KubeCon CloudNativeCon

THE LINUX FOUNDATION

OPEN SOURCE SUMMIT

AI_dev

Open Source GenAI & ML Summit

China 2024


KubeCon
CloudNativeCon
THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
AI_dev
Open Source GenAI & ML Summit
China 2024
Accelerating Serverless AI Large Model Inference with Functionalized Scheduling and RDMA
Wang Chenglong && Li Yiming

- What’s KServe
- Best Practices Based on KServe in Inspur InCloud OS
- Accelerating Serverless AI Large Model Inference with Functionalized Scheduling

Kubernetes + AI 趋势

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

Container and serverless become the primary platforms for future AI application

Kubernetes become the standard for cloud-native applications. According to the Cloud Native Computing Foundation (CNCF) survey, in 2023, 66% of potential/actual consumers were using Kubernetes in production and 18% were evaluating it(84% total)

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

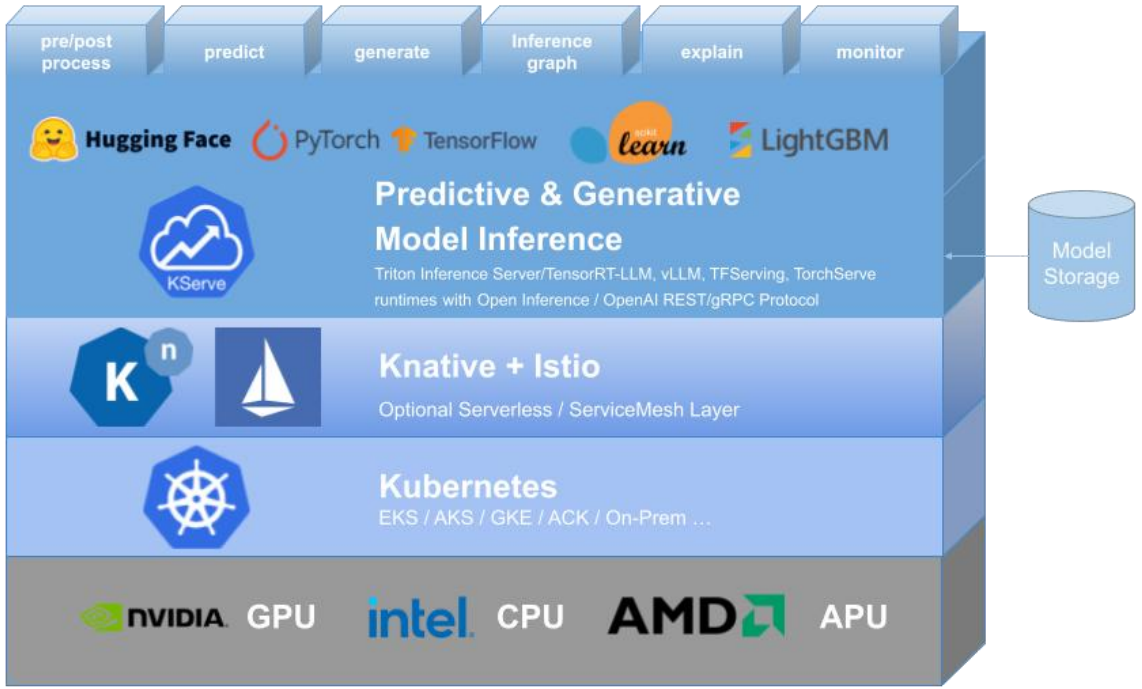
Hugging Face Collaborates with Microsoft to launch Hugging Face Model Catalog on Azure

[Cloud Native Artificial Intelligence Whitepaper]

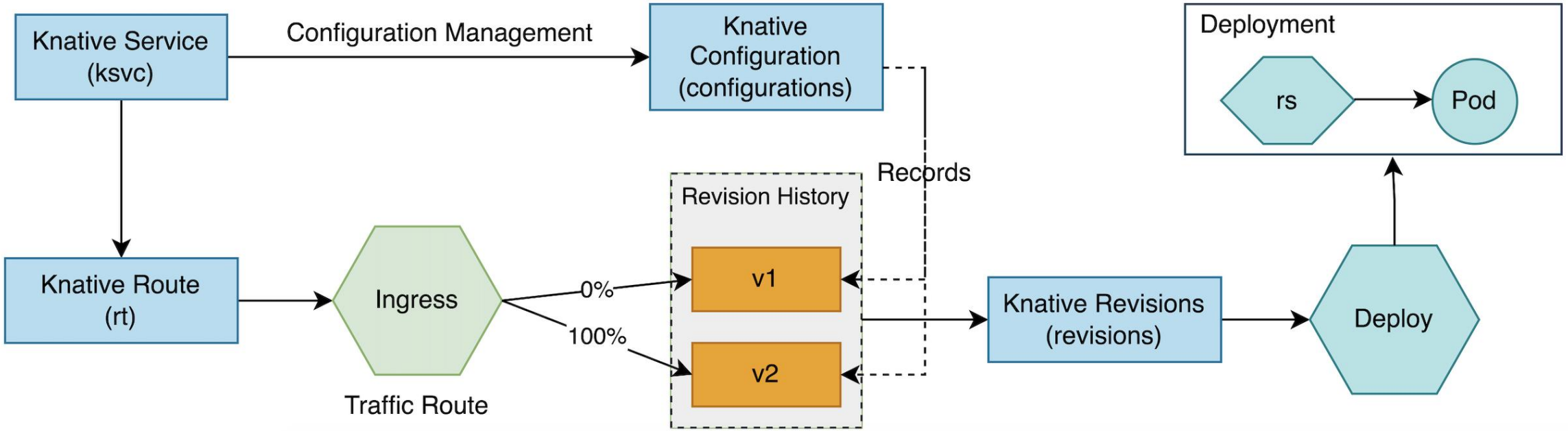
What's KServe

KServe is a standard Model Inference Platform on Kubernetes, built for highly scalable use cases

