KubeCon CloudNativeCon

THE LINUX FOUNDATION

OPEN SOURCE SUMMIT

AI_dev

Open Source GenAI & ML Summit

China 2024

基于函数化弹性调度和RDMA技术加速Serverless AI大模型推理

王成龙 && 李一鸣

KubeCon

CloudNativeCon

THE LINUX FOUNDATION

OPEN SOURCE SUMMIT

AI_dev

Open Source GenAI & ML Summit

China 2024

- KServe 介绍
- 浪潮云海基于 KServe 的最佳实践
- 基于函数化弹性调度技术加速 Serverless AI 大模型推理

Kubernetes + AI 趋势

KubeCon CloudNativeCon China 2024 | THE LINUX FOUNDATION OPEN SOURCE SUMMIT | AI_dev

人工智能服务上云形成趋势，容器和 Serverless 技术将成为未来 AI 应用开发的主要平台

Kubernetes 已成为云原生事实标准，根据云原生计算基金会（CNCF）调查报告，在 2023 年，全球 84% 用户在生产中使用或者计划使用 Kubernetes

KUBERNETES SOLIDIFIES ITS CORE TECHNOLOGY STATUS

In the 2021 CNCF Annual Survey, we stated that Kubernetes had crossed the adoption chasm to become a mainstream global technology. Today, the laggards are finally catching up.

This year's analysis of cloud adoption, containers, and Kubernetes did not include organizations whose primary revenue stream was derived from offering cloud native products and services – mostly vendors. By focusing our research on organizations that are not in the cloud business, but had a potential or actual reason to consume cloud services, we sought to get a more accurate view into the adoption, benefits, and challenges of consuming cloud products and services. We expected that adoption rates would be less than the 2022 metrics because they included both consumers and providers of cloud services – so we did not do any direct comparisons between 2023 and prior years. However, in 2022, 58% percent of providers and consumers (the entire sample) were using Kubernetes in production and 23% (81% total) were actively evaluating it. In 2023, 66% of potential/actual consumers were using Kubernetes in production and 18% were evaluating it (84% total).

USING OR EVALUATING KUBERNETES

84%

Up from 81% in 2022

CNCF Annual Survey 2023

Running AI On Cloud Native Infrastructure

The value of Cloud Native for AI is highlighted by articles in the media published by cloud service providers and/or AI companies.[19] [20] The emergence of AI-related offerings by cloud providers and emerging start-ups in this space are crucial indicators of how Cloud Native principles can shape the systems necessary for the AI evolution.

OPENAI

Scaling Kubernetes to 7,500 nodes

HUGGING FACE

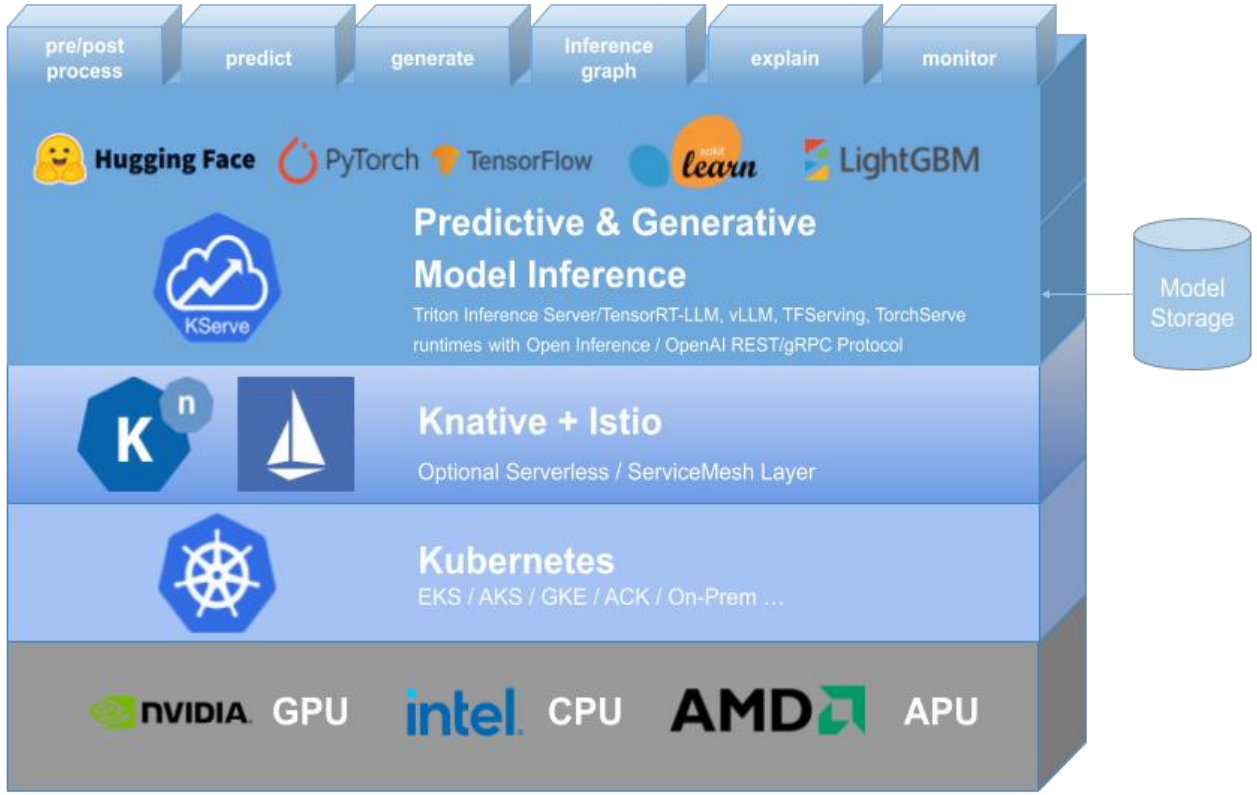
Hugging Face Collaborates with Microsoft to launch Hugging Face Model Catalog on Azure

Cloud Native Artificial Intelligence Whitepaper

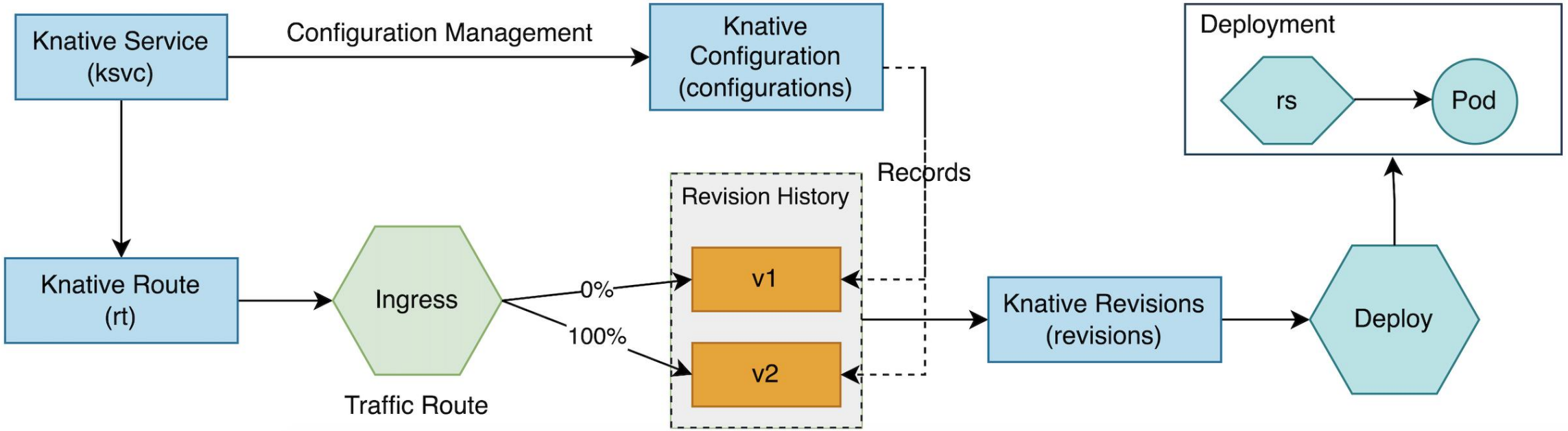
KServe 简介

KServe 是 Kubernetes 上的模型推理平台，用于简化和加速机器学习模型的部署、服务和管理，支持多种机器学习框架、具备弹性扩容能力

- 跨 ML 框架提供高性能、标准化的推理协议
- 提供简单易用且可扩展的生产级 ML 服务，支持预测、前/后处理、监控和解释功能
- 使用 ModelMesh 提供高可扩展性、高密度部署和智能路由和智能路由功能
- 依托于 Knative，支持 GPU 自动缩放、按需扩缩至零
- 提供金丝雀发布、实验、集成和转换器等高级功能



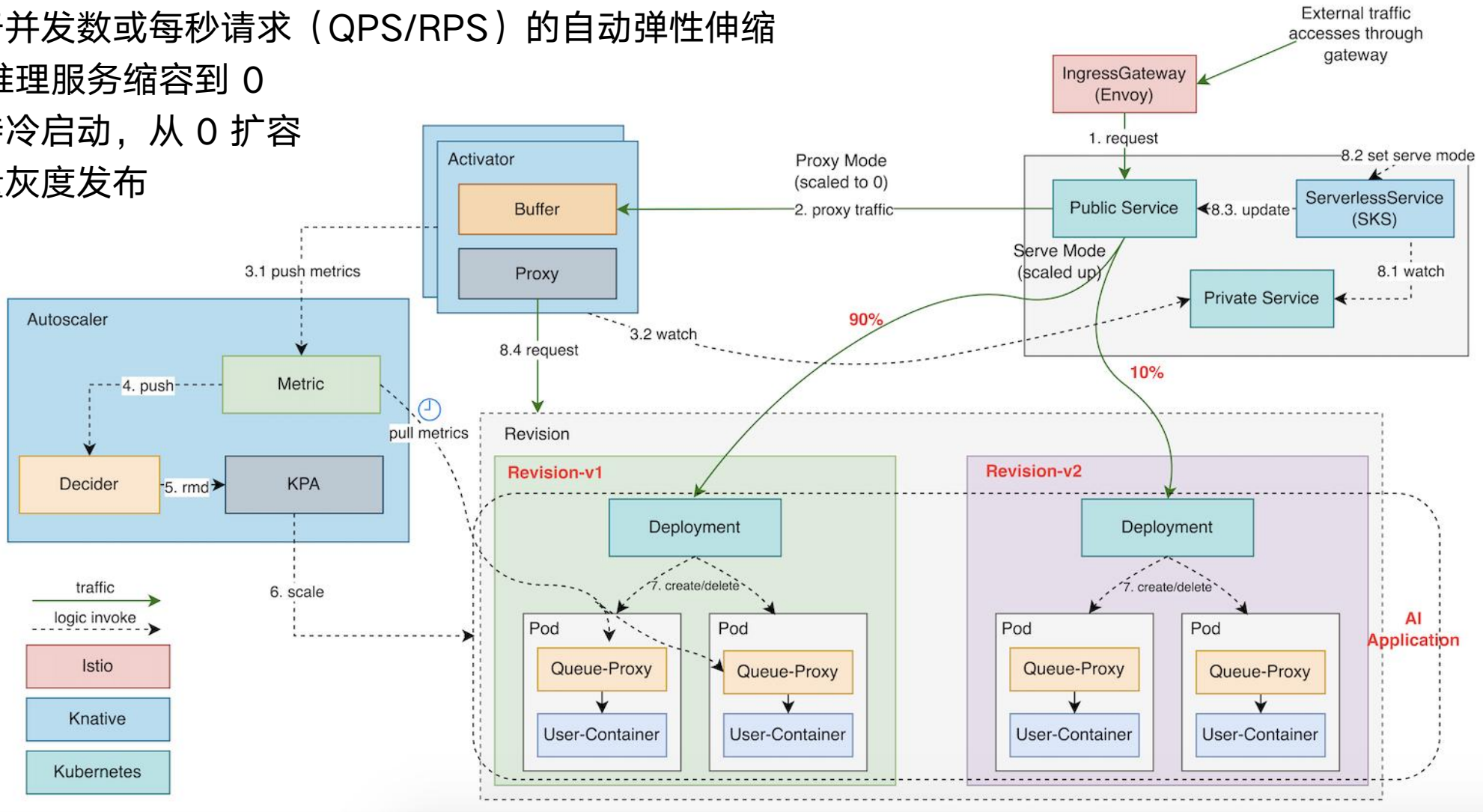
KServe —— Serverless Layer



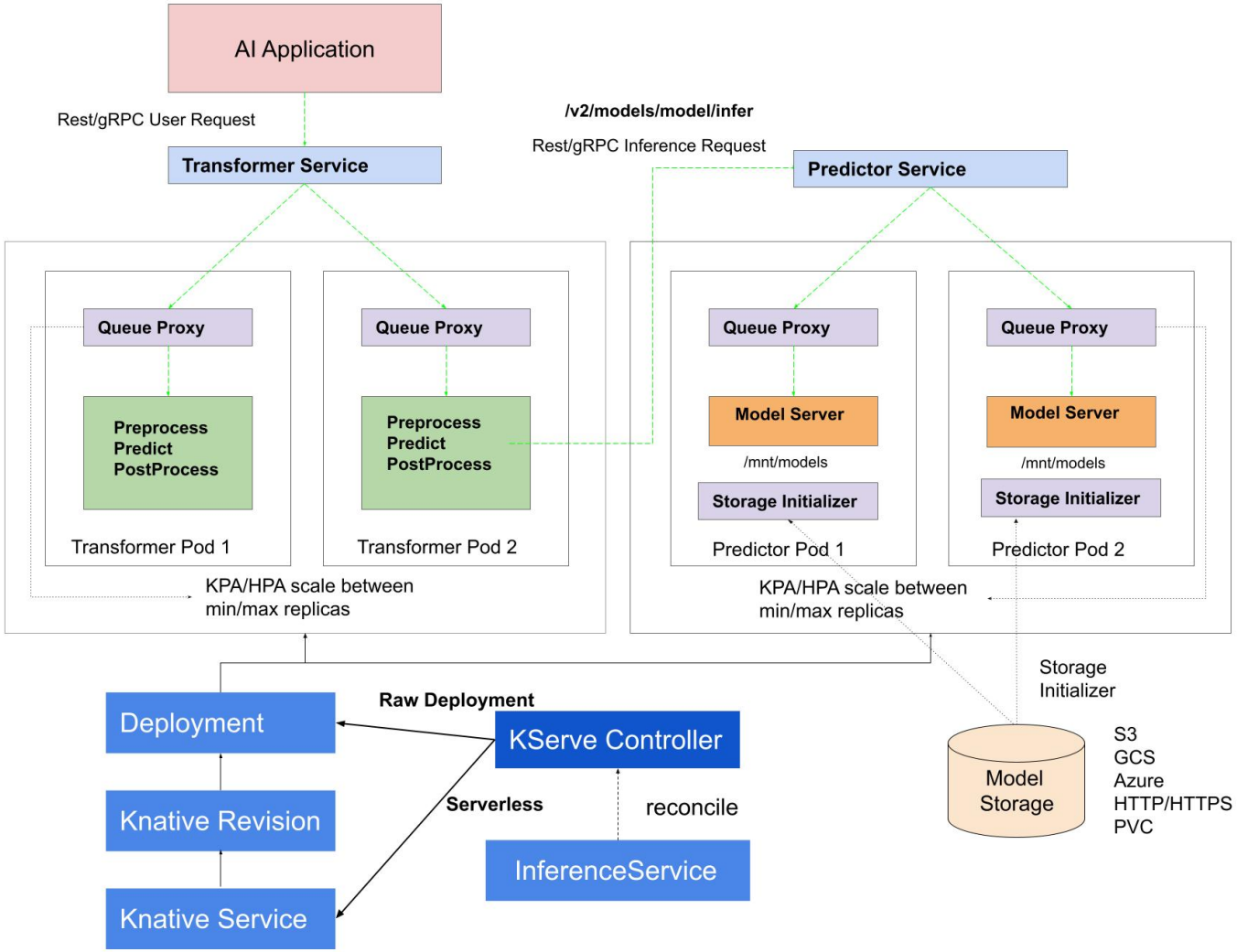
- **Service**：Serverless 应用的抽象，用于定义和管理 Serverless 应用的完整生命周期
- **Configuation**：维护了部署应用的期望状态，每次修改配置会创建一个新的修订版
- **Revision**：在每次变更工作负载时生成的代码和配置的时间点快照。对 Service 或 Configuration 的更改都会生成一个新的 Revision
- **Route**：用于定义流量的路由规则，将网络请求路由到特定的 Revision

