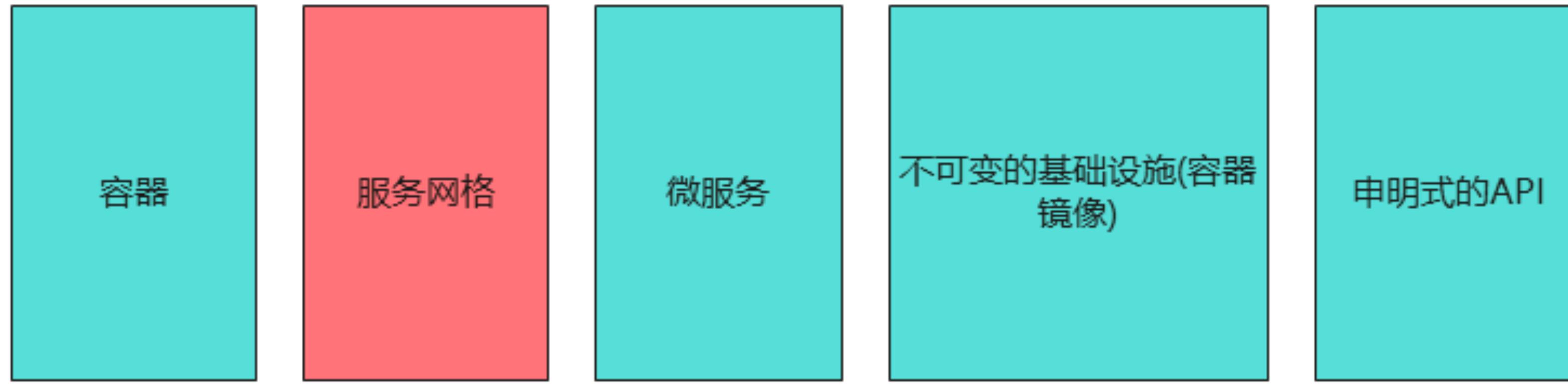
云原生

# 历史

- 2013年，Pivotal公司的Matt Stine首次提出原生云(CloudNative)的概念，用于区分为云而设计的应用和云上部属传统应用。
- 2015年，Matt Stine在《迁移到云原生架构》一书中定义了符合原生云架构的几个特征：12因素、微服务、自敏捷架构、基于API协作、抗脆弱性；
- 2015年云原生计算基金会(CNCF)成立,CNCF作为一个厂商中立的基金会，致力于云原生应用推广和普及。
- 2017年，Matt Stine将原生云架构归纳为模块化、可观察、可部署、可测试、可替换、可处理6特质;而Pivotal最新官网对云原生概括为4个要点：DevOps+持续交付+微服务+容器。



云原生系统  
(可弹性、可管理、可观察、自动化、容错)