- Provides performant, standardized inference protocol across ML frameworks
- Support modern serverless inference workload with Autoscaling including Scale to Zero on GPU
- Provides high scalability, density packing and intelligent routing using ModelMesh
- Simple and Pluggable production serving for production ML serving including prediction, pre/post processing, monitoring and explainability
- Advanced deployments with canary rollout, experiments, ensembles and transformers



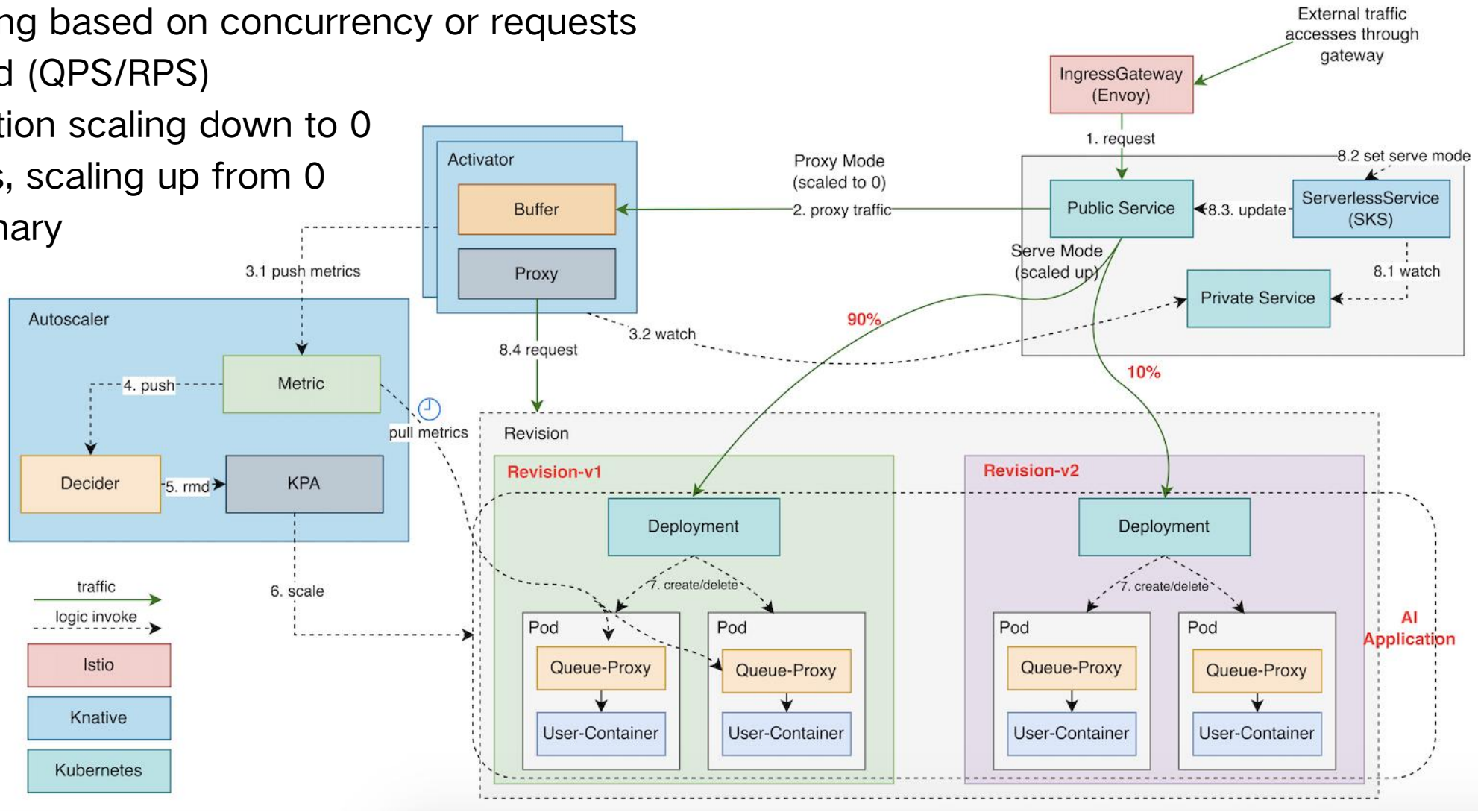
Serverless Layer —— Knative Serving



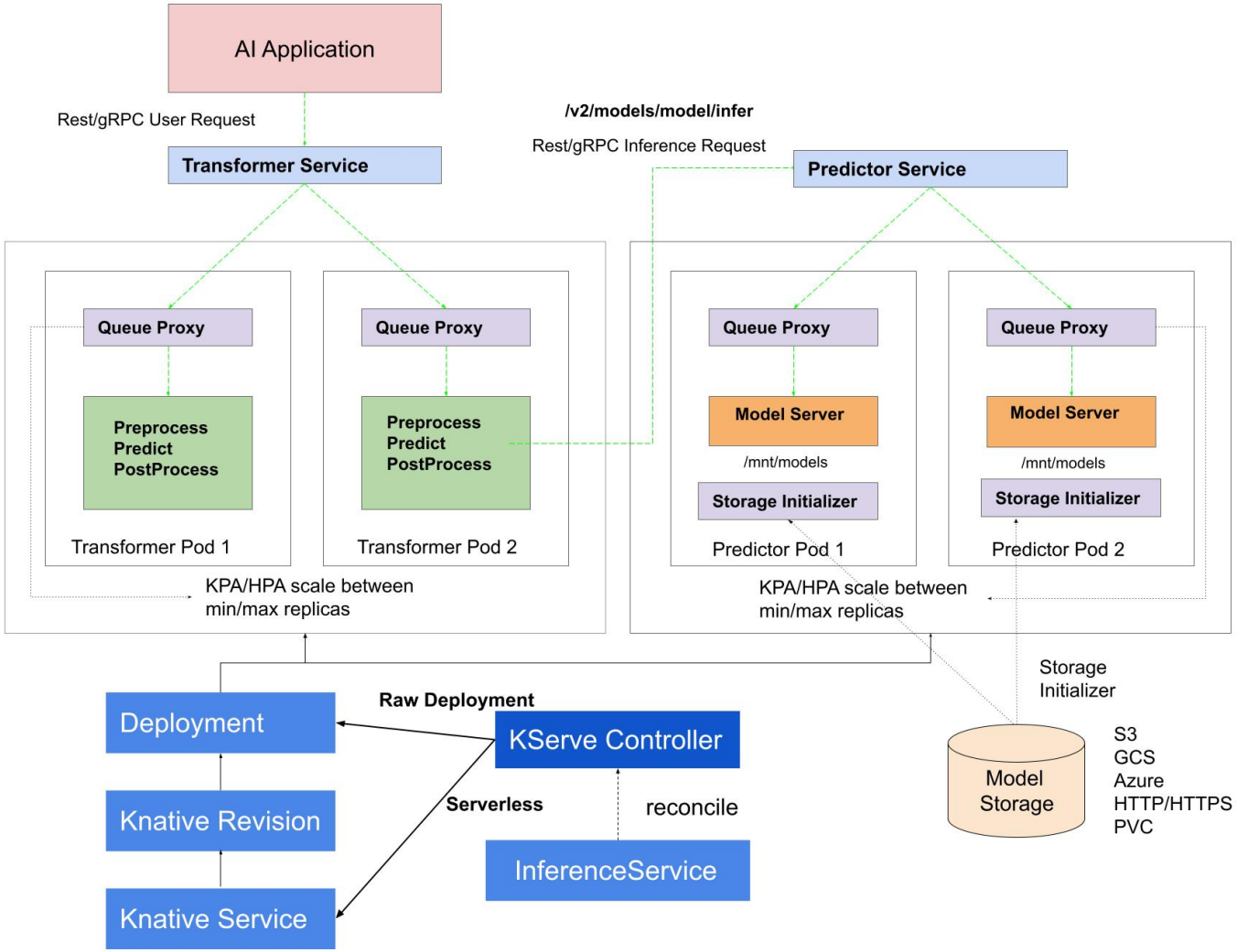
- **Service**: Abstraction of serverless applications, automatically manages the whole lifecycle of your workload
- **Configuation**: Maintains the desired state for your deployment. Modifying a configuration creates a new revision
- **Revision**: Point-in-time snapshot of the code and configuration for each modification made to the workload
- **Route**: Routing rules used to define traffic distribution, directing network requests to specific Revisions

Serverless Layer —— Knative Serving

- Auto-scaling based on concurrency or requests per second (QPS/RPS)
- AI Application scaling down to 0
- Cold starts, scaling up from 0
- Traffic Canary