KServe —— Serverless Layer

- 基于并发数或每秒请求（QPS/RPS）的自动弹性伸缩
- AI 推理服务缩容到 0
- 支持冷启动，从 0 扩容
- 流量灰度发布



KServe —— Control Plane



1 Create an InferenceService

```
kubectl apply -n kserve-test -f - <<EOF
apiVersion: "serving.kserve.io/v1beta1"
kind: "InferenceService"
metadata:
  name: "sklearn-iris"
spec:
  predictor:
    model:
      modelFormat:
        name: sklearn
      storageUri: "gs://kfserving-examples/models/sklearn/1.0/model"
EOF
```

2 Determine the ingress address

```
NAME           URL
sklearn-iris   http://sklearn-iris.kserve-test.example.com
```

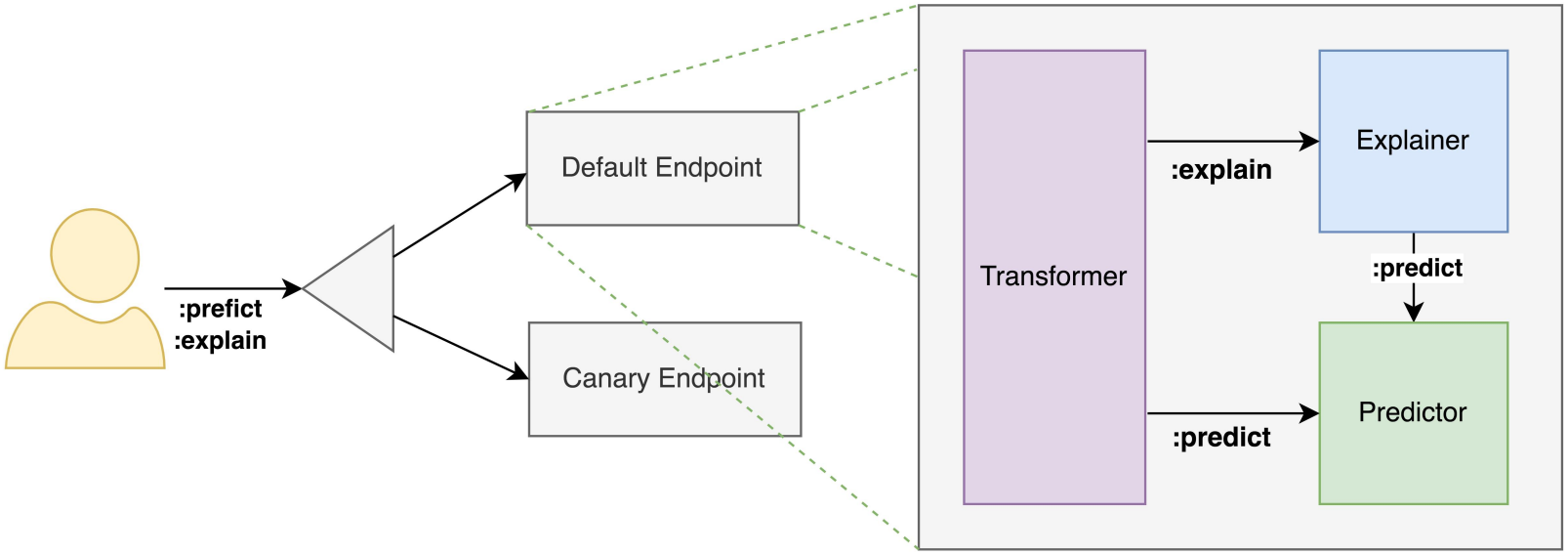
3 Perform inference

```
curl -v http://sklearn-iris.kserve-test.example.com/v1/models/
sklearn-iris:predict -d @./iris-input.json
```

KServe —— Data Plane

每个 InferenceService 由多个组件组成：“预测器”、“解释器”和“转换器”

- **预测器**：系统的核心组件，负责实际的模型推理
- **解释器**：提供模型解释功能，有助于调试和验证模型的行为，特别是在高风险或需要高透明度的应用场景中
- **转换器**：用于对输入数据进行预处理，或对输出数据进行后处理。可以执行数据清洗、特征提取、格式转换等操作，以确保输入数据符合模型要求，或将输出结果转换为用户期望的格式



KServe —— Data Plane Protocol

KServe 提供了对多种协议的支持，包括 REST 和 gRPC，模型部署和服务调用变得更加标准化和统一化，方便跨系统的集成与调用

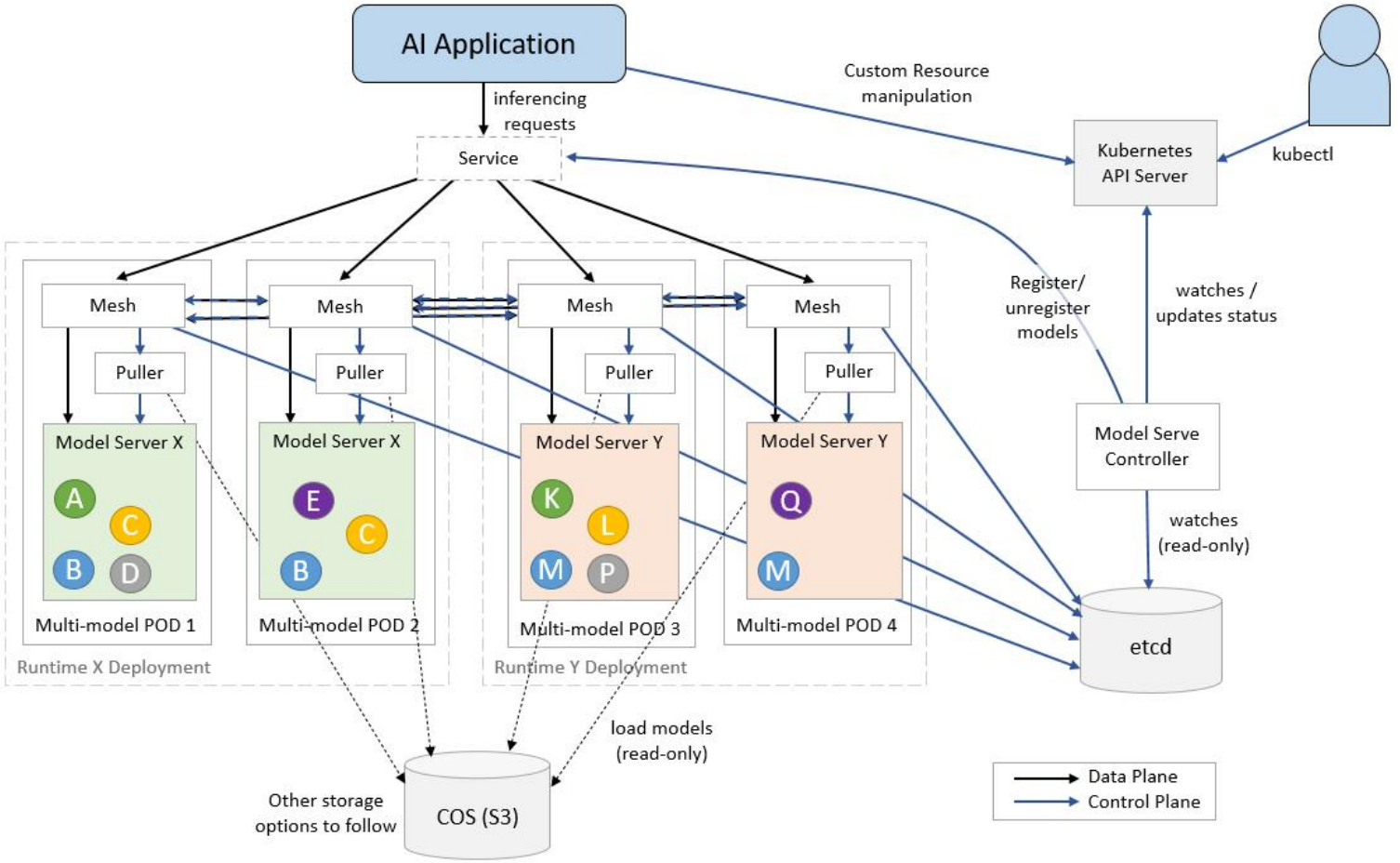
REST（v1 Inference Protocol）
GET /v1/models
GET /v1/models/{model_name}
POST /v1/models/{model_name}:predict
POST /v1/models/{model_name}:explain

REST（v2 Inference Protocol）	gRPC（v2 Inference Protocol）
GET v2/health/live	rpc ServerLive(ServerLiveRequest) returns (ServerLiveResponse)
GET v2/health/ready	rpc ServerReady(ServerReadyRequest) returns (ServerReadyResponse)
GET v2/models/{model_name}	rpc ModelMetadata(ModelMetadataRequest) returns (ModelMetadataResponse)
GET v2/models/{model_name}/ready	rpc ModelReady(ModelReadyRequest) returns (ModelReadyResponse)
GET v2	rpc ServerMetadata(ServerMetadataRequest) returns (ServerMetadataResponse)
POST v2/models/{model_name}/infer	rpc ModelInfer(ModelInferRequest) returns (ModelInferResponse)

KServe —— ModelMesh

ModelMesh 是一种用于大规模机器学习模型部署和推理的高级架构组件。解决大规模、多模型部署环境中的一些关键挑战：

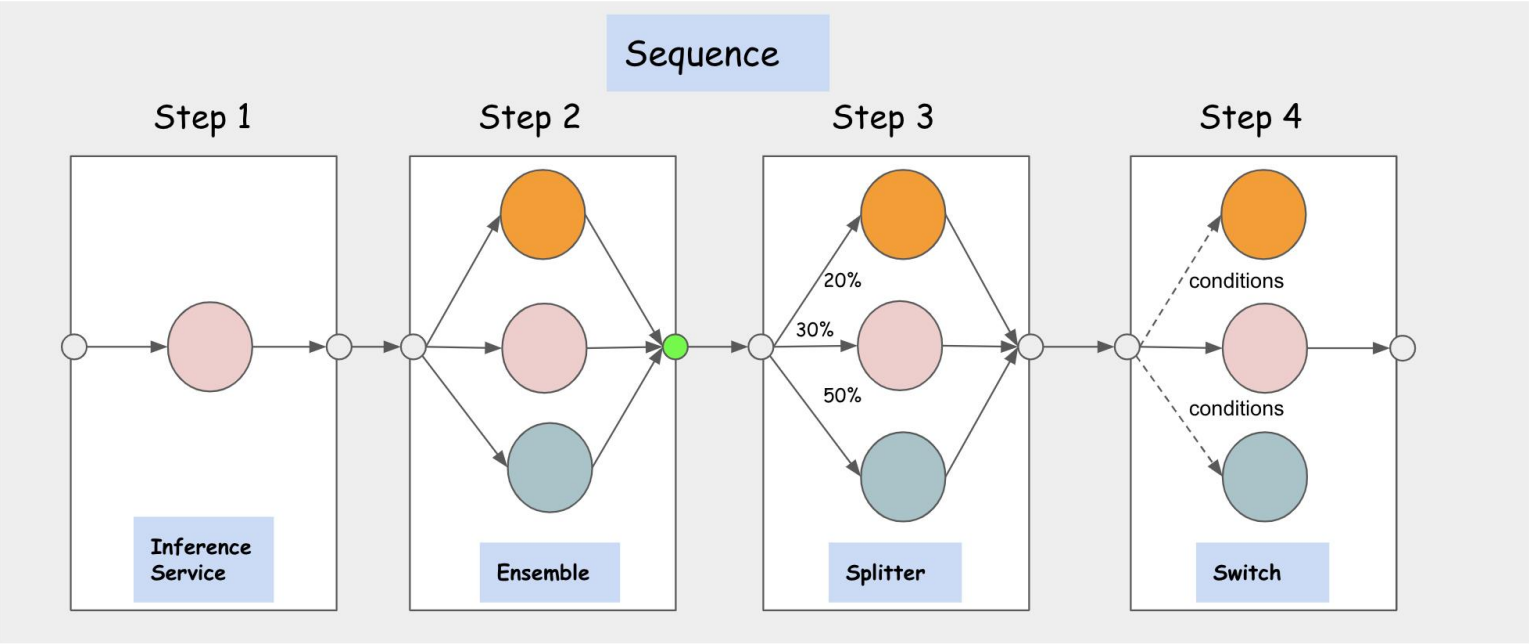
- **计算资源限制**（每个 Pod 中注入了 Sidecar，每个 InferenceService 额外增加 0.5核 CPU 和 0.5G 内存开销）
- **最大 Pod 限制**（Kubernetes 建议每个节点最多 110 个 Pod，假设每个 InferenceService 平均有 4 个 Pod，50 节点集群最多可以运行 1000 个模型）
- **最大IP地址限制**（4096 个 IP 地址的集群最多可以部署 1024 个模型）