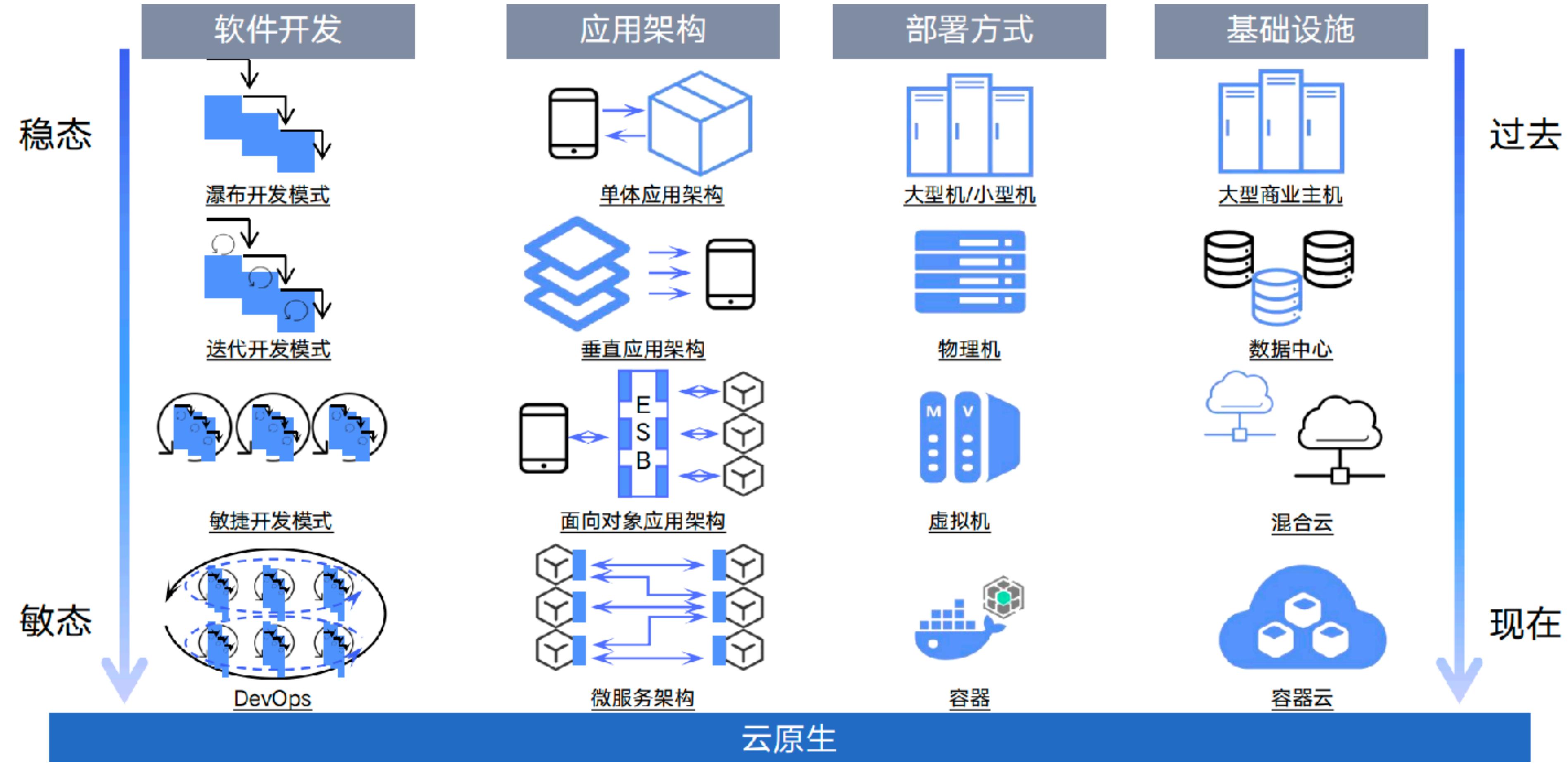


Cloud 环境  
公有云、私有云、混合云

# 云原生技术

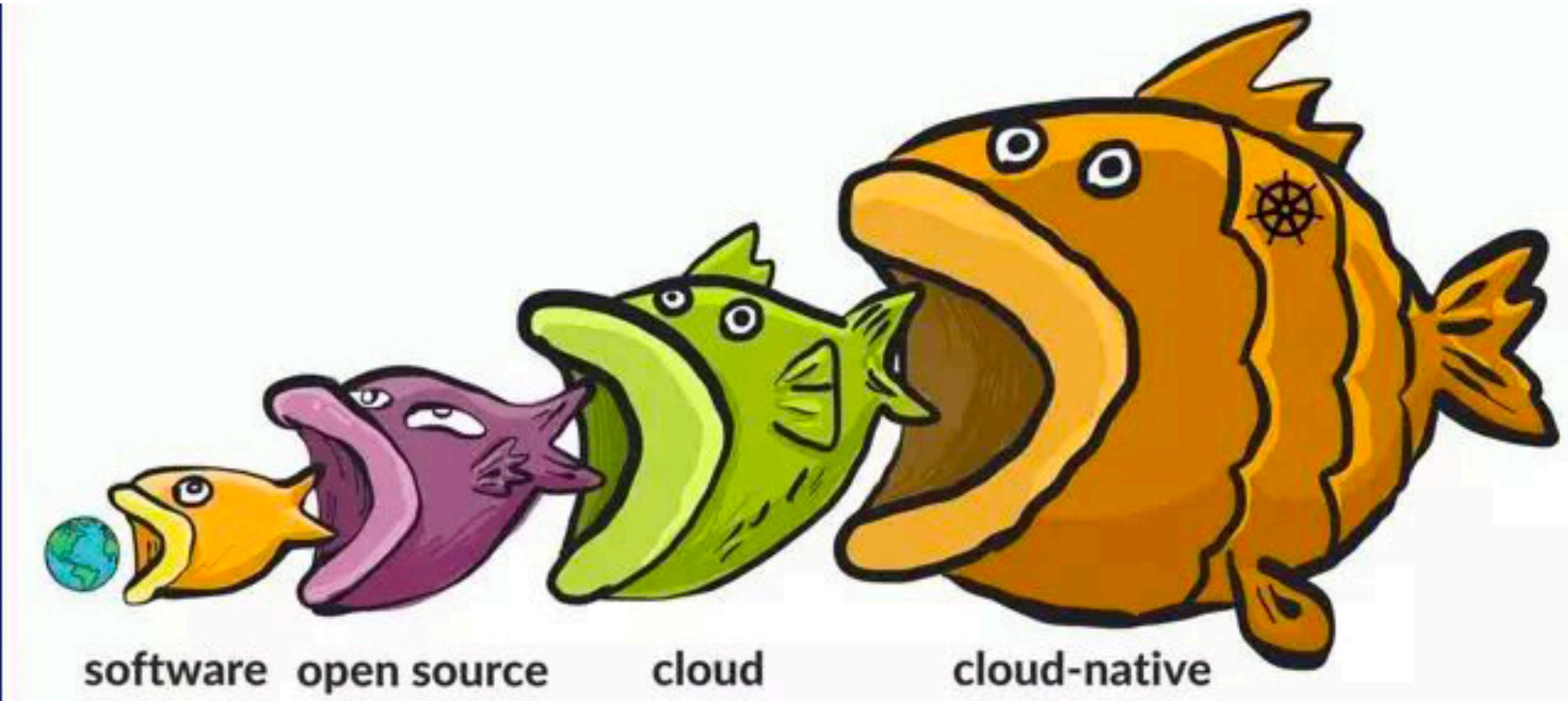
# 定义

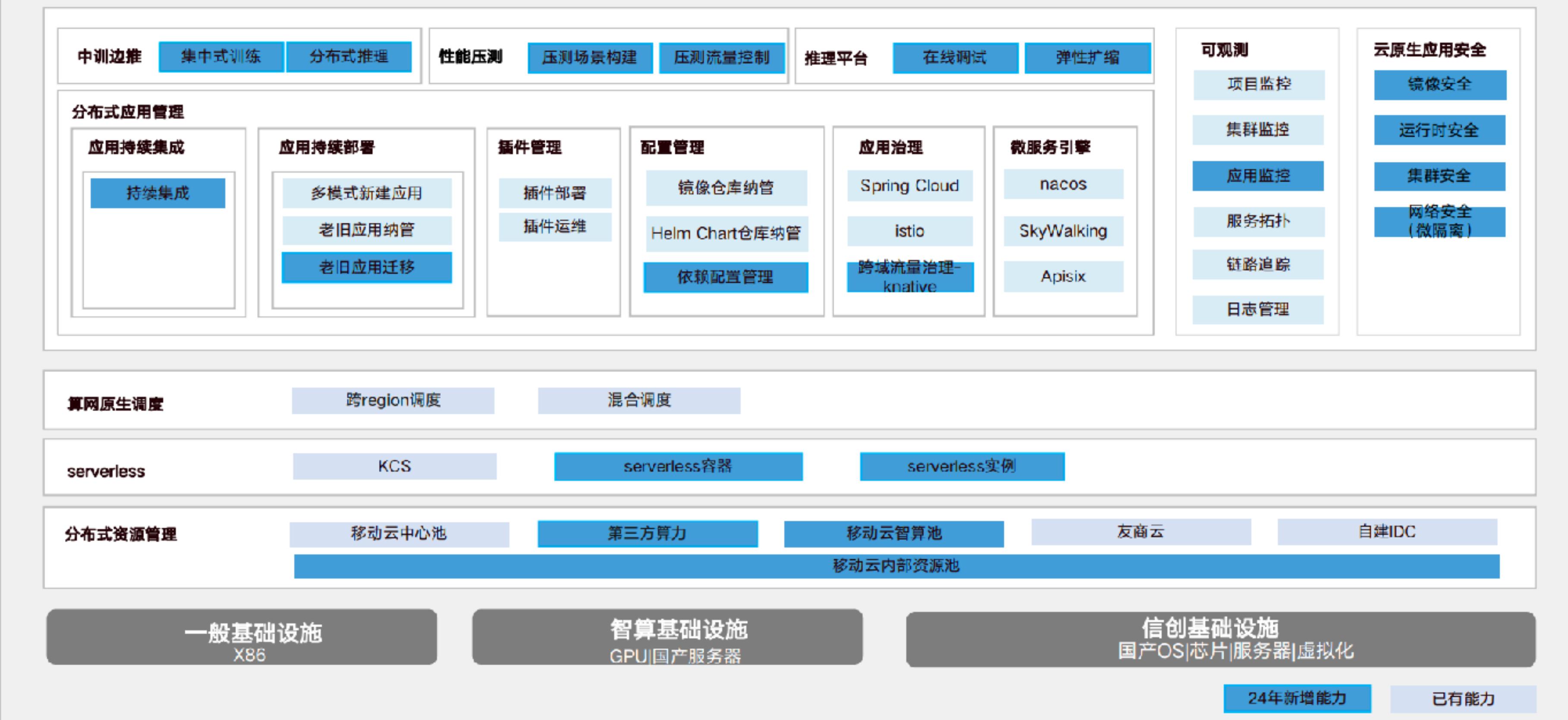
- Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.
- 云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式 API。
- These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.
- 这些技术能够构建容错性好、易于管理和便于观察的松耦合系统。结合可靠的自动化手段，云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更。
- The Cloud Native Computing Foundation seeks to drive adoption of this paradigm by fostering and sustaining an ecosystem of open source, vendor-neutral projects. We democratize state-of-the-art patterns to make these innovations accessible for everyone.
- 云原生计算基金会（CNCF）致力于培育和维护一个厂商中立的开源生态系统，来推广云原生技术。我们通过将最前沿的模式民主化，让这些创新为大众所用。
-



# 云原生的演变历史

“ Software Cloud Native  
is Eating the World.