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
```

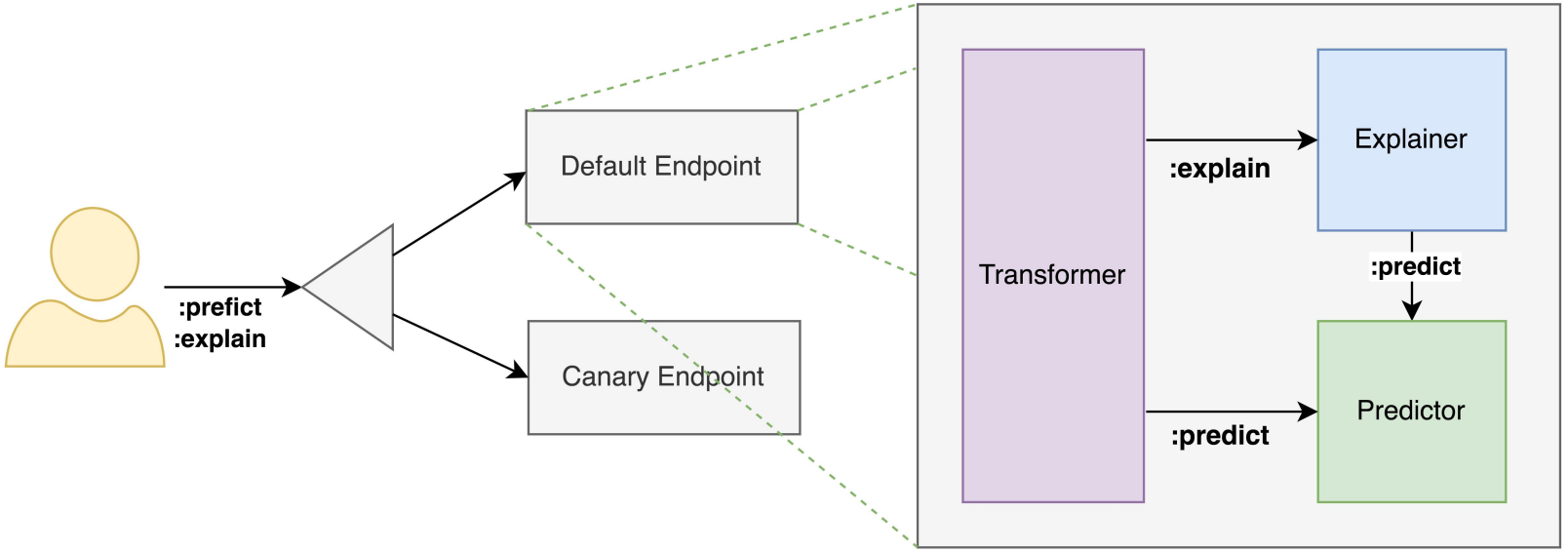
3 Perform inference

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

KServe —— Data Plane

Each endpoint is composed of multiple components: "predictor", "explainer", and "transformer"

- **Predictor**: The workhorse of the InferenceService. It is simply a model and a model server that makes it available at a network endpoint
- **Explainer**: Provides model explanations. Users may define their own explanation container, which configures with relevant environment variables like prediction endpoint
- **Transformer**: Enables users to define a pre and post processing step before the prediction and explanation workflows



KServe —— Data Plane Protocol

KServe supports multiple protocols, including REST and gRPC, making model deployment and service invocation more standardized and unified

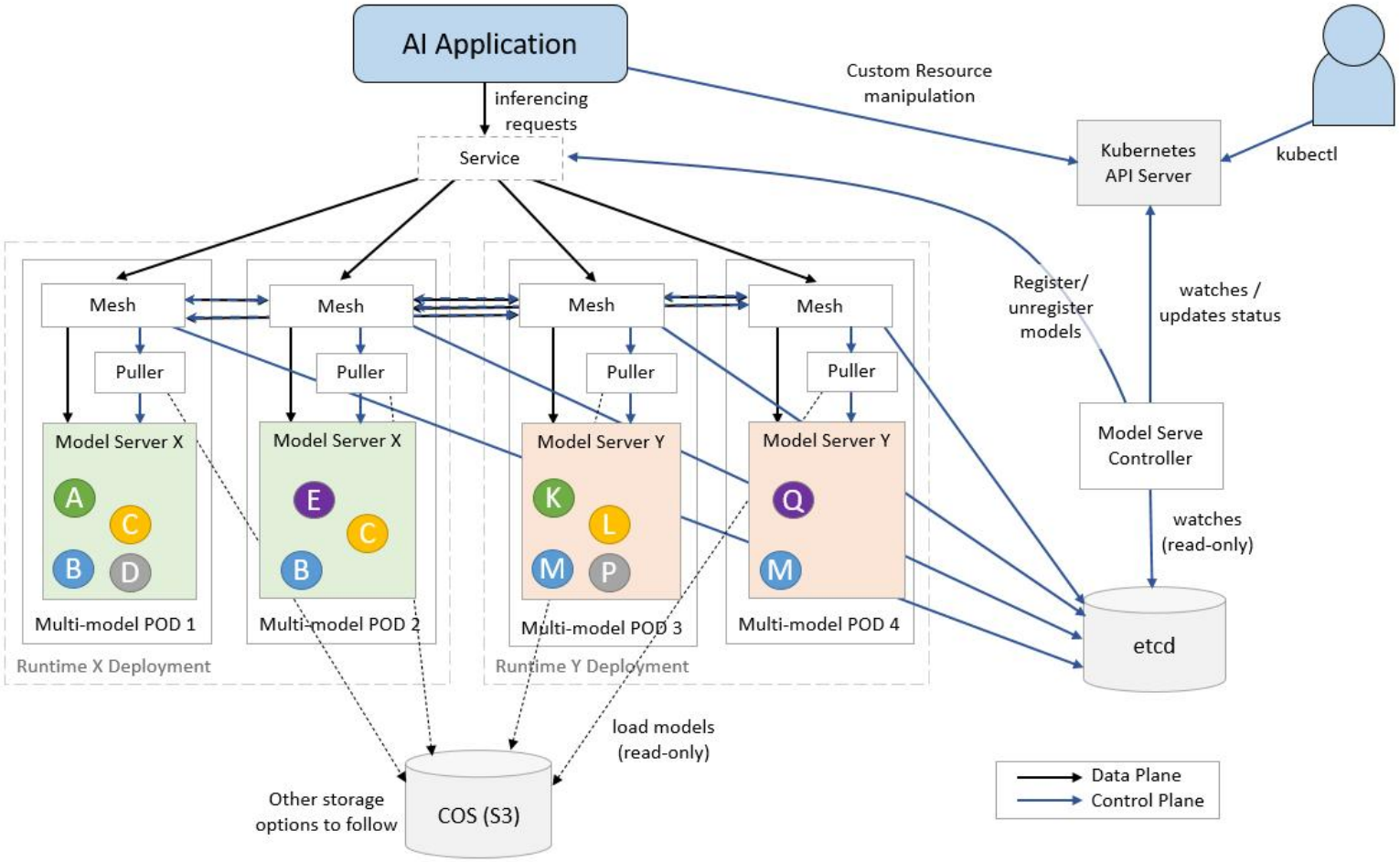
REST（v1 Inference Protocol）
GET /v1/models
GET /v1/models/{model_name}
POST /v1/models/{model_name}:predict
POST /v1/models/{model_name}:explain