KServe —— Inference Graph

推理图（Inference Graph）允许用户将多个机器学习模型和处理步骤链接在一起，形成一个图形化的推理工作流。简化了复杂机器学习应用的部署和管理。支持四种不同类型的路由方式：Sequence、Ensemble、Splitter、Switch

- 序列（Sequence）：按照定义的顺序依次执行多个步骤
- 集成（Ensemble）：并行执行多个步骤，并将结果组合起来
- 分离器（Splitter）：将输入数据拆分成多个部分，并分别路由到不同的服务
- 切换（Switch）：根据condition条件将流量路由到不同的服务

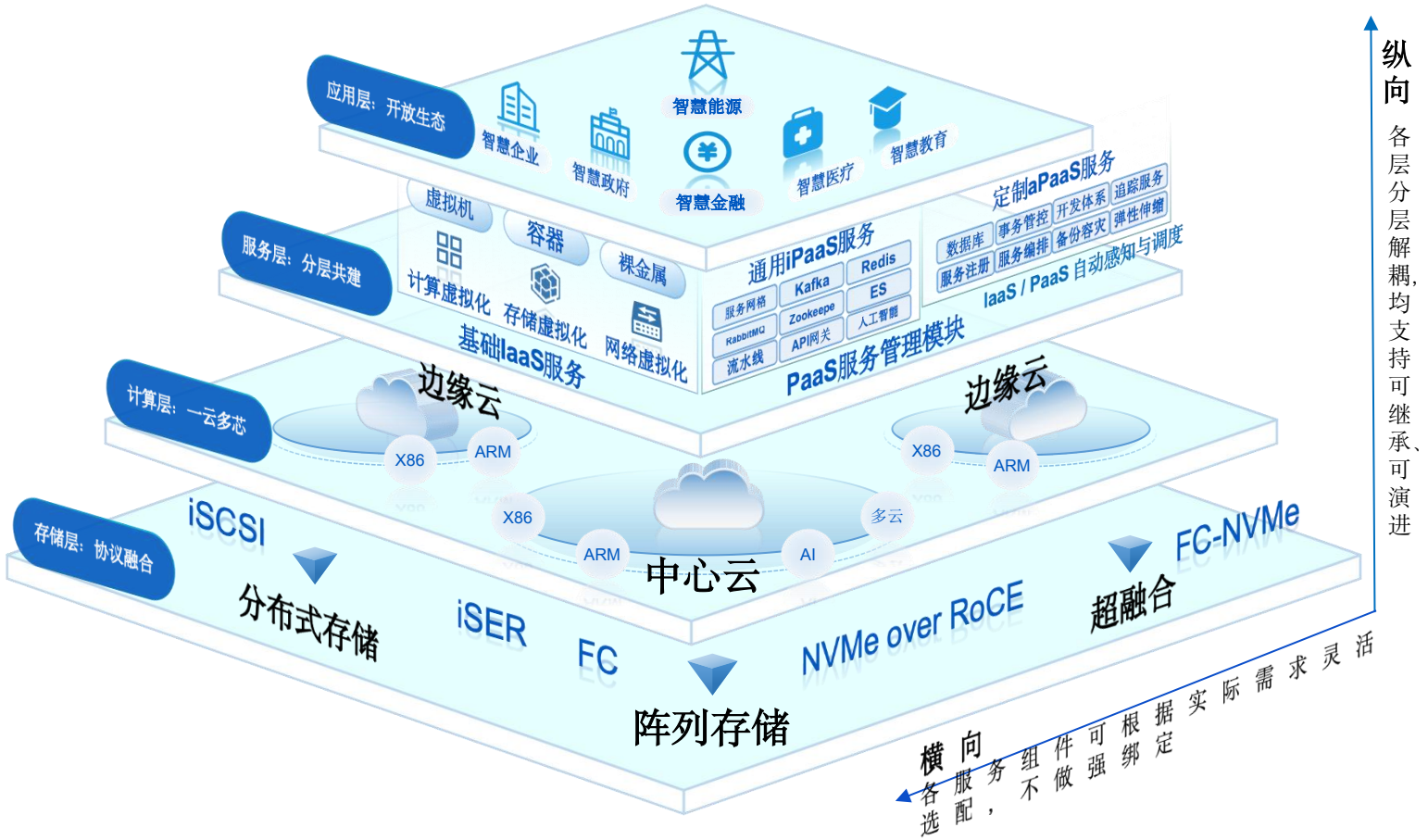



```yaml
apiVersion: "serving.kserve.io/v1alpha1"
kind: "InferenceGraph"
metadata:
  name: "inference-graph"
spec:
  nodes:
    ...
    root:
      routerType: Sequence
      steps:
      - serviceName: isvc1
      - serviceName: isvc2
    ....
    root:
      routerType: Ensemble
      routes:
      - serviceName: sklearn-iris
        name: sklearn-iris
      - serviceName: xgboost-iris
        name: xgboost-iris
    ....
    root:
      routerType: Splitter
      routes:
      - serviceName: sklearn-iris
        weight: 20
      - serviceName: xgboost-iris
        weight: 80
    ...
    root:
    routerType: Switch
    steps:
    - serviceUrl: http://single-1.default.{$domain}/switch
      condition: "[@this].#(source==client)"
    - serviceUrl: http://single-2.default.{$domain}/switch
      condition: "instances.#(intval>10)"
```


KubeCon
CloudNativeCon
THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
AI_dev
Open Source GenAI & ML Summit
China 2024
浪潮云海基于 KServe 的最佳实践

产品简介

浪潮云海云操作系统 InCloud OS 是浪潮面向私有云领域，以可继承、可演进为理念自研的一套开放、稳定、高效、安全的云平台软件，提供云主机、容器、数据库、中间件、云安全等服务和智能运维、灵活运营等能力，助力政企数字化转型

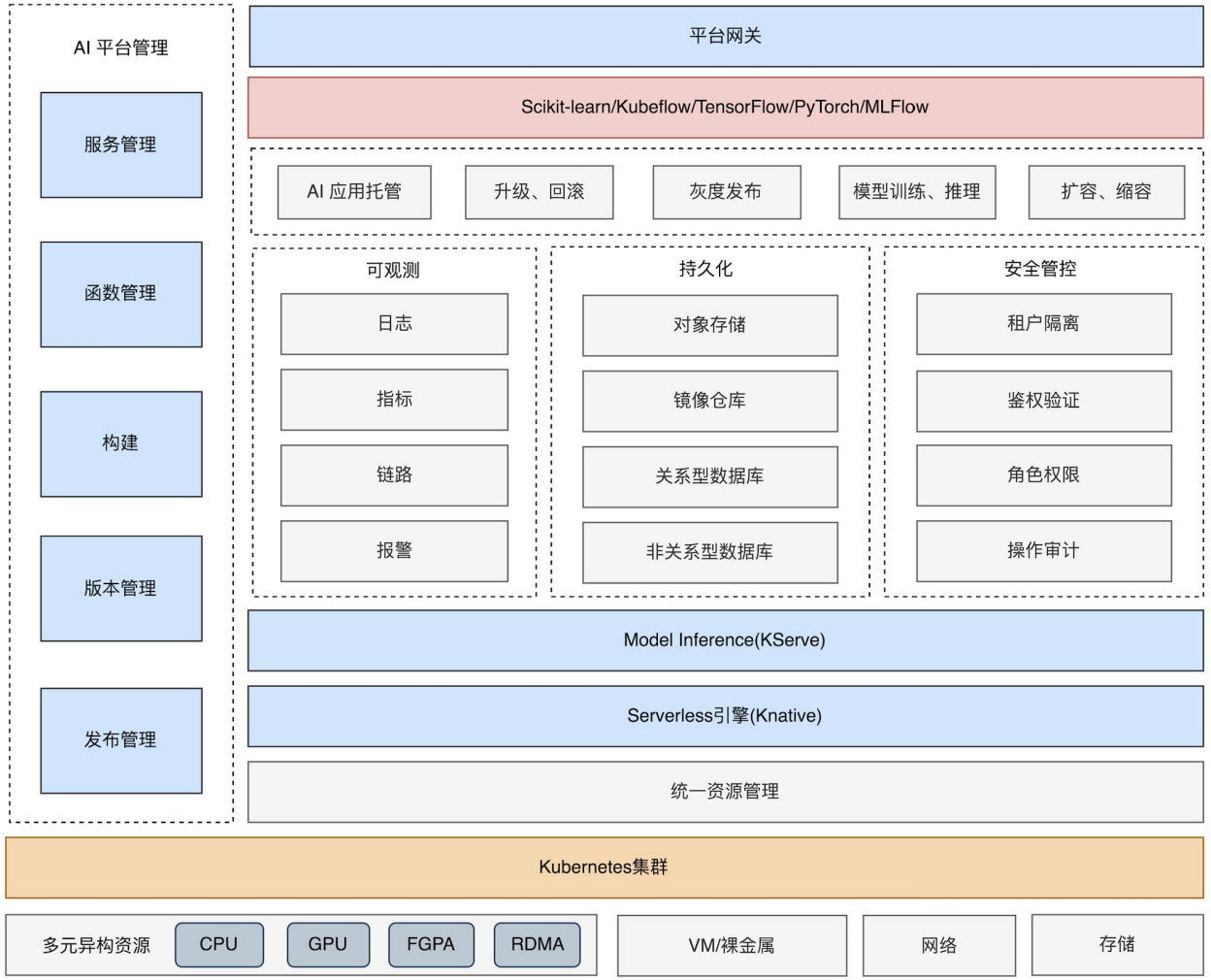


浪潮云海基于 KServe 的实践方案

浪潮云海 AI 平台技术架构

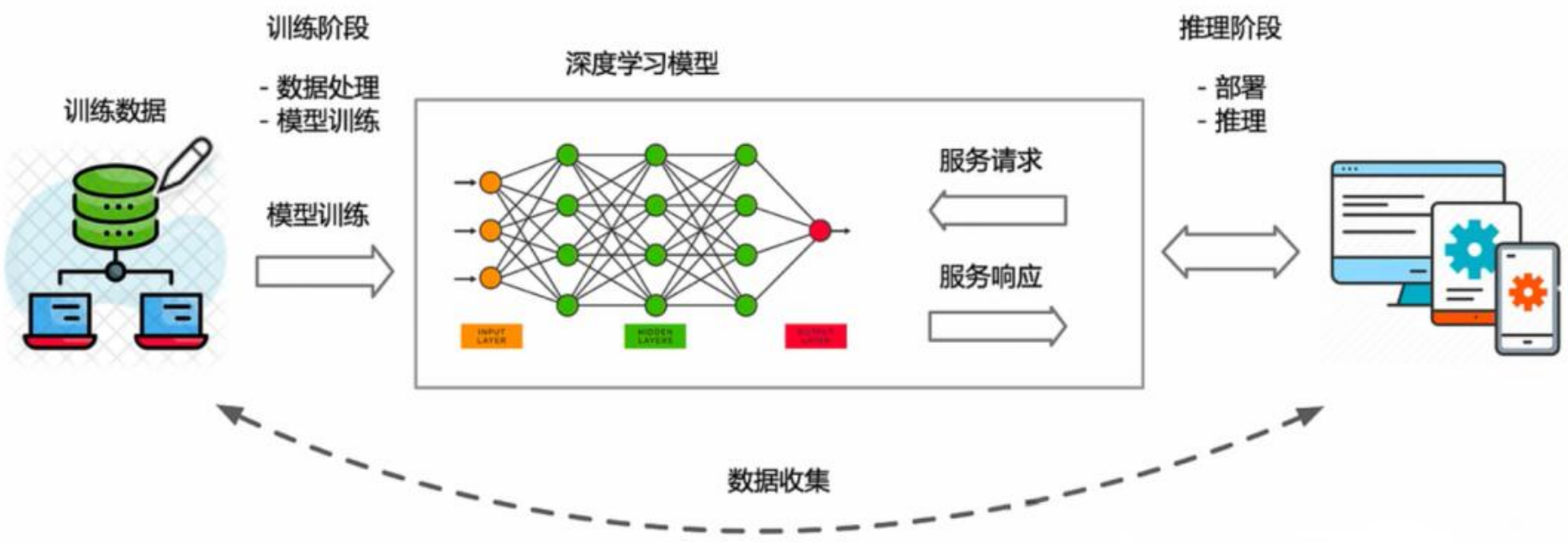
- 平台以容器为底座，构建于 Kubernetes 集群之上，以 Knative 为 Serverless 核心引擎，以 KServe 为模型推理框架，支撑多种 AI 业务场景

- 平台支持对 AI 推理任务的生命周期进行统一管理，充分利用云的资源弹性、异构算力、标准化服务等云原生技术特性，为 AI 提供低成本、可扩展的端到端解决方案



AI 模型推理服务上云

AI 服务的生产流程涵盖了从数据准备、模型训练与微调，到模型推理服务上线的全周期管理，形成一个自我增强的闭环。在推理阶段生成的结果以及使用过程中收集的新数据，会回流至数据处理环节，持续推动模型的优化与迭代



浪潮云海基于 KServe 的最佳实践

AI 模型推理服务上云

① 使用 Docker 将模型打包成容器

② 创建持久卷声明 (PVC)