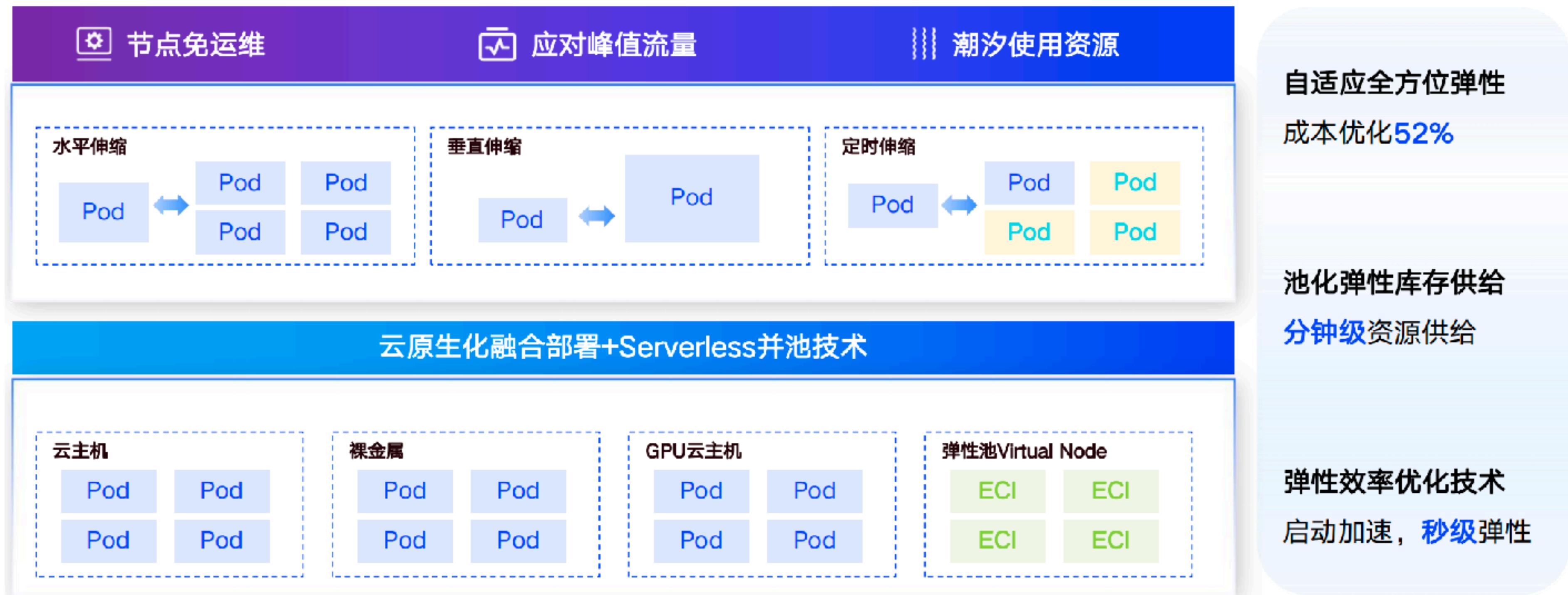
# 移动云CNP架构

移动云秉承“算力网络”理念，自主研发了无需关心资源、无论何时何地、业务快速上云的分布式云原生星罗算力基础设施，具备分布式资源管理、serverless、统一资源调度、现代化应用开发等核心能力，实现资源高效供给、无论何时何地、业务快速上云。



# 云原生架构

攻克业界“卡脖子”的serverless并池技术难题，构建全栈serverless技术体系。面向资源层，基于云主机+裸金属池化资源，构建弹性库存供给；面向应用层，根据负载情况和资源供给情况，智能地调整应用的调度容量。满足弹得起(虚拟实例库存保障)、弹得快(弹性效率高)、弹得准(精益成本不浪费)！



# 资源高效供给

移动云云原生计算产品线提供专为云原生应用打造的计算环境：容器服务、容器镜像服务。

- 容器服务基于Kubernetes技术，依托移动云基础资源，构建高性能、可扩展的Kubernetes集群，为用户容器应用提供一站式生命周期管理，助力用户应用轻松上云。
- 容器镜像服务是一种简单易用、安全可靠的镜像管理服务，提供云原生资产的安全托管和全生命周期管理，与容器服务、对象存储等产品无缝集成，打造云原生应用一站式解决方案。