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

Multi-model serving is designed to address three types of limitations KServe will run into:

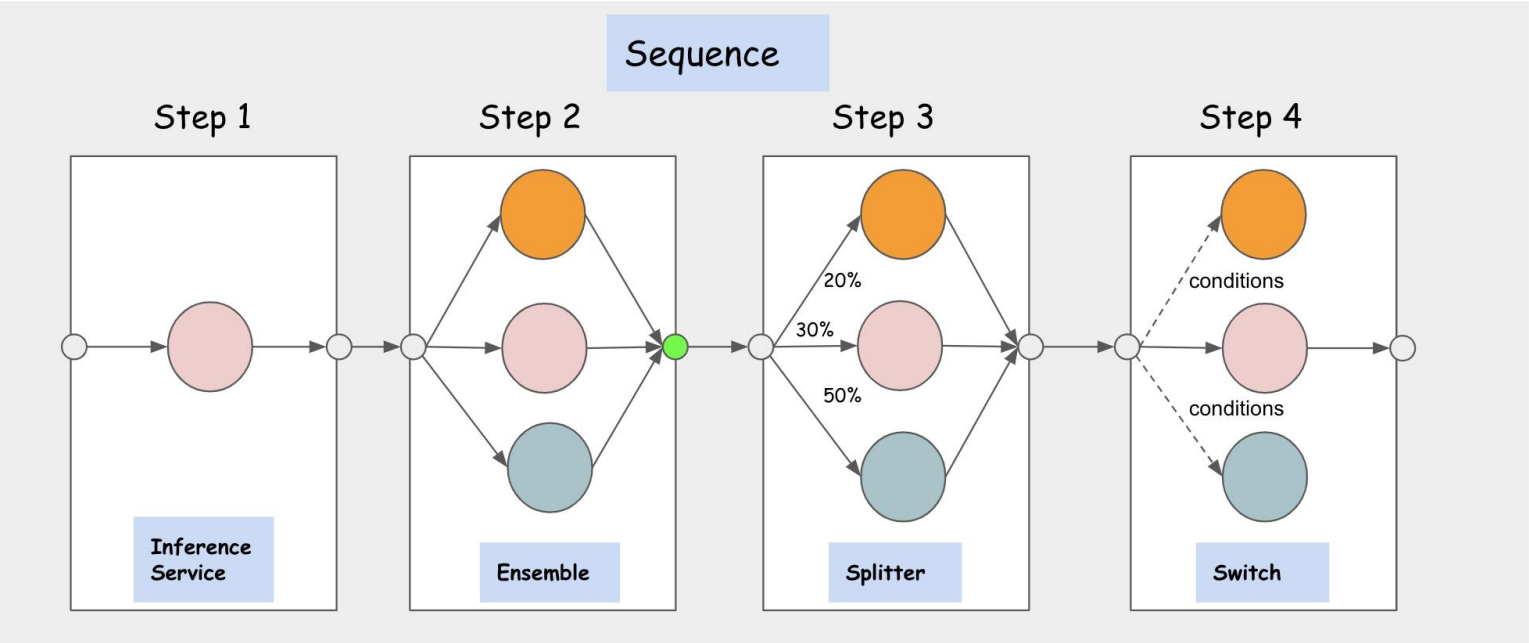
- **Compute resource limitation** (Each InferenceService has a resource overhead because of the sidecars injected into each pod)
- **Maximum pods limitation** (With K8s 110 limitation, a typical 50-node cluster with default pod limit can run at most 1000 models)
- **Maximum IP address limitation** (4096 IP addresses can deploy at most 1024 models)



KServe —— Inference Graph

InferenceGraph enables users to deploy complex ML inference pipelines to production in a declarative and scalable way. Four node types that are supported: Sequence, Switch, Ensemble, Splitter

- **Sequence:** Execute multiple steps sequentially in the defined order
- **Ensemble:** Execute multiple steps in parallel and combine the results
- **Splitter:** Split input data into multiple parts and route them to different services
- **Switch:** Route traffic to different services based on specified conditions