③ 创建一个 Pod 来访问 PVC

④ 将模型存储在持久卷上

⑤ 自定义运行时：基于 KServe API 实现自定义 REST ServingRuntime 模型

⑥ 部署新的推理服务

⑦ 运行推理请求

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: "llm-pvc"
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 4Gi
  storageClassName: inspur-storage
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: "pvc-access"
spec:
  containers:
    - name: main
      image: ubuntu
      command: ["/bin/sh", "-ec", "sleep 10000"]
      volumeMounts:
        - name: "pvc"
          mountPath: "/mnt/models"
  volumes:
    - name: "pvc"
      persistentVolumeClaim:
        claimName: "llm-pvc"
```

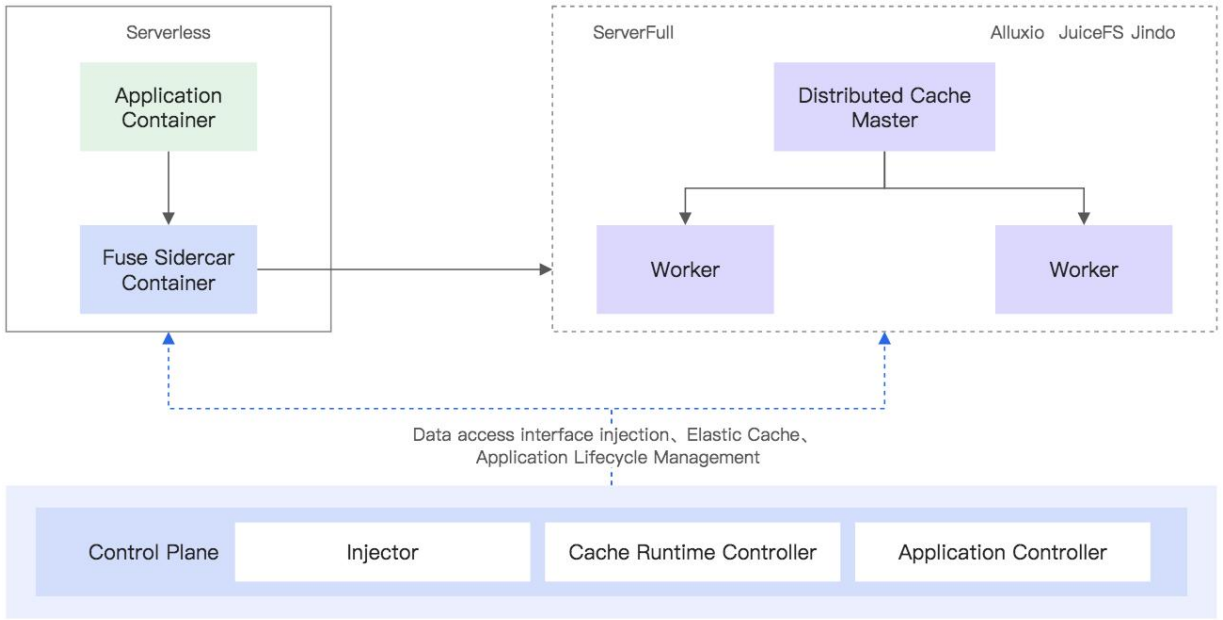
```yaml
apiVersion: serving.kserve.io/v1alpha1
kind: ServingRuntime
metadata:
  annotations:
    inspur.com/template-display-name: ServingRuntime for llm
  name: llm
spec:
  containers:
  - args:
    - --model=/mnt/models/
    - --served-model-name=llm
    - --tensor-parallel-size=2
    - --max-model-len=2048
    image: 10.49.38.104:5000/llm:v1
    name: kserve-container
    ports:
    - containerPort: 8080
      name: http1
      protocol: TCP
  supportedModelFormats:
  - autoSelect: true
    name: pytorch
```

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: llm
spec:
  predictor:
    minReplicas: 3
    model:
      modelFormat:
        name: pytorch
      runtime: llm
      storageUri: pvc://llm-pvc/mnist.pt
      resources:
        limits:
          cpu: "4"
          memory: 20Gi
          nvidia.com/gpu: 2
```

挑战一：模型镜像大，拉取时间长，影响应用启动速度

- 引入 Fluid（开源的 K8s 原生的分布式数据集编排和加速引擎）， 与 KServe 相结合，通过数据预热到分布式缓存加速模型加载流程，提升冷启动速度

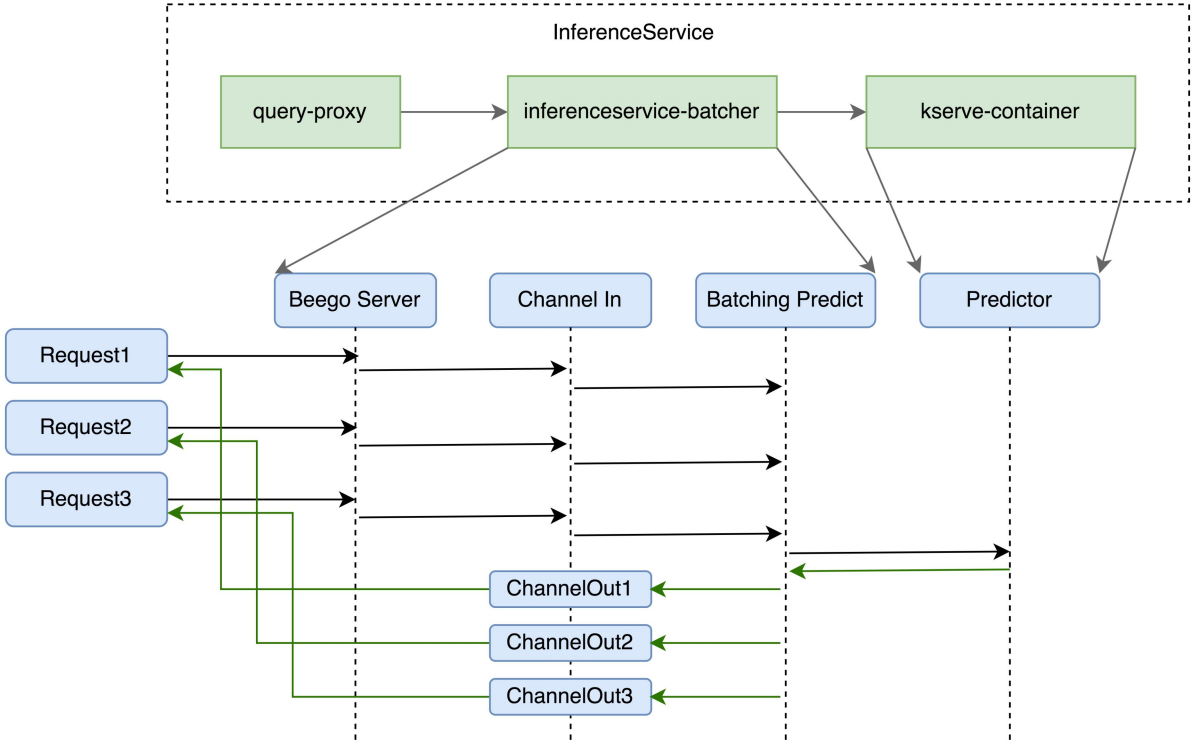
- 通过数据复用，多个任务能够共享数据缓存，避免重复拉取同一份数据，减少网络消耗



挑战二：高并发场景下，推理存在延迟，影响应用的响应时间

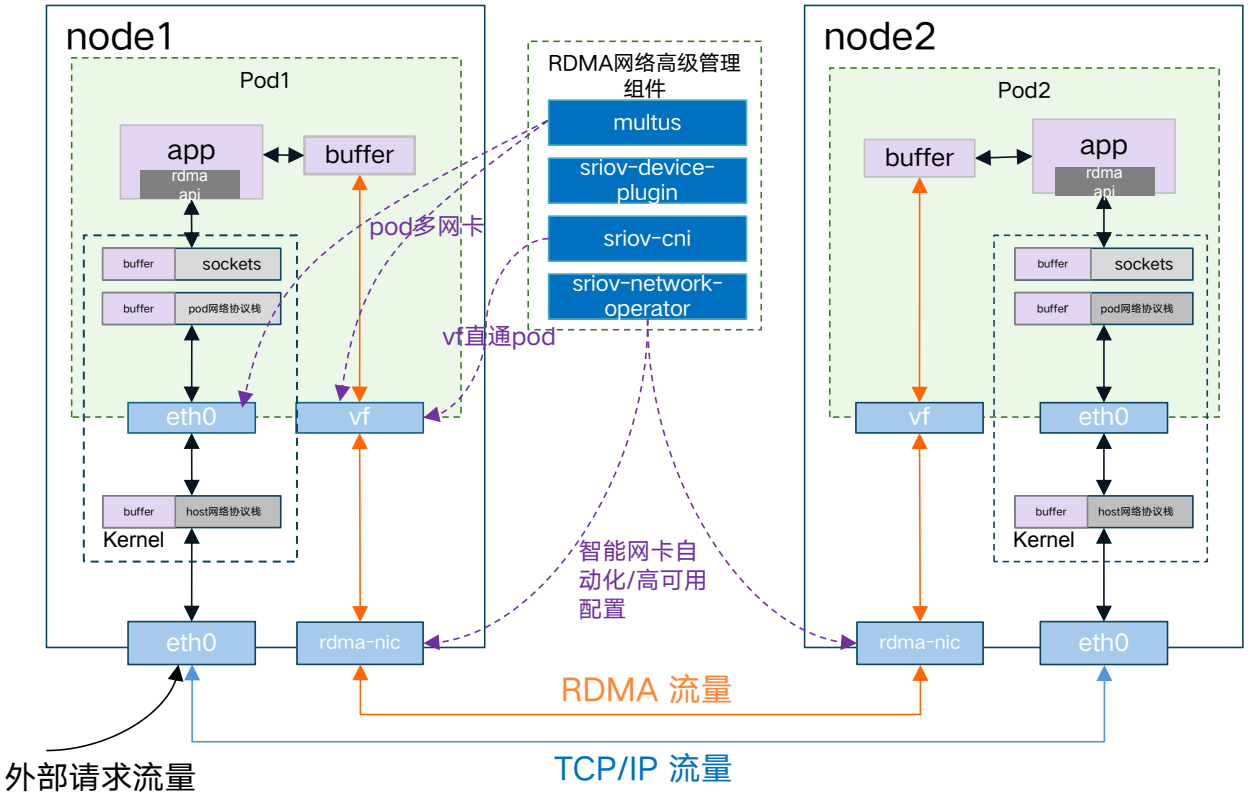
- 自适应批处理：将多个推理请求组合成单个批处理，从而优化吞吐量和延迟

- 自适应弹性伸缩：模型推理服务Serverless部署，基于请求数快速弹性伸缩，加快处理速度



挑战三：模型推理过程中传输的 KV 缓存数据高达 GB 级别，通信开销高

- 基于 SR-IOV 和容器多网卡技术，为容器提供 RDMA 和标准 K8s 网络的双重能力
- 通过 RDMA 高性能通信技术，加速模型推理中的高速 KV 缓存迁移，避免传统网络协议栈的高开销

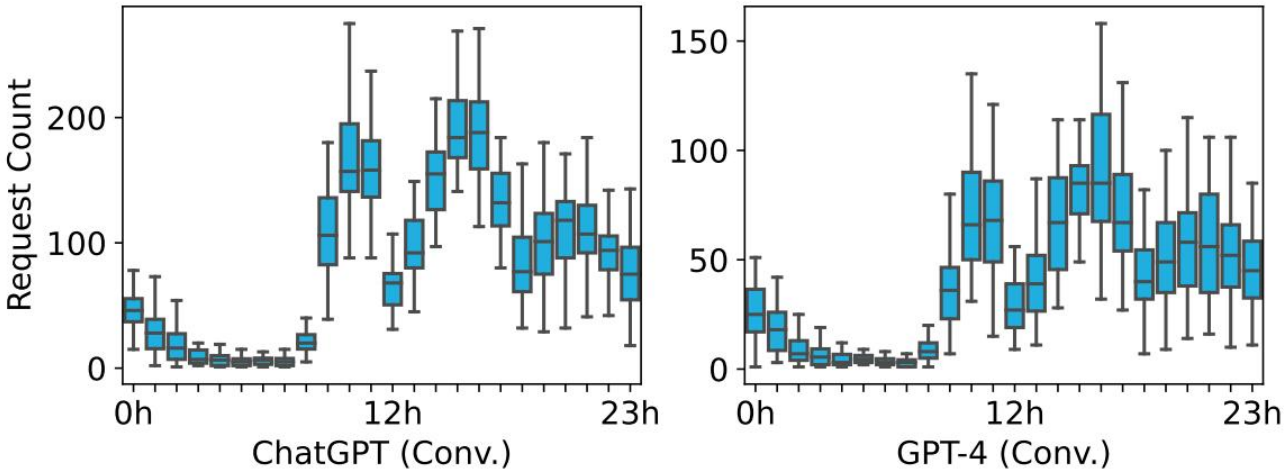


挑战四：K8s 的集中式控制平面无法实现对大规模突发请求的秒级响应

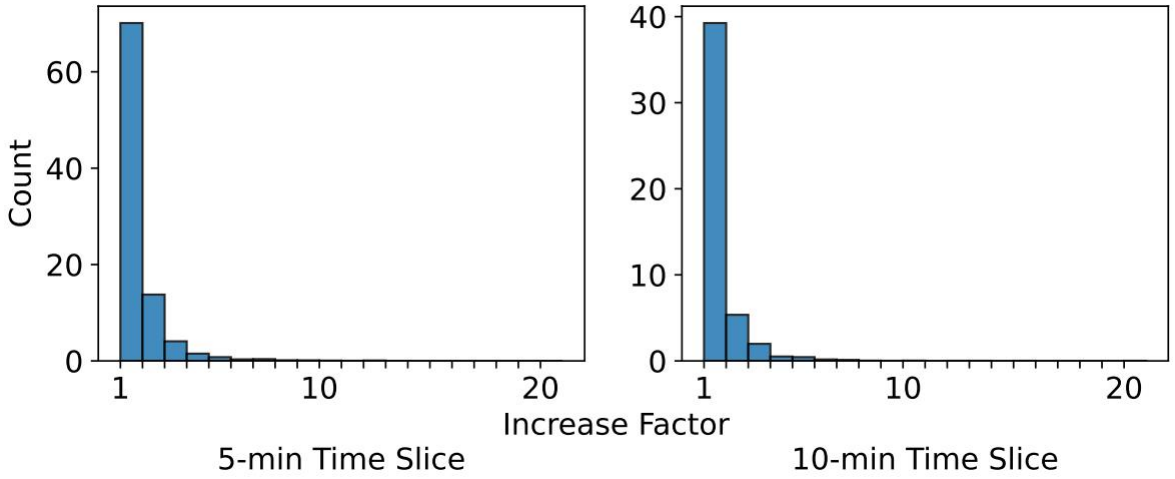

KubeCon
CloudNativeCon
THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
AI_dev
Open Source GenAI & ML Summit
China 2024
基于函数化弹性调度技术加速
Serverless AI 大模型推理