### ➤ 容器镜像服务

- 高效、完善的云原生资产管理
- 多场景多规格（免费版、企业版）
- 云内镜像中心（容器服务、函数计算、代码托管等）

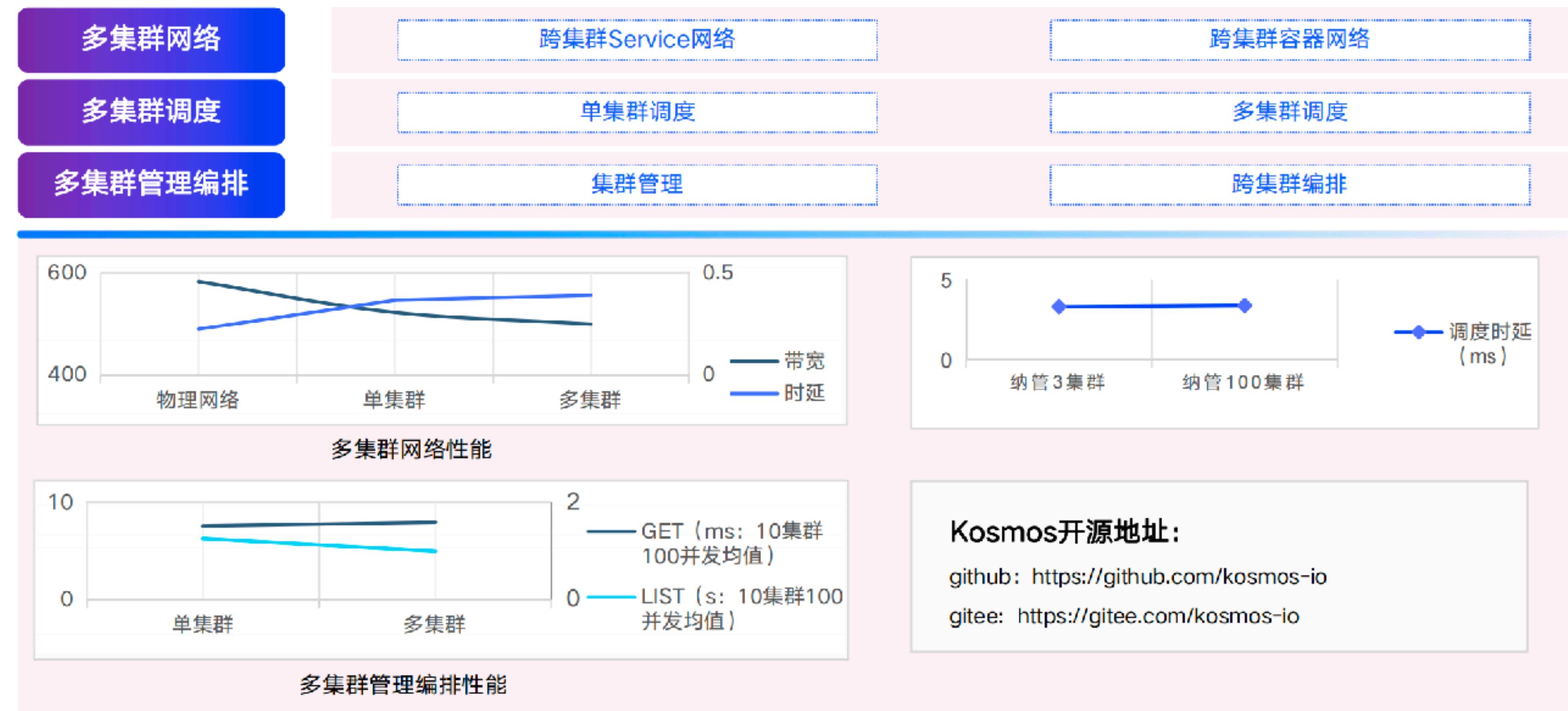
### ➤ 容器服务

- 原生K8s服务（线上线下一致体验）
- 多种集群类型（专有、托管）
- 多算力（x86、gpu、arm）
- 多场景（混合云、边缘云等）



# 容器镜像服务、容器服务

构建自主掌控的**分布式云原生regionless生态**, 开源**分布式多集群KOSMOS**, 全方位覆盖分布式云原生多集群管理编排、多集群调度、多集群网络等能力, 实现分布式集群精准调度, 打破传统应用的地域限制。