```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

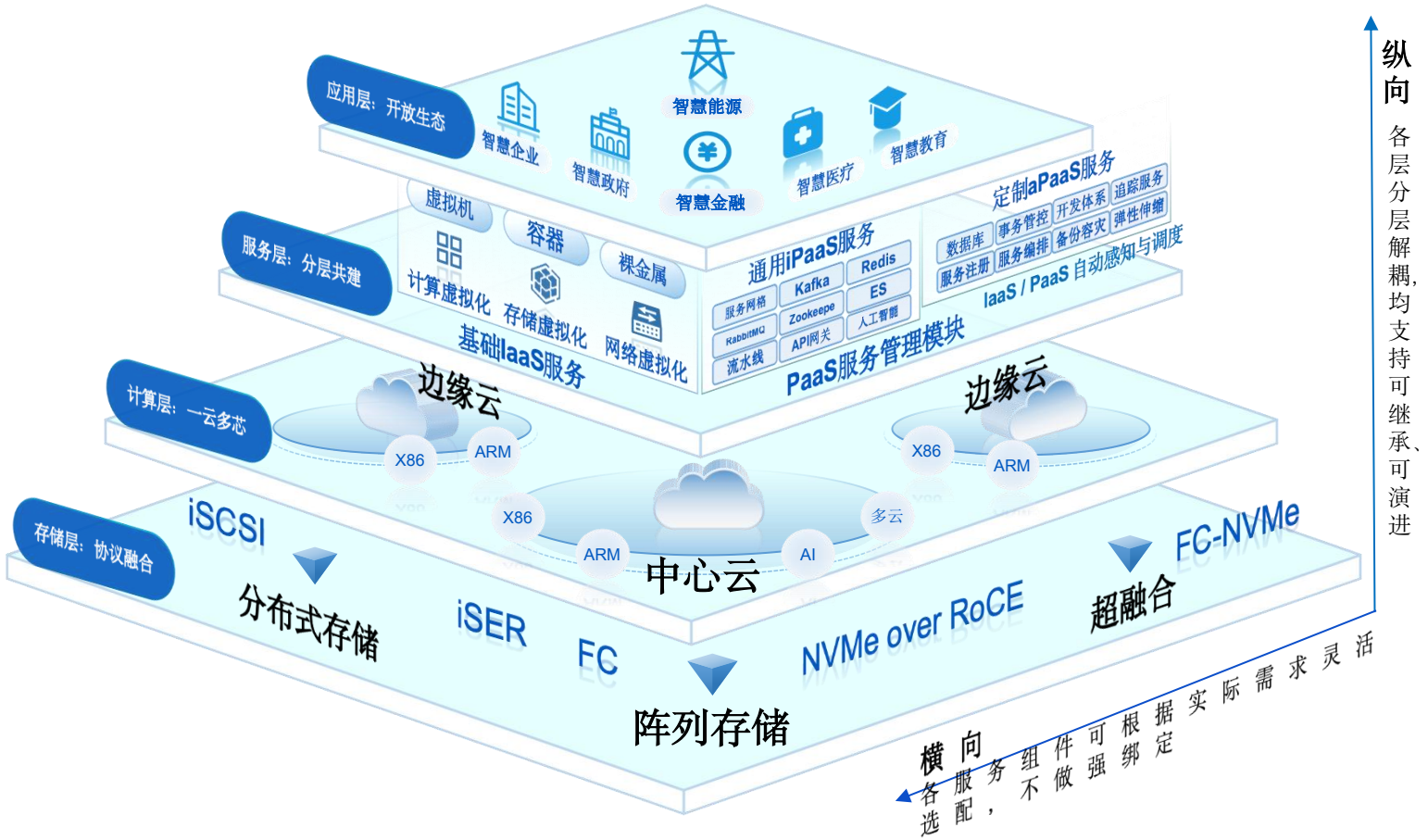
Open Source GenAI & ML Summit

China 2024

Best Practices Based on KServe in Inspur InCloud OS

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