大模型真实负载

- **高动态负载**
  - 高波动：峰谷比 > **200倍**
  - 高突发：请求量**翻倍**在**5分钟**内发生**65次**

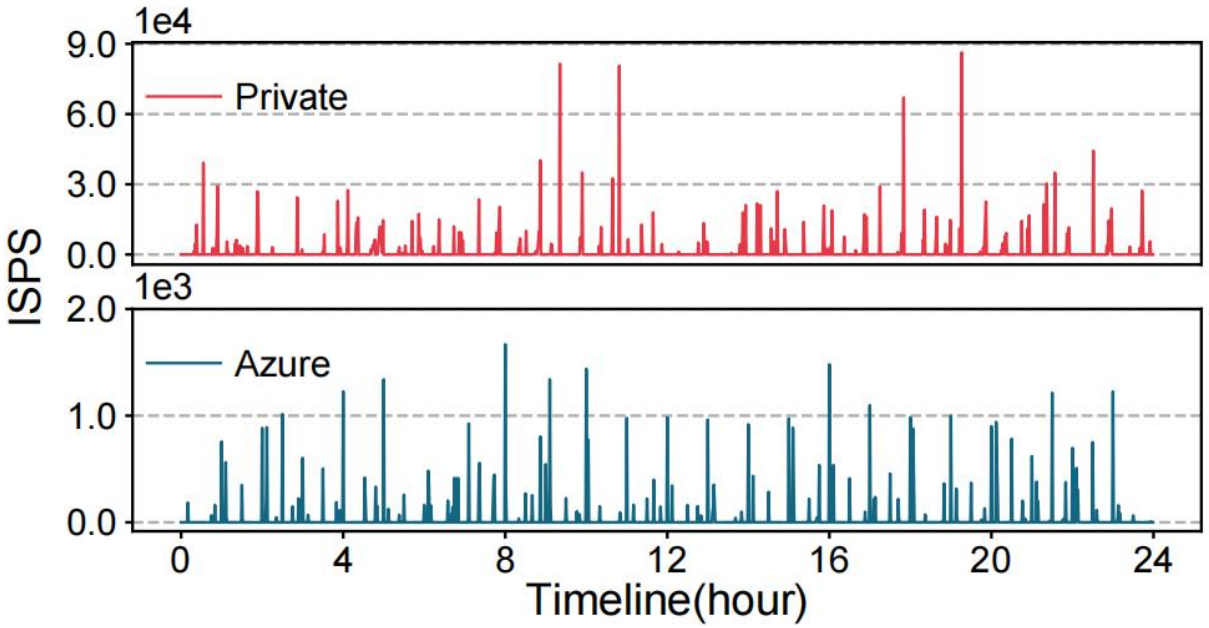


日常周期性对话服务的请求量



短期内负载突发的频率

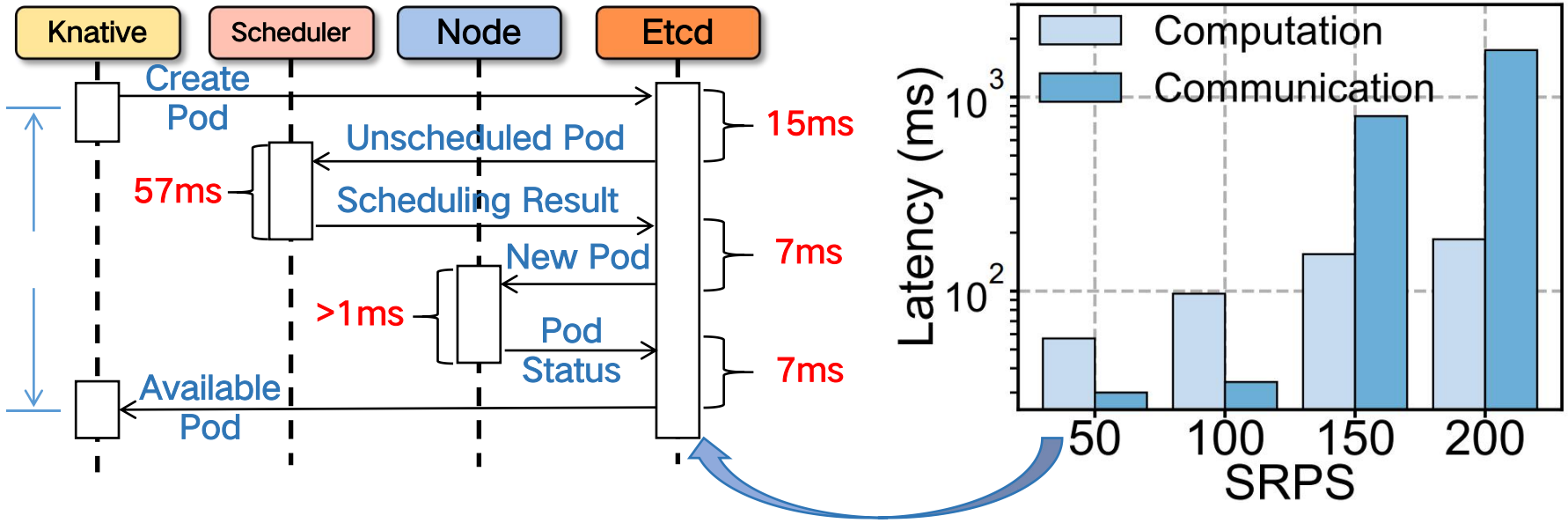
- **Serverless 中扩容的稀疏性**
  - Azure函数服务中超过 **90.1%** 的时间没有扩容事件
  - 某私有函数服务中超过 **86%** 的时间没有扩容事件



一天内每秒的扩容请求量

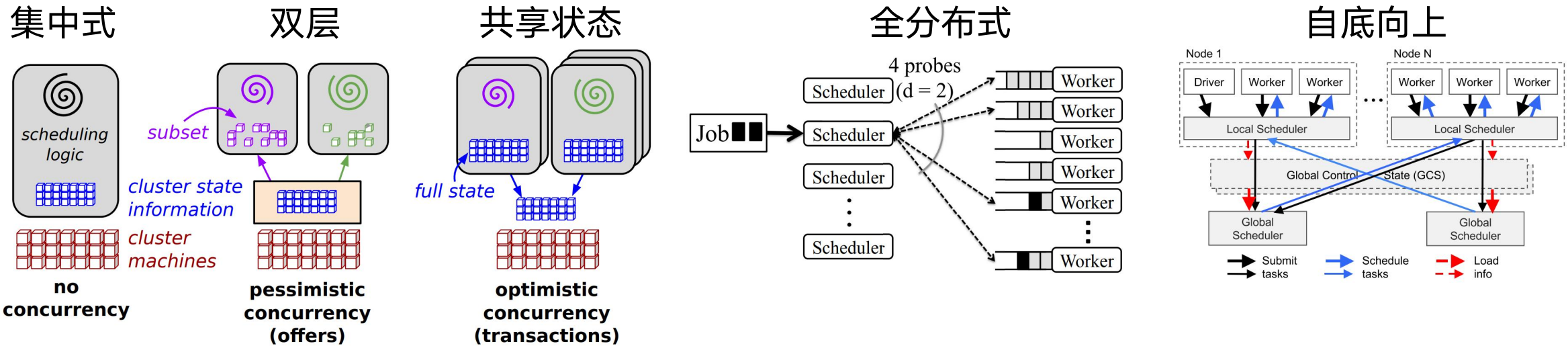
控制平面

- **集中式架构**
  - 无法满足高并发扩容请求
  - 存在吞吐上限，调度延迟高



现有工作

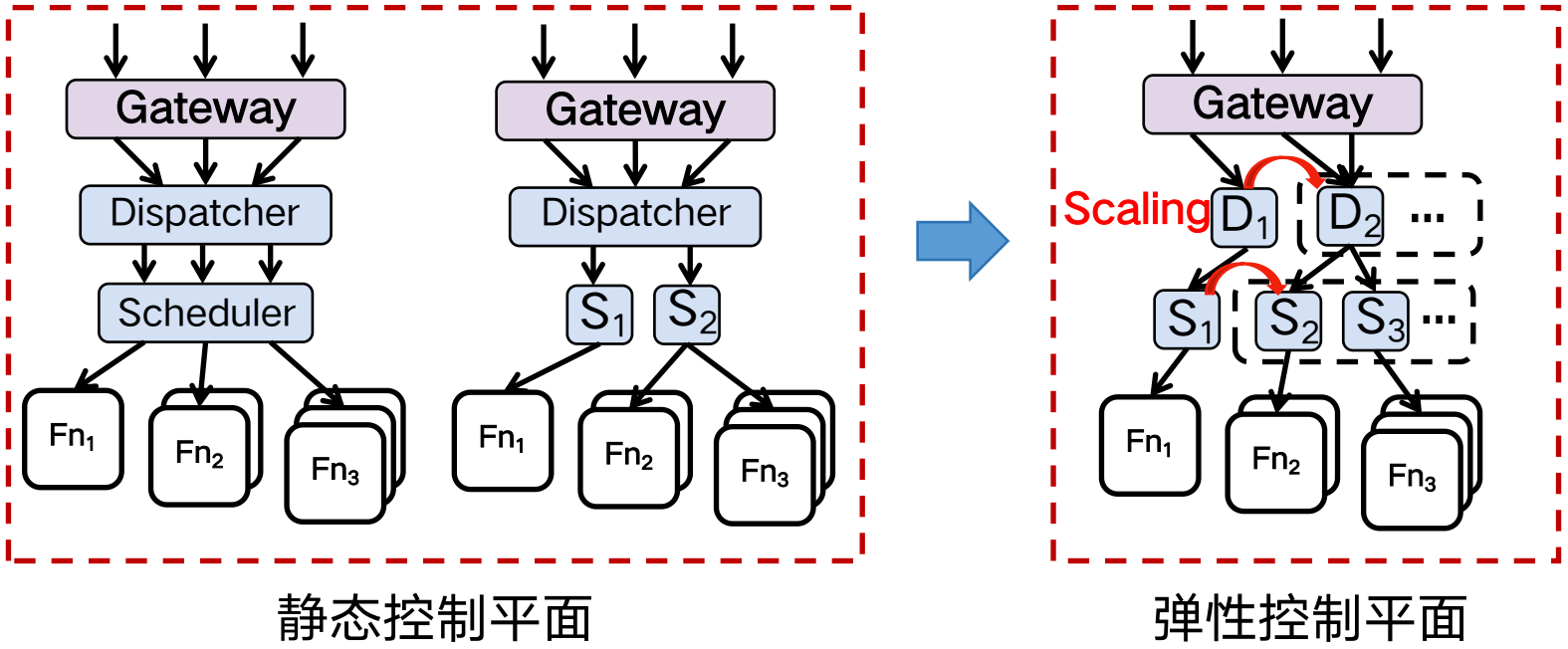
- **分布式调度器**
  - 双层架构：Mesos (NSDI 11), Yarn (SoCC 13)
  - 共享状态：Omega (EuroSys 13)
  - 全分布式：Sparrow (SOSP 13)
  - 自底向上：Ray (OSDI 18)



现有工作

- **不足之处**
  - 只解决了调度器的瓶颈
    - 扩容决策和服务发现可能成为**新瓶颈**
  - 集群资源紧张时调度能力大幅下降
    - **调度冲突和失败**的处理低效
  - 资源浪费或吞吐瓶颈
    - **静态控制平面**不适用于波动负载

- **函数化控制平面**
  - 将控制平面解耦为可拓展的控制函数，根据请求负载动态地调整每个控制函数的实例数量



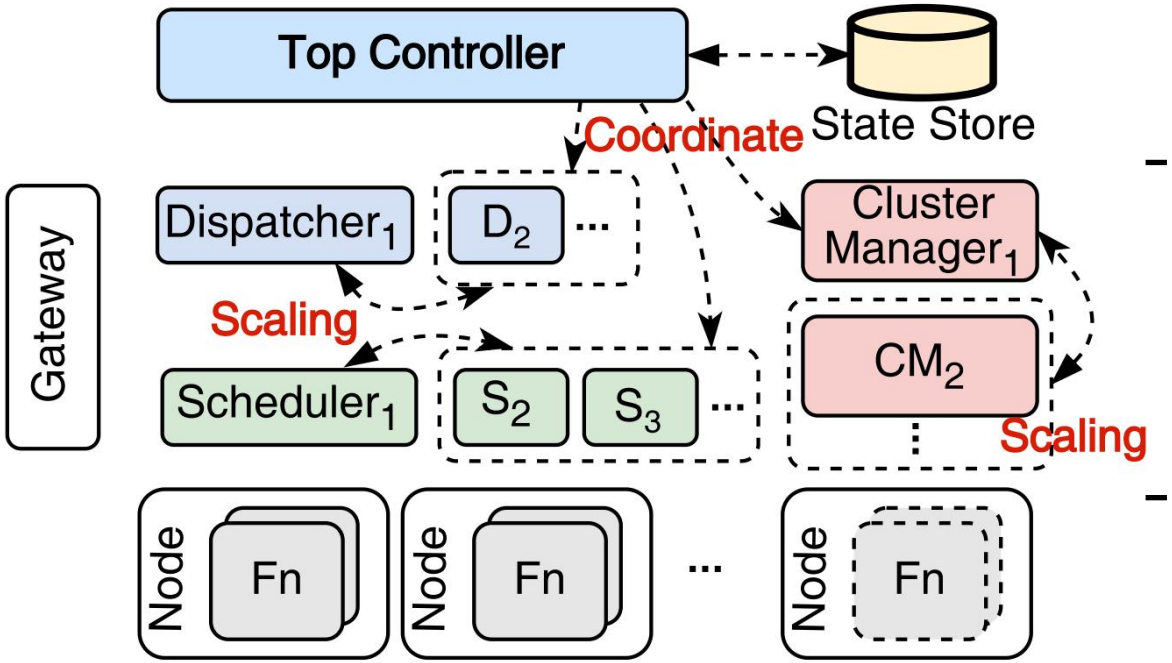
静态控制平面　　弹性控制平面

挑战

- 如何将控制平面解耦为控制函数？
- 如何确保函数实例的快速调度？
- 如何根据负载扩缩容每个控制函数？

我们的设计

- **弹性Serverless控制平面**



顶层控制平面

函数化控制平面

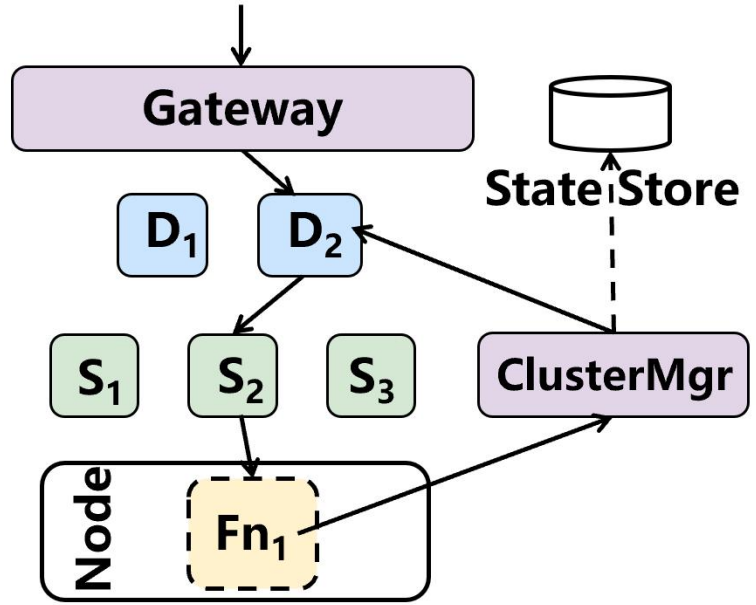
数据平面

- **控制函数**

组件	定义	描述
Dispatch函数	func Dispatch(request)->InstanceAddr	将请求路由至可用实例，无可用实例时触发扩容
	func ModInstance(funcName,address) ->StatusCode	
	func Coornadite(actions)->StatusCode	
Schedule函数	func Schedule(funcConstarints)->StatusCode	为新扩容实例选择可用节点
	func ModNode(nodeStatus,address) ->StatusCode	
	func Coornadite(actions)->StatusCode	
ClusterMgr函数	func Update(nodeStatus)->StatusCode	同步节点状态