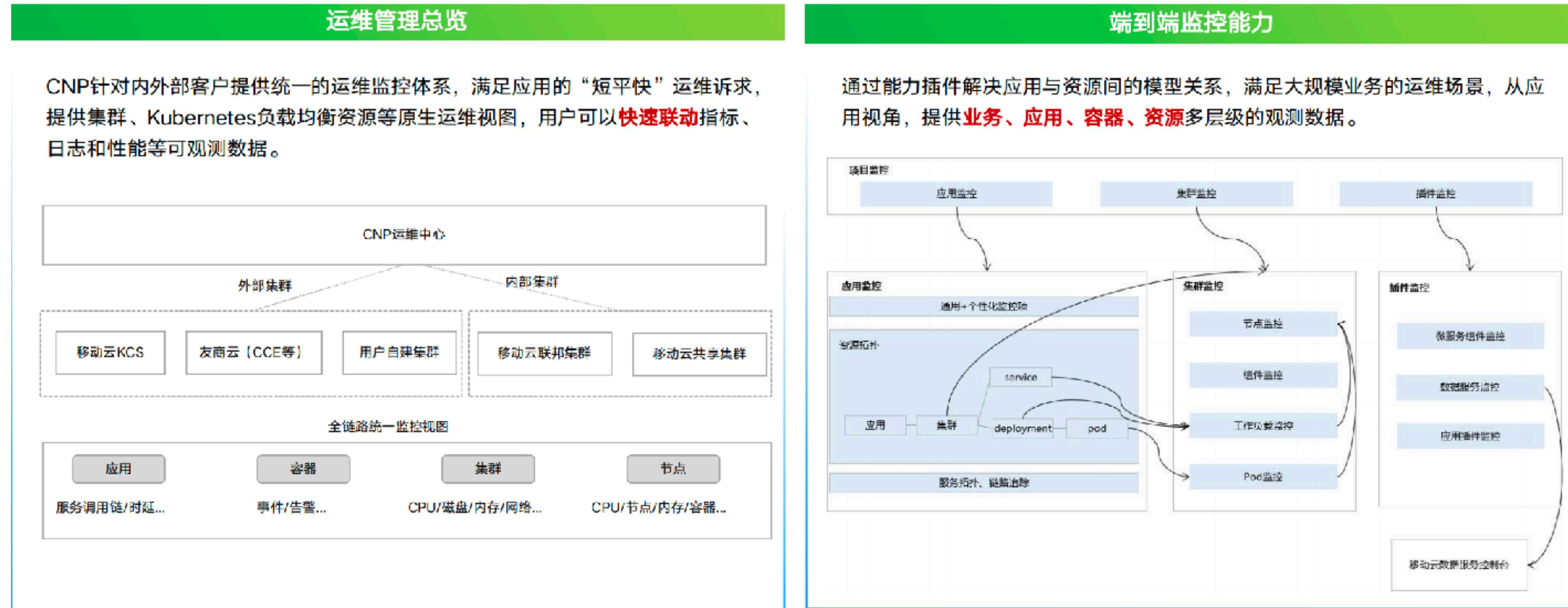
# 开源分布式多集群KOSMOS

CNP将应用生态统一，业务跨云、跨地域高效部署，提升应用开发、部署的效率，降低应用全生命周期成本。



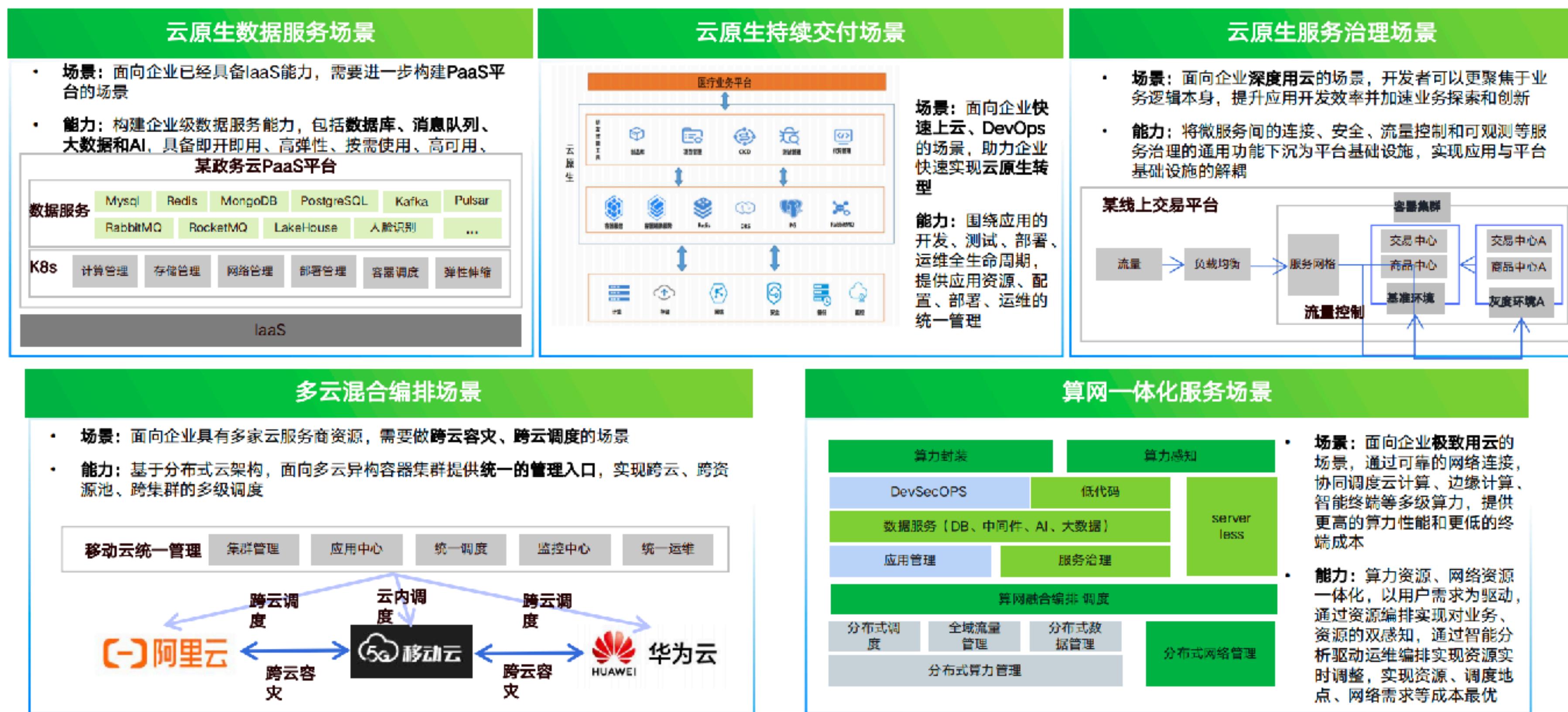
# 现代化应用开发

CNP提供运维工作台，基于指标、链路、日志、事件的全类型监控数据，结合强大的可视化和告警能力，提供一体化监控解决方案，满足**全链路、端到端**的统一监控诉求。



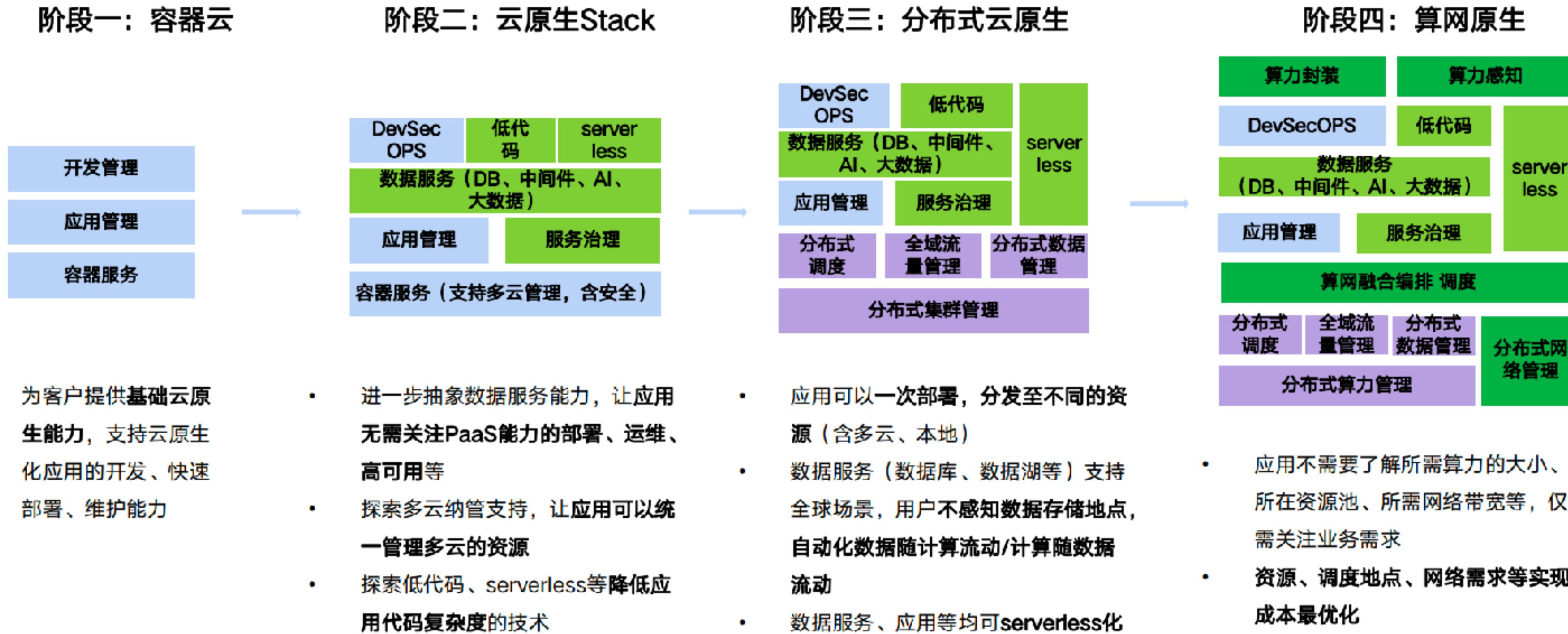
# 可观测

云原生有5大应用典型场景：云原生数据服务场景、云原生持续交付场景、云原生服务治理场景、多云混合编排场景、算网一体化服务场景。