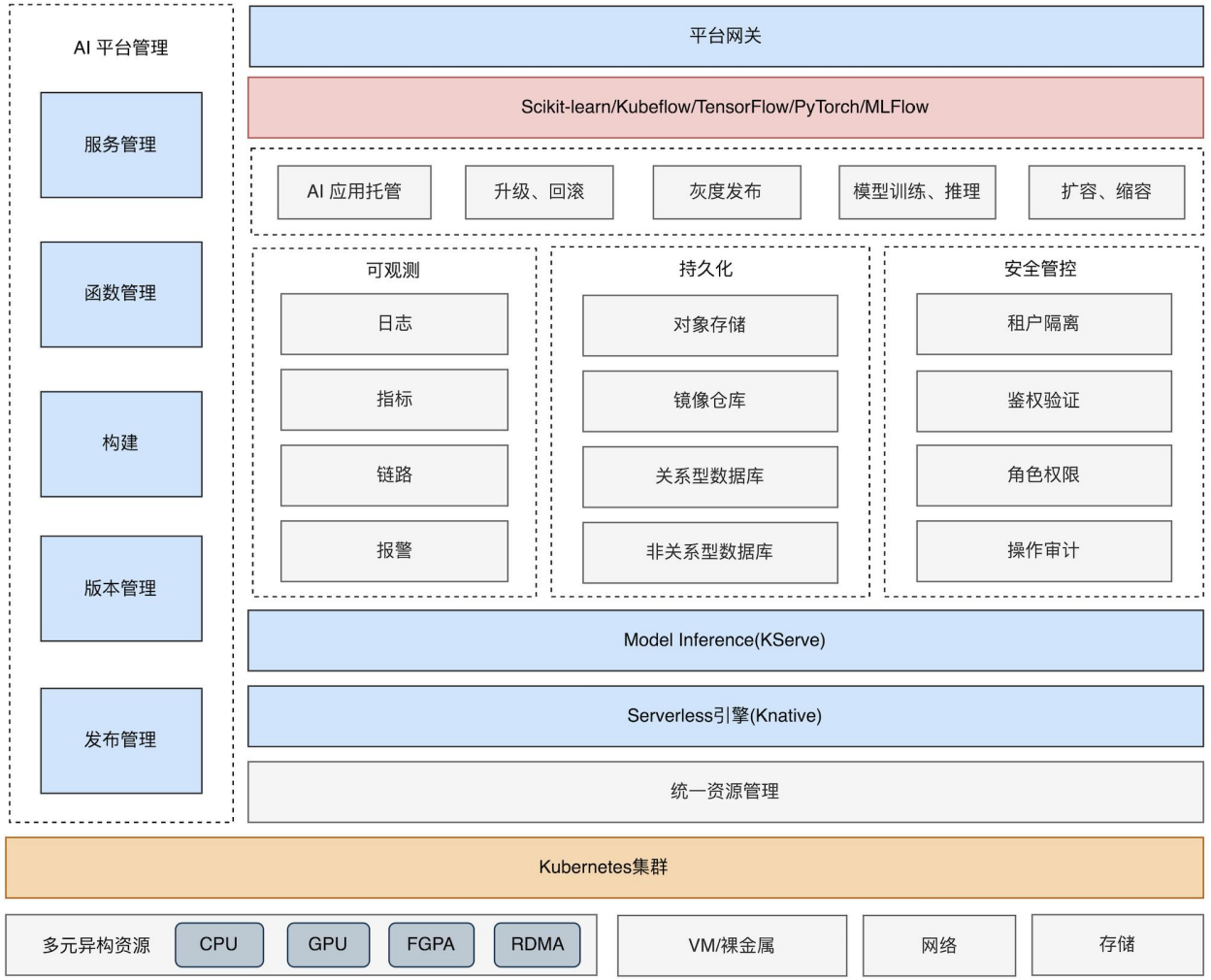


浪潮云海基于 KServe 的实践方案

浪潮云海 AI 平台技术架构

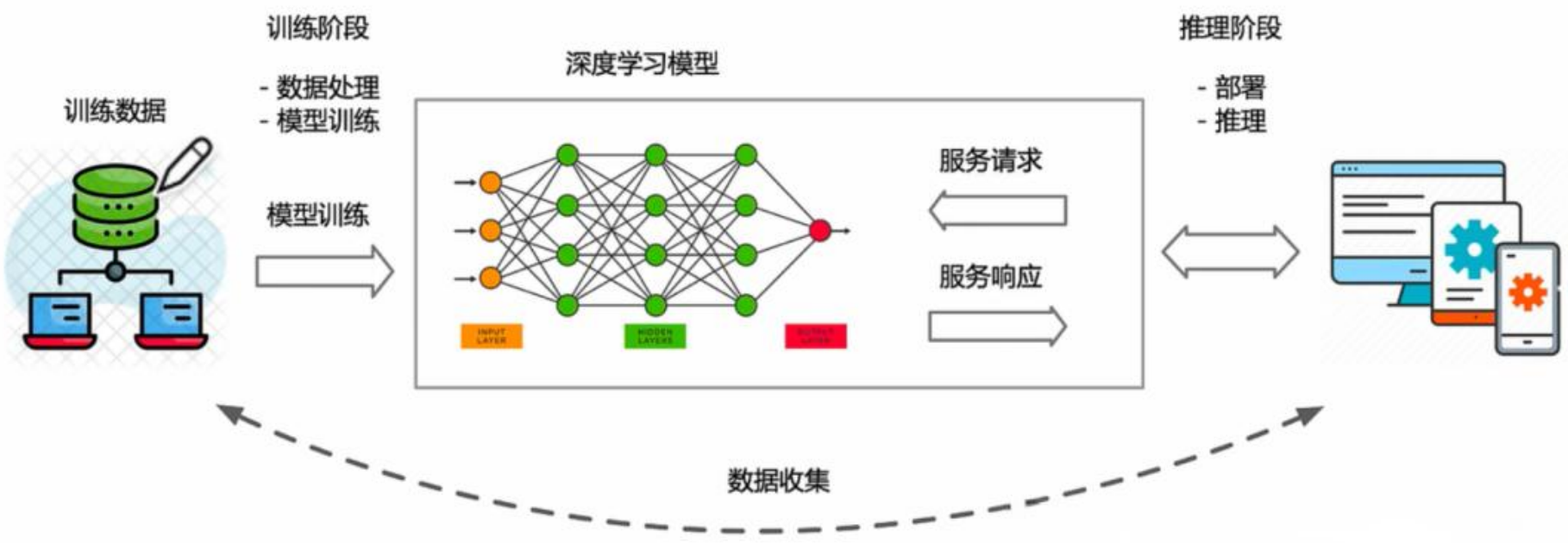
- 平台以容器为底座，构建于 Kubernetes 集群之上，以 Knative 为 Serverless 核心引擎，以 KServe 为模型推理框架，支撑多种 AI 业务场景