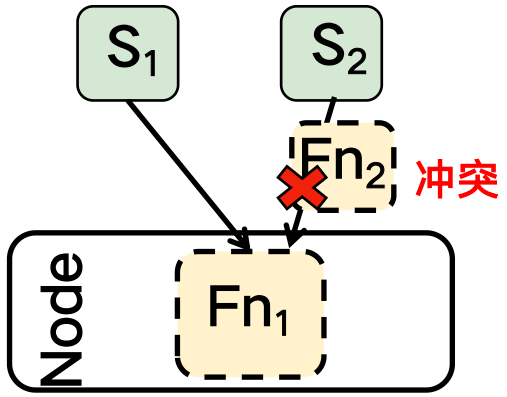
我们的设计

- **控制函数**
  - 计算瓶颈
    - 各个控制函数可以单独扩容以提高吞吐
  - 通信瓶颈
    - 控制函数实例内部维护下游函数的地址，以函数调用的方式进行控制函数间通信
    - 异步状态存储保证扩容的关键路径上无集中式组件



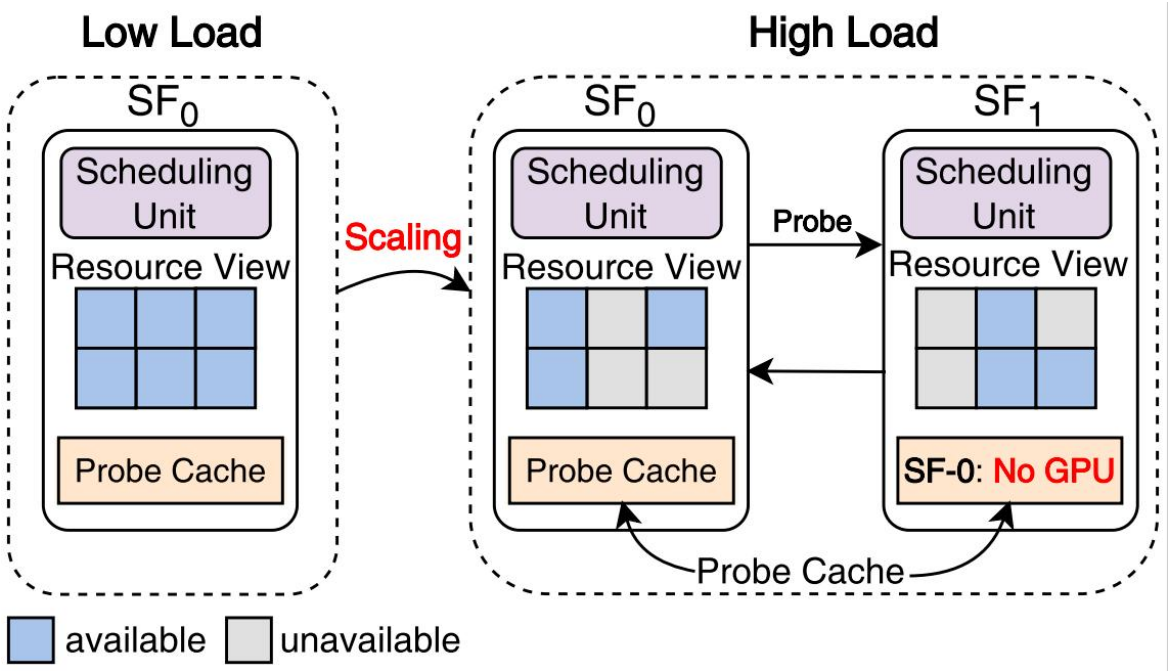
我们的设计

- **快速调度**
  - 分布式组件在并行决策时会导致冲突，影响调度的开销
  - 冲突处理
    - 引入集中式组件会导致显著的调度开销，如Ray
    - 节点级的冲突检测会导致大量的重调度，限制吞吐提升，如Sparrow



我们的设计

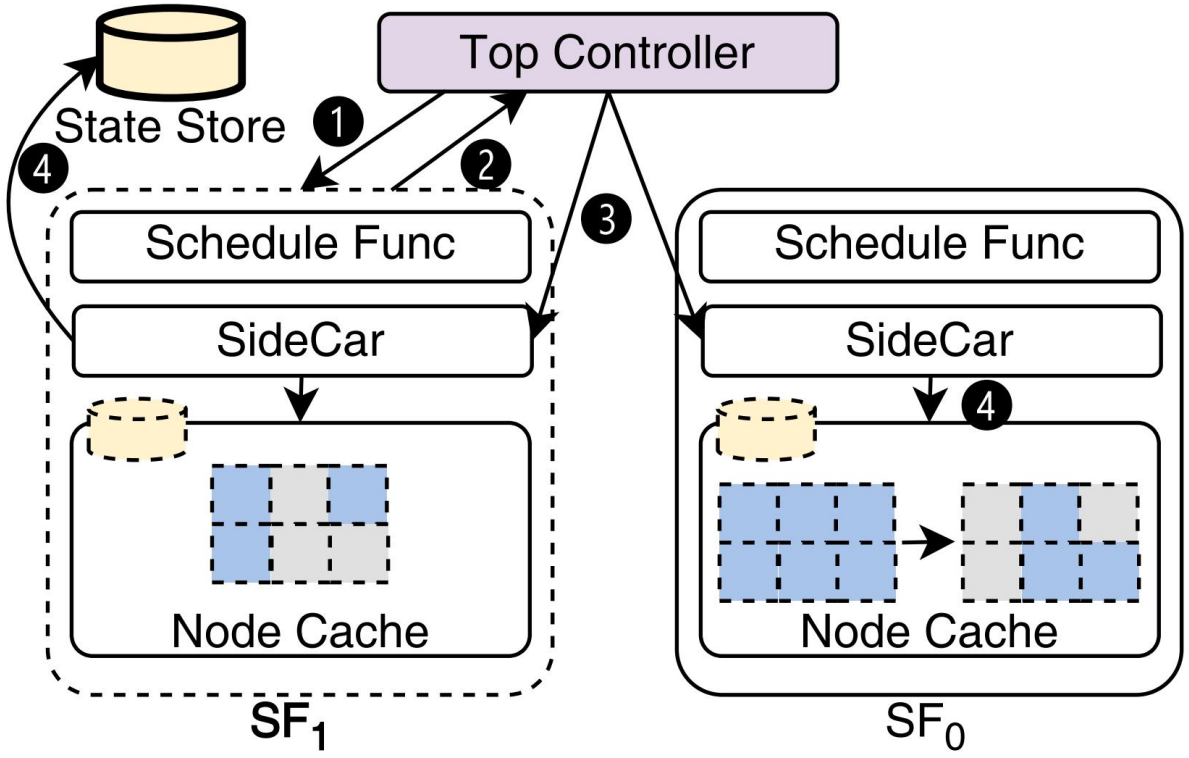
- **快速调度**
  - **动态分区**：避免冲突，权衡调度质量和调度速度
  - **探测**：避免引入中心瓶颈，支持探测缓存进行预先过滤



- **控制平面的灵活调整**
  - 指标
    - 系统每秒扩容量：Scaling Request Per Second (SRPS)
    - 每个控制函数的最大吞吐：T_{DF}, T_{SF}, T_{CMF}
  - 决策
    - 由Dispatch函数定期向顶层控制器汇报系统负载
    - 控制函数的吞吐超过阈值时主动申请扩容
    - 顶层控制器统一决策，避免冗余的控制函数扩容

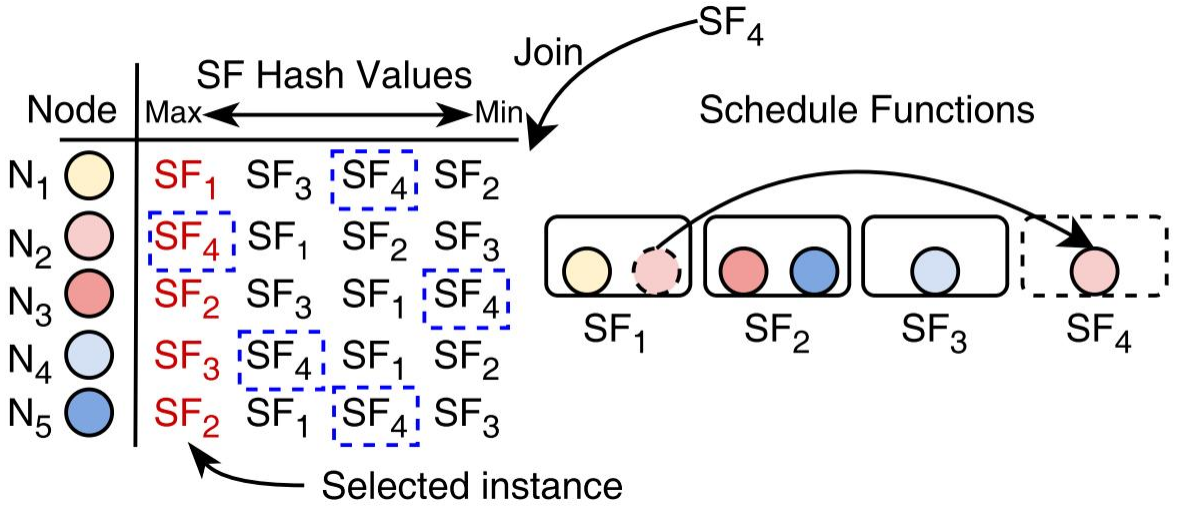
我们的设计

- **控制平面的灵活调整**
  - 扩容流程
    - 选择可用节点启动控制函数实例
    - 注册顶层控制器
    - 更新控制函数实例分区
    - 从状态存储获取状态缓存



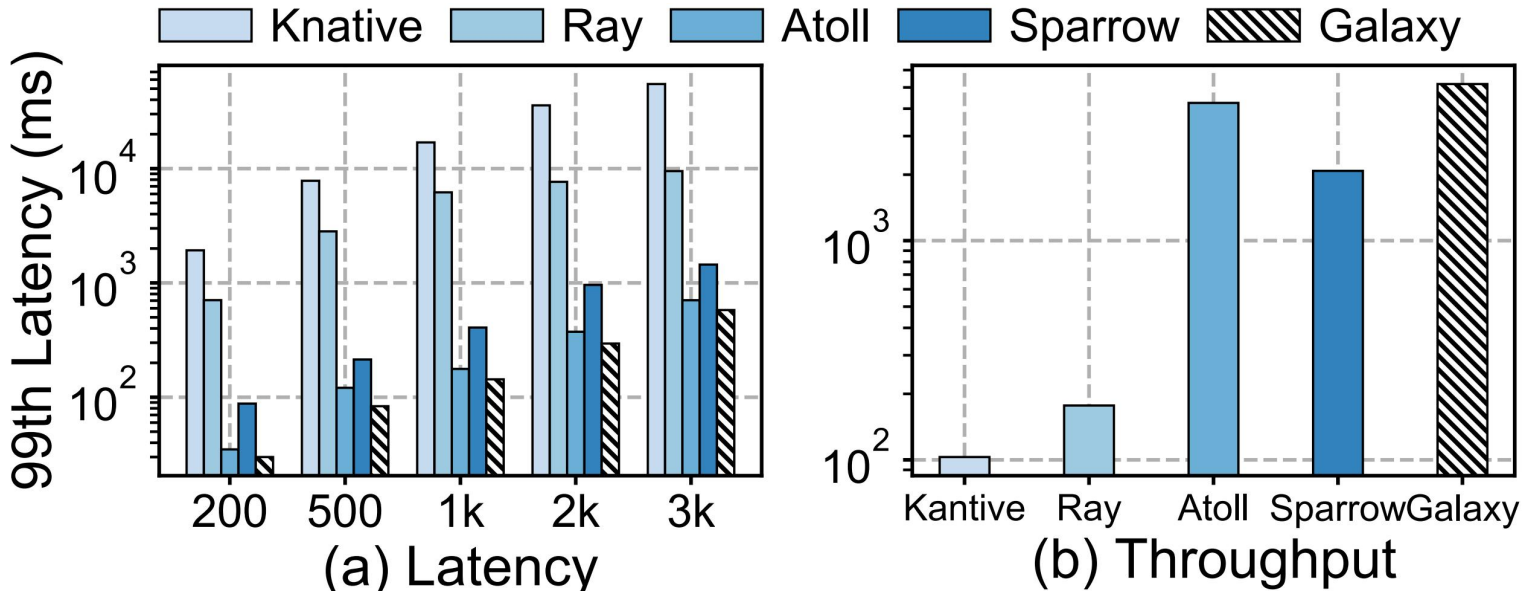
我们的设计

- **控制平面的灵活调整**
  - 实例重分区
    - 分区平衡
    - 最小化视图的变更
  - 会合哈希算法
    - 计算所有的<节点，调度实例>的哈希值
    - 将节点分配给哈希值最大的调度实例，如将N_1分配给SF_1
    - 在增加和删除实例时只更新最大哈希值改变的节点的分区，如仅有N_2被重新分配给SF_4