# 应用场景

移动云云原生发展理念——面向云原生下一阶段：算网原生。



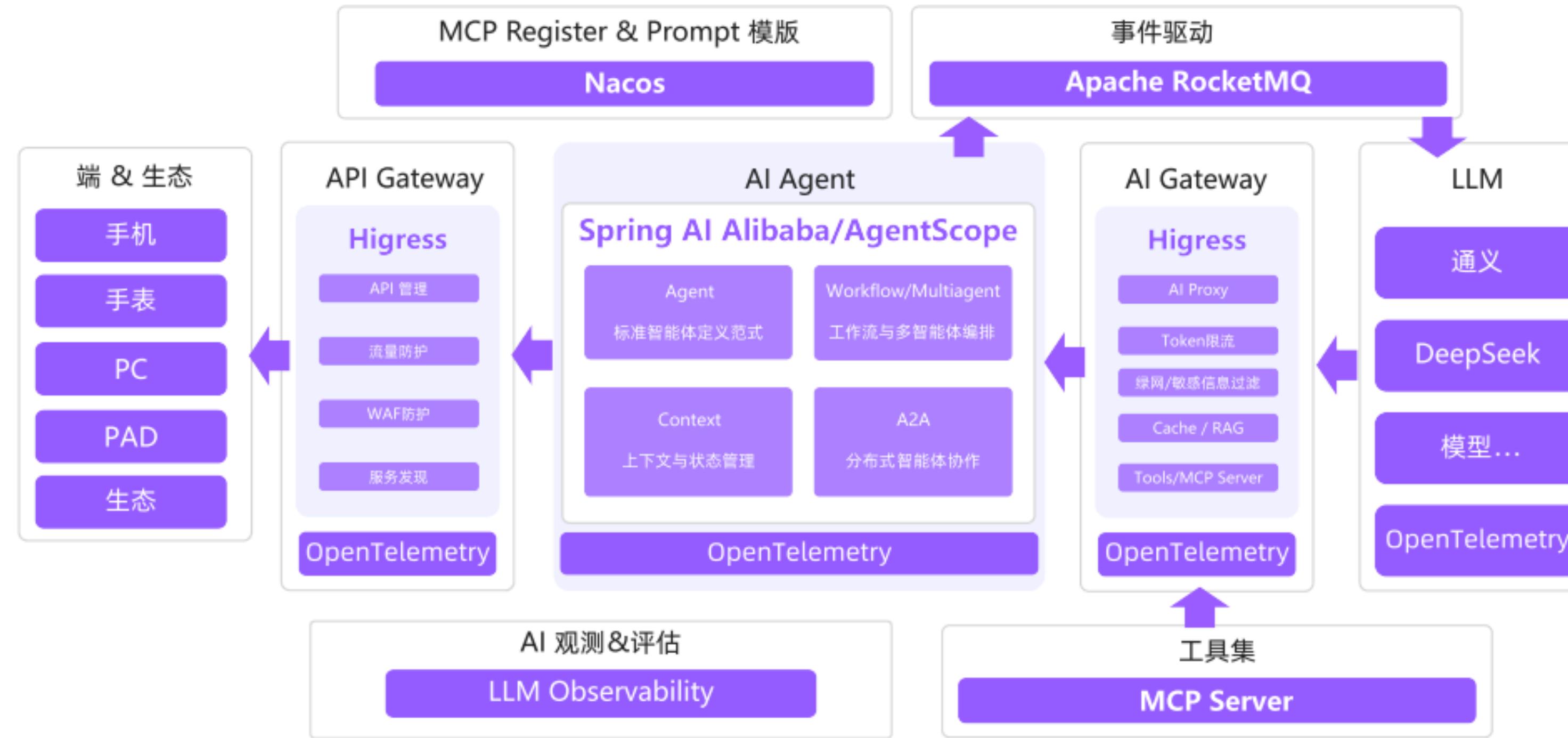
下一代：**算网原生**

级别	英文	中文	定义
1级	PoC Level	验证级	在特定业务场景中，通过基础大模型实现效率提升的初步探索。
2级	Pilot Deployment Level	试用级	AI 应用开始处理更复杂的任务，形成感知-决策-反馈的初步闭环能力。
3级	Operational Integration Level	应用级	AI 应用已深度融入现有业务系统并能够驱动核心业务流程，具备多模态感知和复杂推理能力。
4级	Enterprise Maturity Level	成熟级	高度自主化与自适应的 AI 原生应用，成为业务创新的核心引擎。