- 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"
```

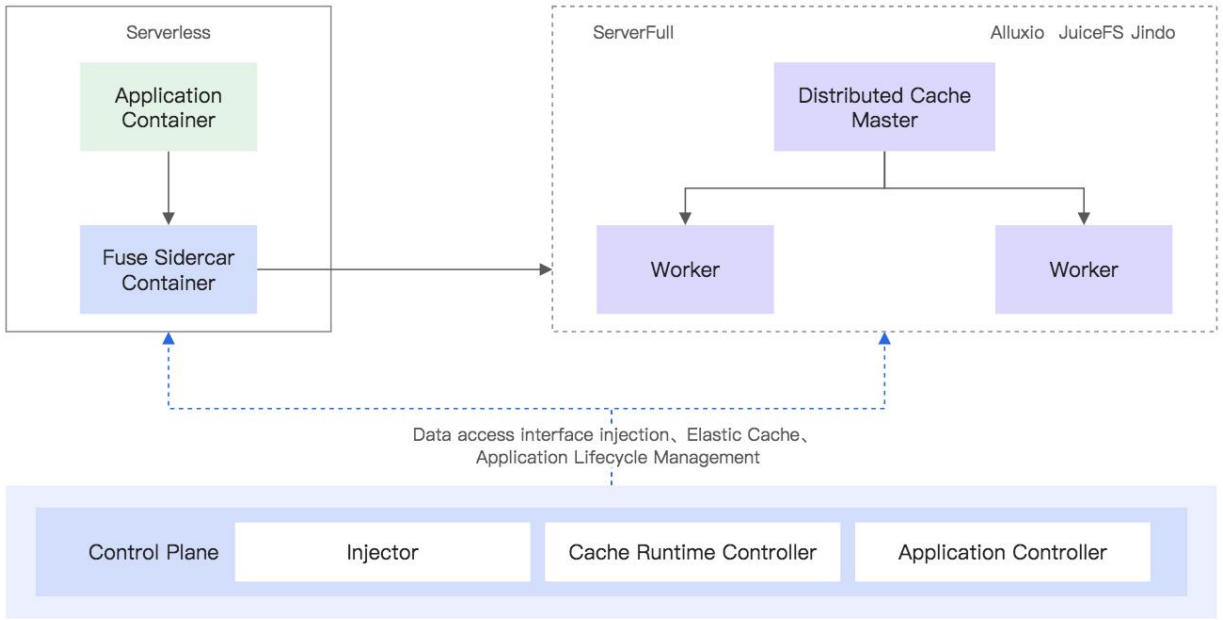
```yaml
apiVersion: serving.kserve.io/v1alpha1
kind: ServingRuntime
metadata:
  annotations:
    inspur.com/template-display-name: ServingRuntime for llm
  name: llm
spec:
  containers:
  - args:
    - --model=/mnt/models/
    - --served-model-name=llm
    - --tensor-parallel-size=2
    - --max-model-len=2048
    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

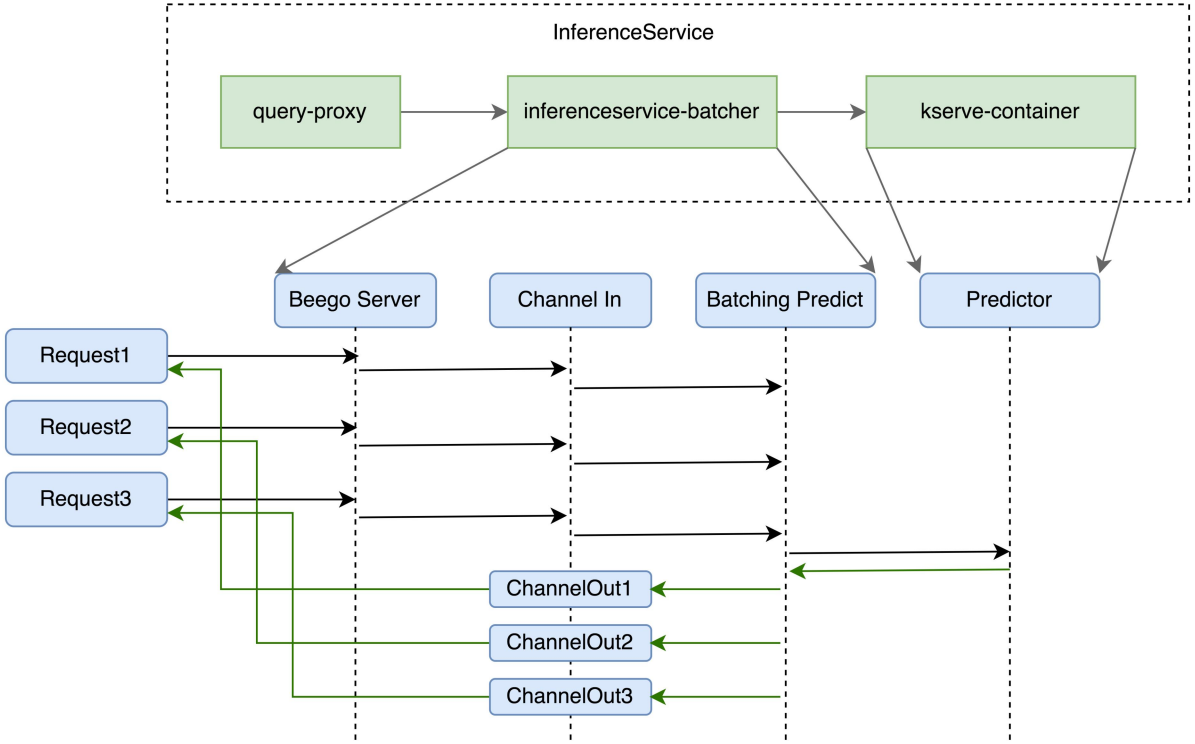
- 通过数据复用，多个任务能够共享数据缓存，避免重复拉取同一份数据，减少网络消耗