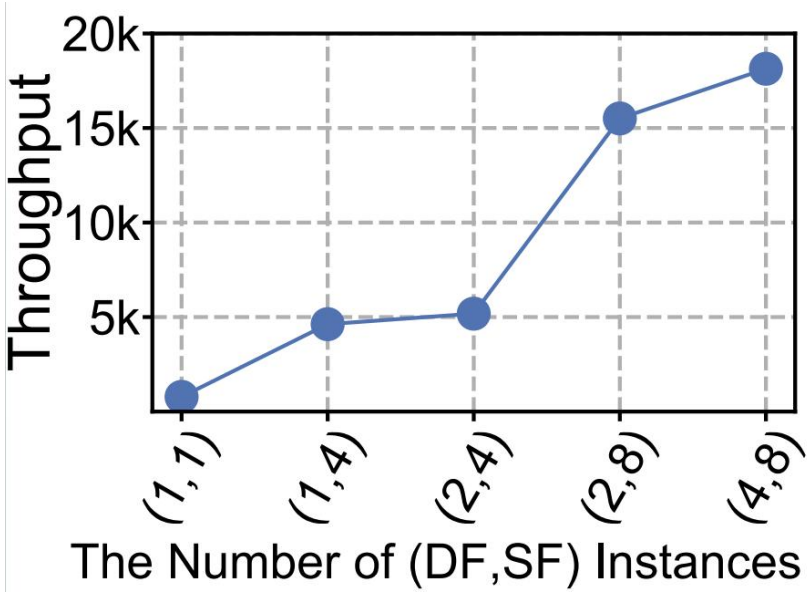


- **实验设置**
  - 210个节点组成的集群
  - 对比系统
    - Knative：集中式架构
    - Sparrow：全分布式架构
    - Ray：自底向上架构
    - Atoll：双层架构

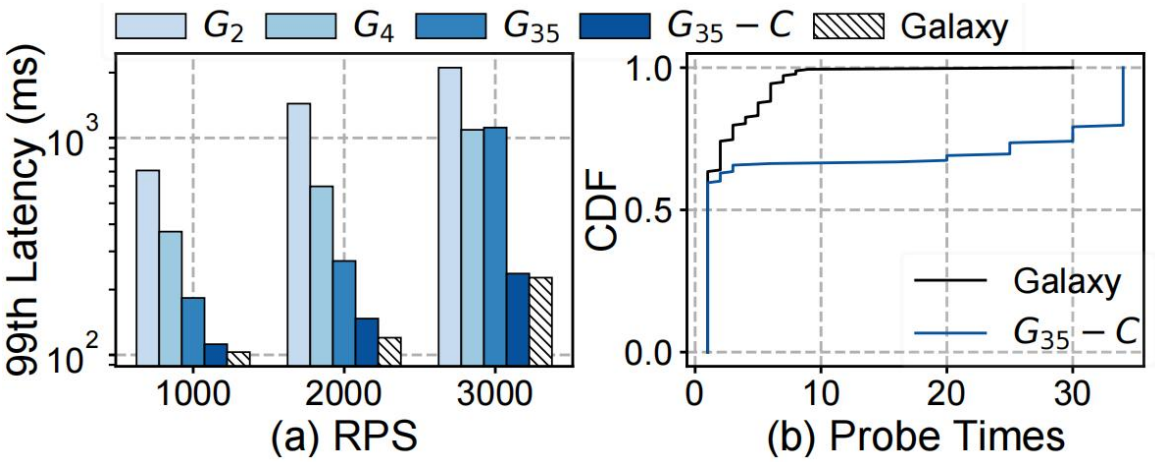
- **整体性能**
  - 调度尾延迟：降低 87.9%~95.3%
  - 系统吞吐：提高 1.2~29.3 倍



- **拓展性**
  - 最多每秒可拓展**2万**个函数实例
  - 增加单个控制函数的实例对吞吐的影响有限

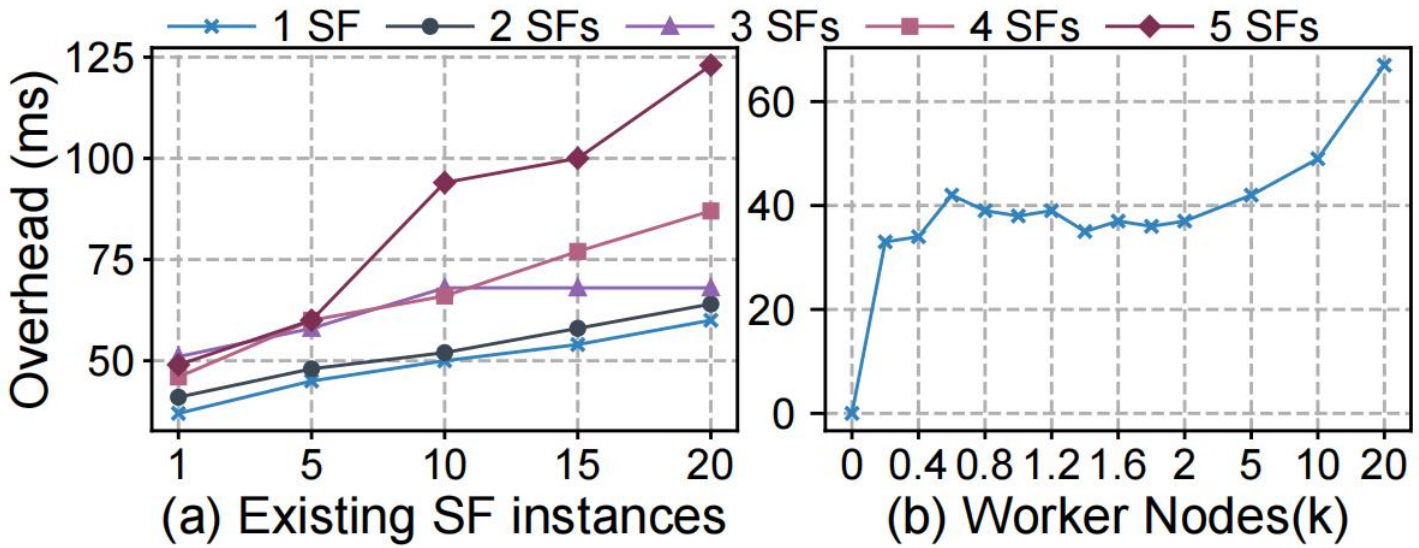


- **优化分析**
  - 控制函数扩容可以显著降低调度延迟
  - 连接缓存可以降低38.7-78.7%的开销
  - 探测缓存减少了79%的探测次数，降低75.9%的探测开销

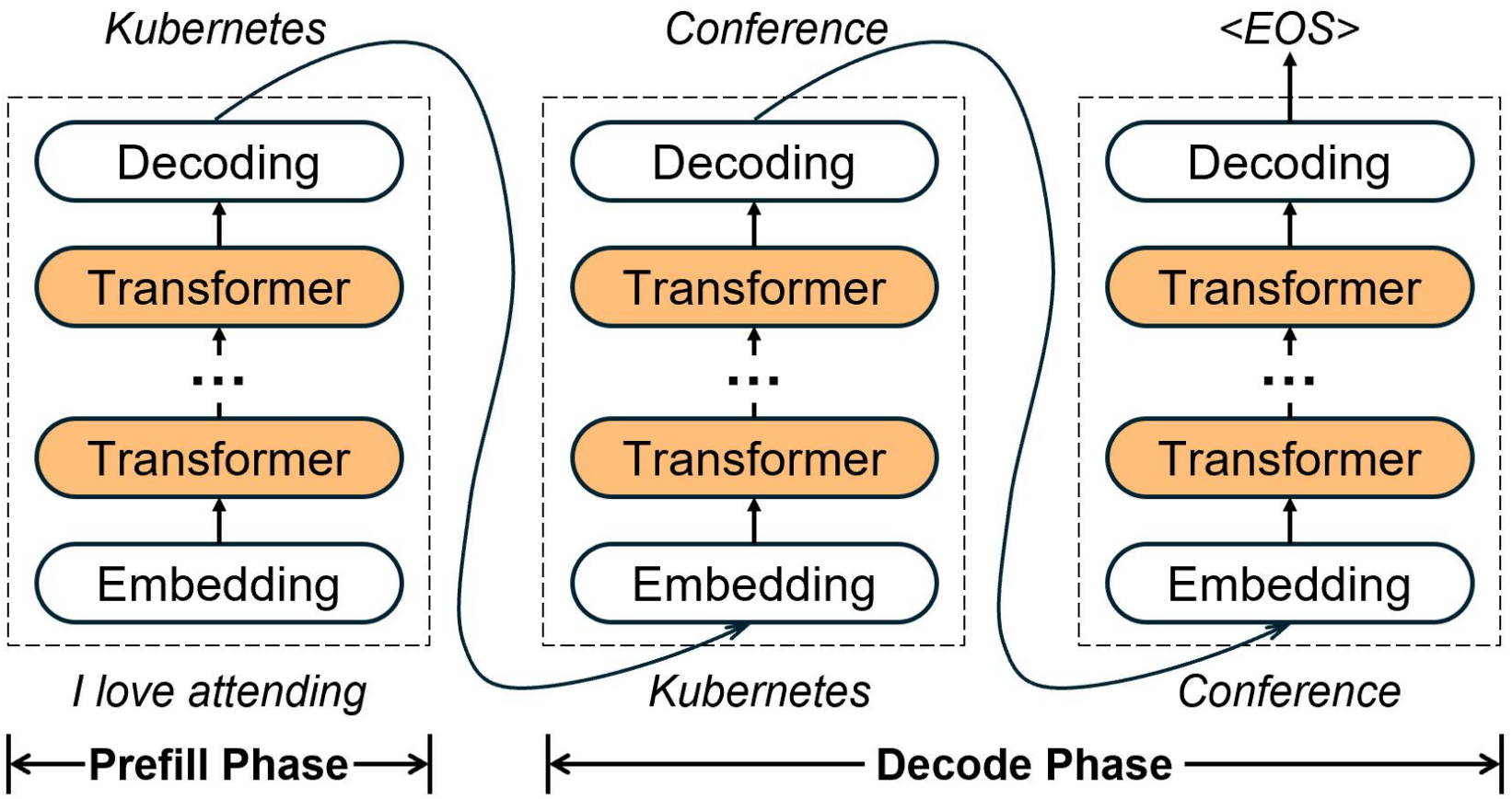


Mode	Description
G_n	n instances of each control function in the control plane
$G_{35}-C$	Adding Connection Cache to the G_{35}
Galaxy	Adding a probe cache to the $G_{35}-C$

- **开销**
  - 受到已有实例数量、扩容实例数量以及集群大小的影响
  - 扩容1个调度函数的重分配开销至少为30ms



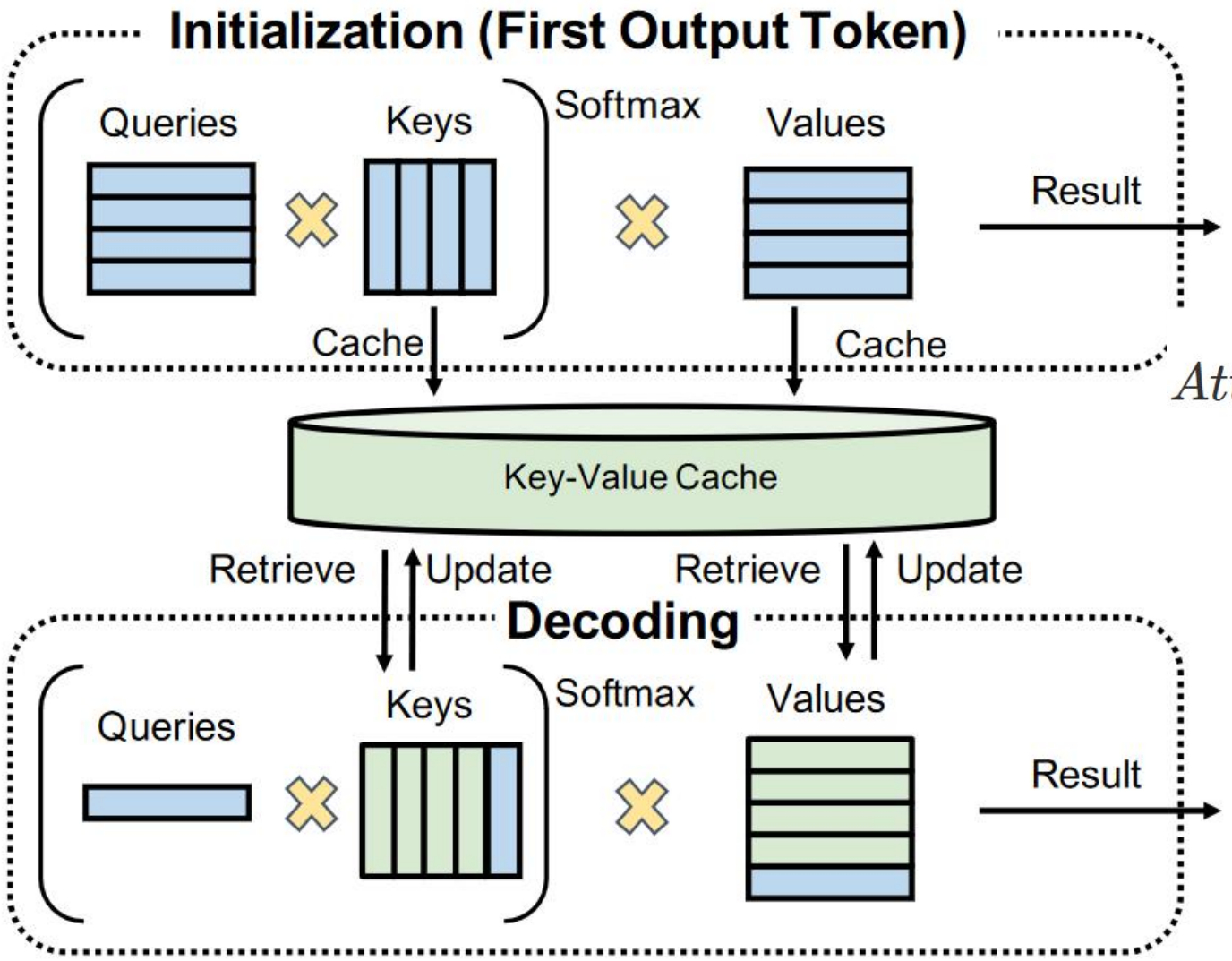
- **自回归模型**



Prefill和Decode

- Prefill
  - 推理的初始阶段
  - 并行处理用户输入的所有的token生成第一个新token
  - 计算密集型
- Decode
  - 自回归式地每次产生一个新token
  - 每个新生成的token都需要之前所有迭代轮次的输出状态
  - 访存密集型