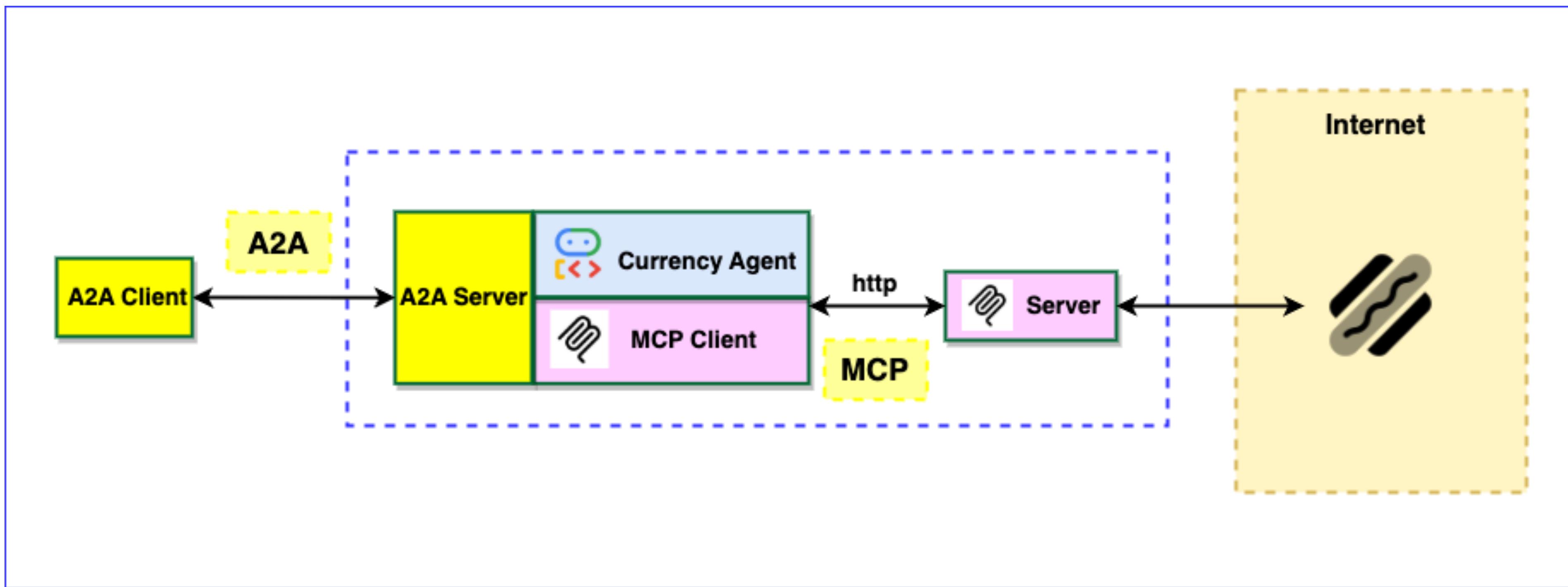
# AI原生应用架构成熟度分级描述

# AI 原生应用架构

基于模型, Agent 驱动, 以数据为中心, 整合工具链



# AI 原生应用架构



# 一个案例

# 框架与协议

## Model Context Protocol (MCP)

[Model Context Protocol \(MCP\)](#) 是一种开放协议，可标准化应用向 LLM 提供上下文的方式。MCP 提供了一种将 AI 模型连接到资源、提示和工具的标准化方式。

## 智能体开发套件 (ADK)

[智能体开发套件 \(ADK\)](#) 是一个灵活的编排框架，用于开发和部署 AI 智能体。ADK 不限模型、与部署无关，并且可与其他框架兼容。ADK 的设计旨在让智能体开发更像软件开发，从而让开发者能够更轻松地创建、部署和编排从简单任务到复杂工作流的智能体架构。

## Agent2Agent (A2A) 协议

[Agent2Agent \(A2A\) 协议](#) 是一种开放标准，旨在让 AI 智能体之间实现无缝通信和协作。正如 MCP 提供了一种标准化的方式来让 LLM 访问数据和工具一样，A2A 也提供了一种标准化的方式来让智能体与其他智能体对话！在一个代理由不同供应商使用各种框架构建的世界中，A2A 提供了一种通用语言，打破了孤岛并促进了互操作性。

# 演示项目

- <https://icyfenix.cn/exploration/projects/>
- 文档工程：
  - 凤凰架构：<https://icyfenix.cn>
  - Vepress 支持的文档工程：<https://github.com/fenixsoft/awesome-fenix>
- 前端工程：
  - Mock.js 支持的纯前端演示：<https://bookstore.icyfenix.cn>
  - Vue.js 2 实现前端工程：<https://github.com/fenixsoft/fenix-bookstore-frontend>
- 后端工程：
  - Spring Boot 实现单体架构：[https://github.com/fenixsoft/monolithic\\_arch\\_springboot](https://github.com/fenixsoft/monolithic_arch_springboot)
  - Kubernetes 为基础设施的微服务架构：[https://github.com/fenixsoft/microservice\\_arch\\_kubernetes](https://github.com/fenixsoft/microservice_arch_kubernetes)
  - Istio 为基础设施的服务网格架构：[https://github.com/fenixsoft/servicemesh\\_arch\\_istio](https://github.com/fenixsoft/servicemesh_arch_istio)

# 部署文档

- 单体
- config/index.js 增加如下配置即可
- proxyTable: {
  - '/restful': {
    - target: 'http://127.0.0.1:8080',
    - changeOrigin: true,
    - pathRewrite: {
      - '^/restful': '/restful'
  - }
- }
- },
- Kubernetes
  - <https://nqq5jc94uf.feishu.cn/docx/EffqdAcAnolcOrx7JFRcY7kBnng>
- Istio
  - <https://nqq5jc94uf.feishu.cn/docx/PtDwdURPBoRfMNxWYkXcCyTbn8e>