KV Cache



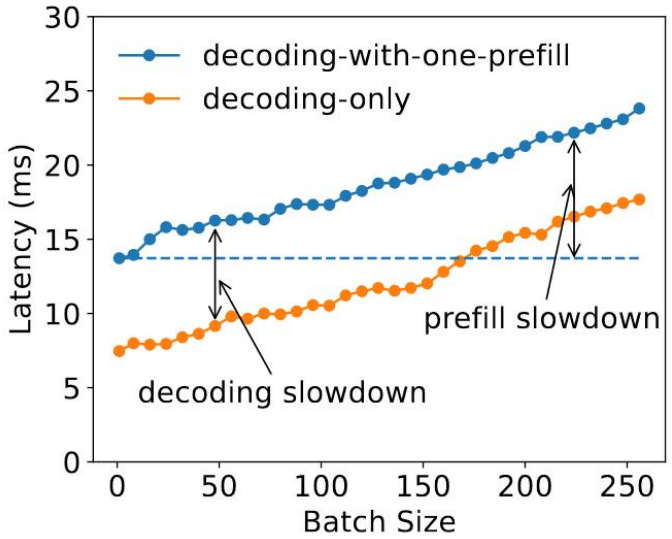
- Self-Attention

$$Attention(Q, K, V) = Softmax(\frac{QK^\top}{\sqrt{d_k}})V$$

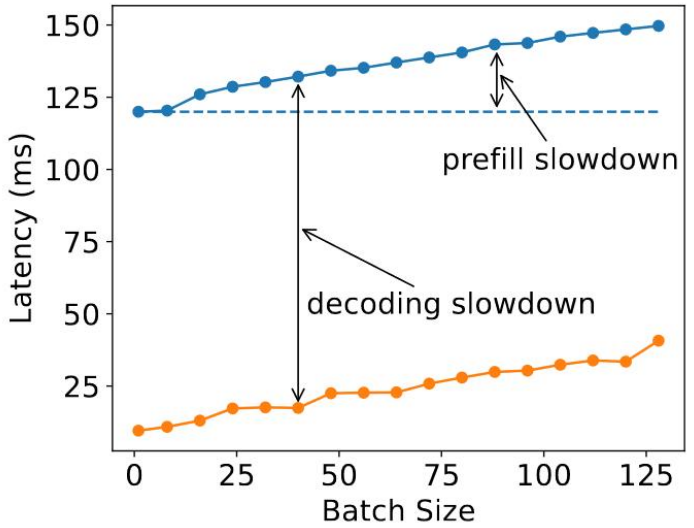
- KV Cache
  - 新计算的张量
  - 被复用的张量

Prefill 和 Decode

- **Prefill 和 Decode 阶段的干扰**
  - 将 Prefill 与 Decode 请求进行批处理会显著降低两个阶段的推理速率



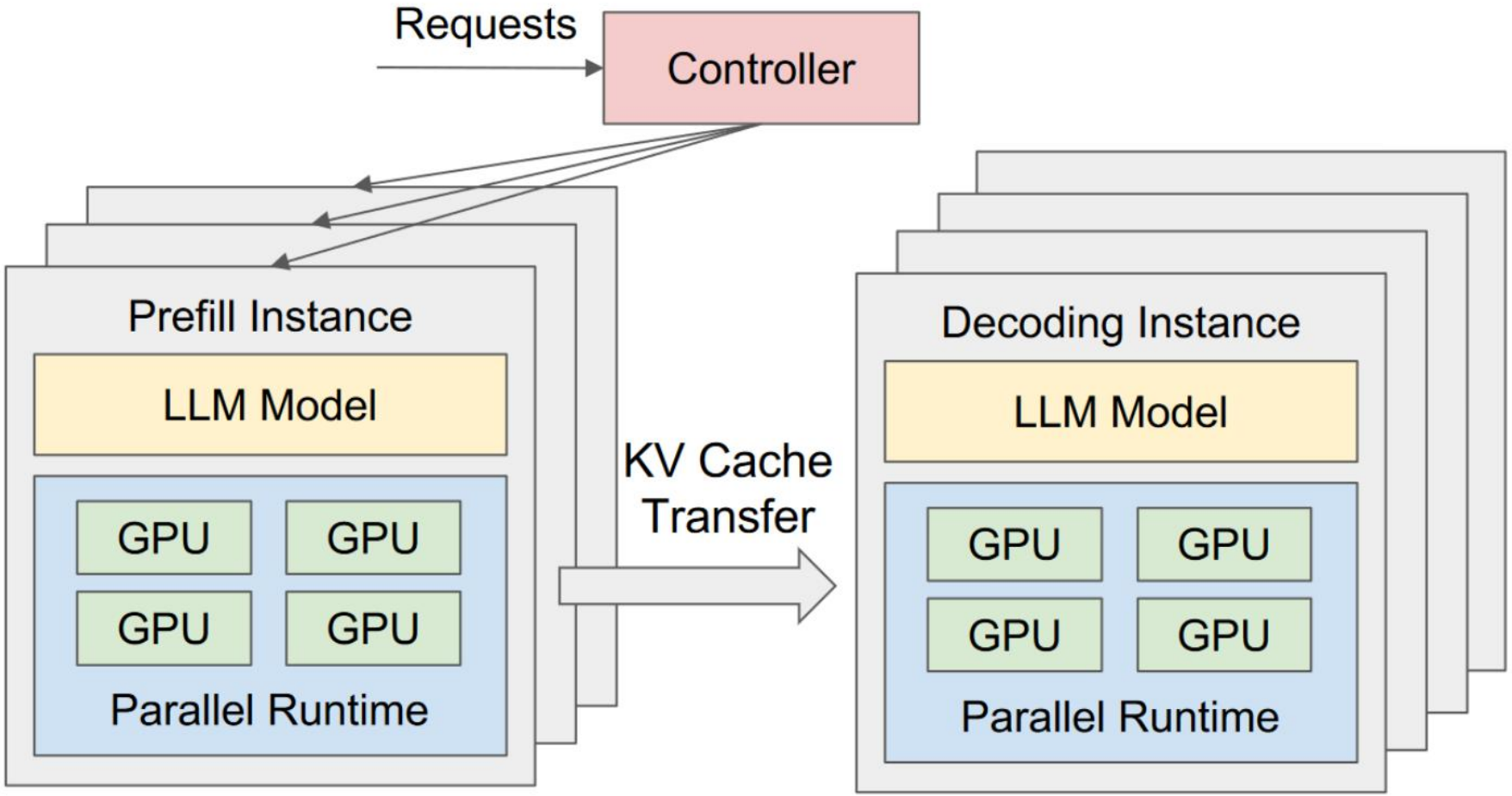
(a) Input length = 128



(b) Input length = 1024

Prefill 和 Decode

- **Prefill 和 Decode 解耦**



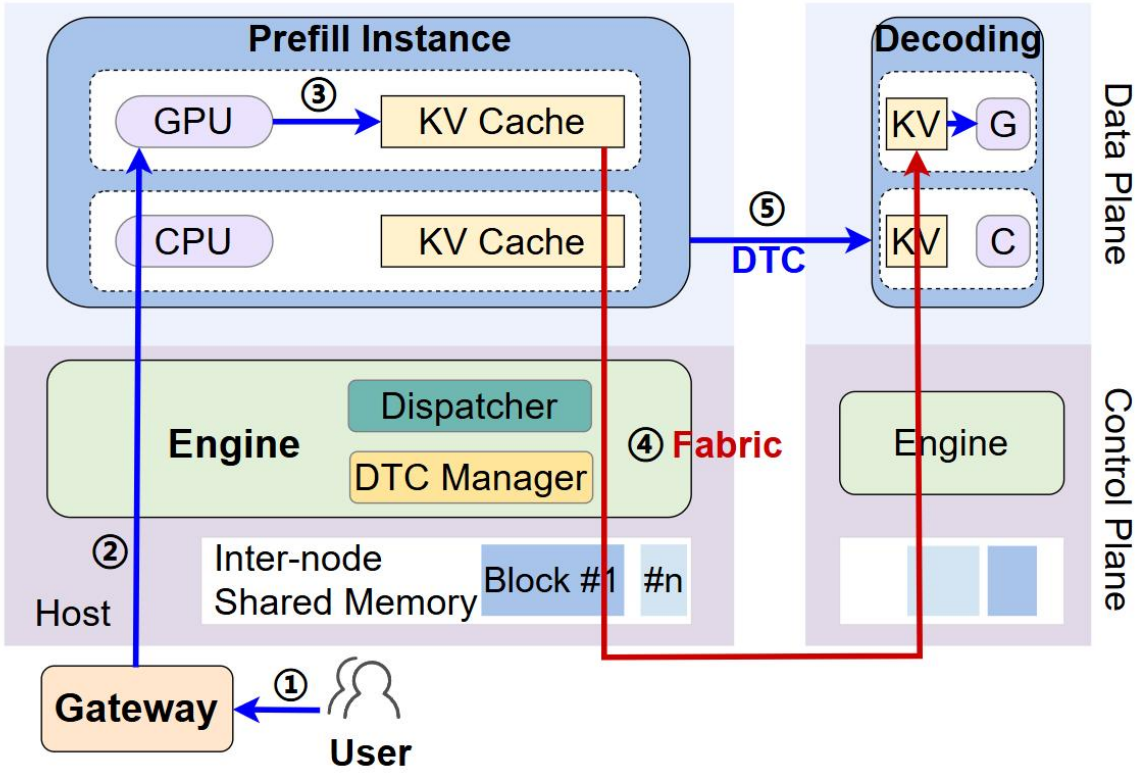
- **传输 KV Cache 的开销不容忽视**
  - KV Cache 的体积随着上下文长度线性增长
  - 上下文长度较大 KV Cache 的体积甚至远远超过模型参数大小

Context length	10k	100k	500k	1000k
KV Cache size	8.19GB	81.9GB	409.6GB	819.2GB
Misc size	26GB	26GB	26GB	26GB

LLaMA2-13B模型中不同上下文长度的KV Cache体积

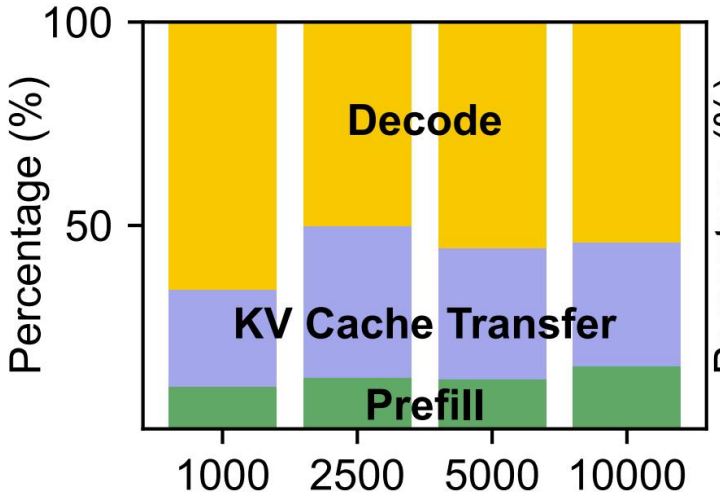
我们的设计

- 基于RDMA的KV Cache直传
  - 控制引擎
    - 数据转发
    - 直连通道建立
    - 传输方法选择
  - 直连通道
    - 基于RDMA的双全工连接
    - 避免冗余数据传输
    - 本地 GPU → 本地 CPU → 远端 CPU → 远端 GPU

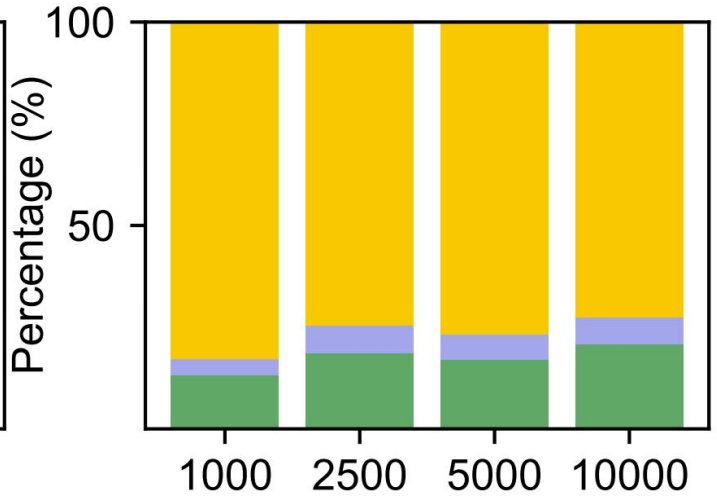


实验结果

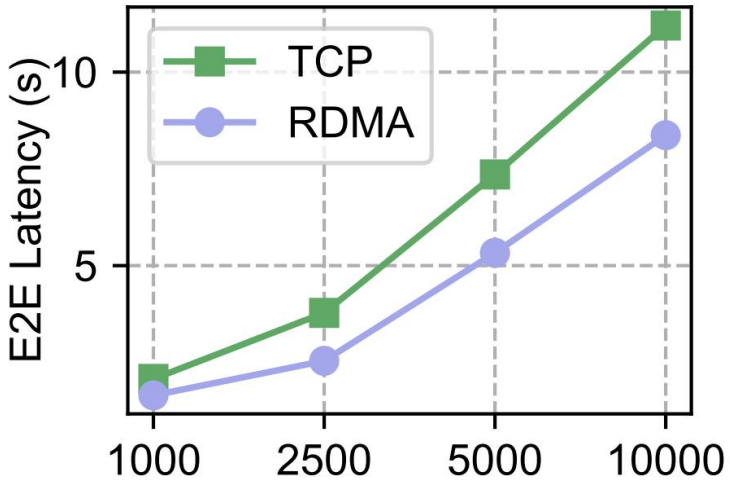
- Llama-3-70B 不同 Token 大小的性能对比
  - 传输性能：加速6~8.1倍，占比从23.7%~37.4%降至4~6.8%
  - 总延迟：降低20.5%-32.9%



TCP传输时各阶段比例



RDMA传输时各阶段比例



总延迟

总结

- 问题：Serverless大模型推理的调度和 KV Cache 传输开销大
- 方案：函数化弹性调度和RDMA直连通信
- 结果：调度吞吐提高29倍，传输加速8.1倍