挑战二：高并发场景下，推理存在延迟，影响应用的响应时间

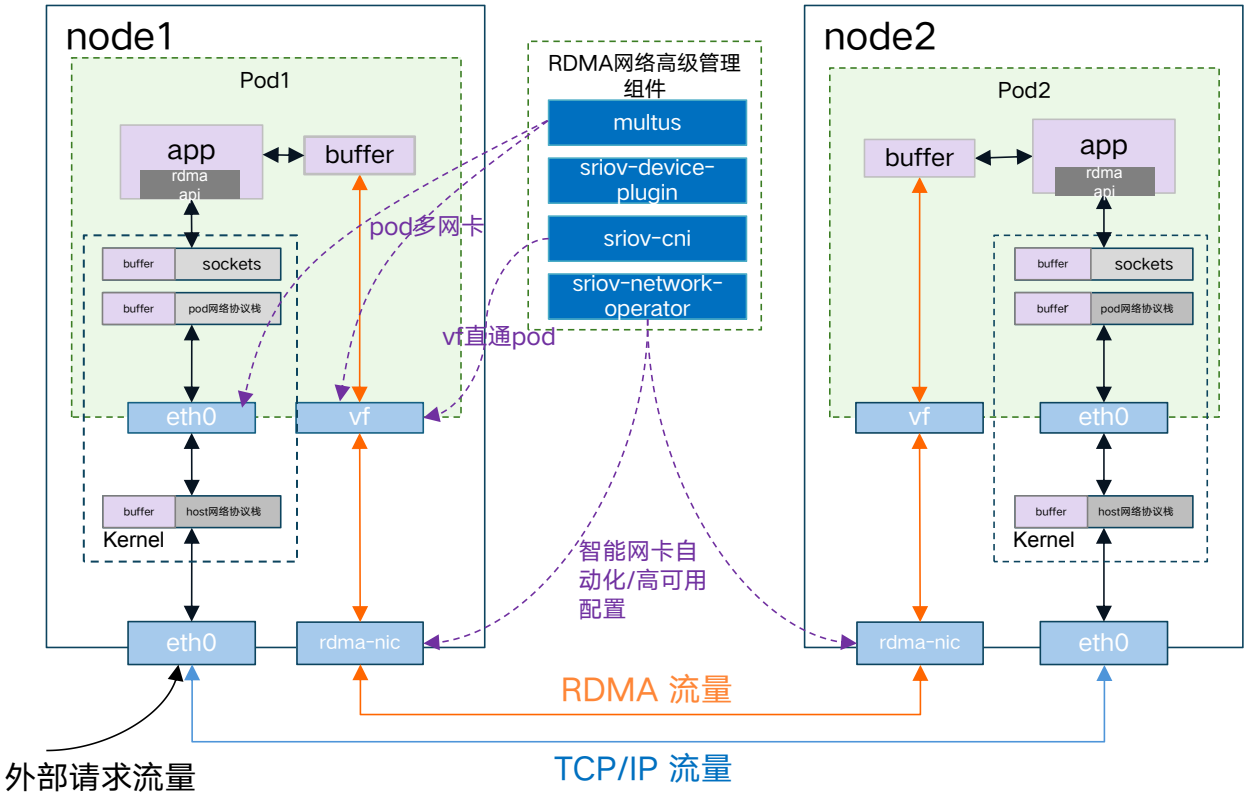
- 自适应批处理：将多个推理请求组合成单个批处理，从而优化吞吐量和延迟

- 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

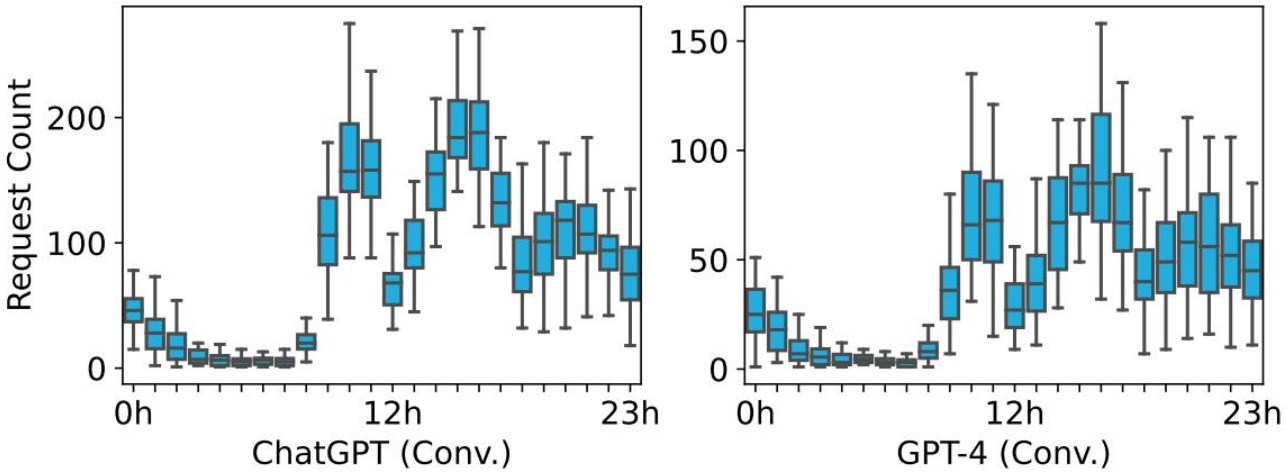
- 基于 SR-IOV 和容器多网卡技术，为容器提供 RDMA 和标准 K8s 网络的双重能力
- 通过 RDMA 高性能通信技术，加速模型推理中的高速 KV 缓存迁移，避免传统网络协议栈的高开销



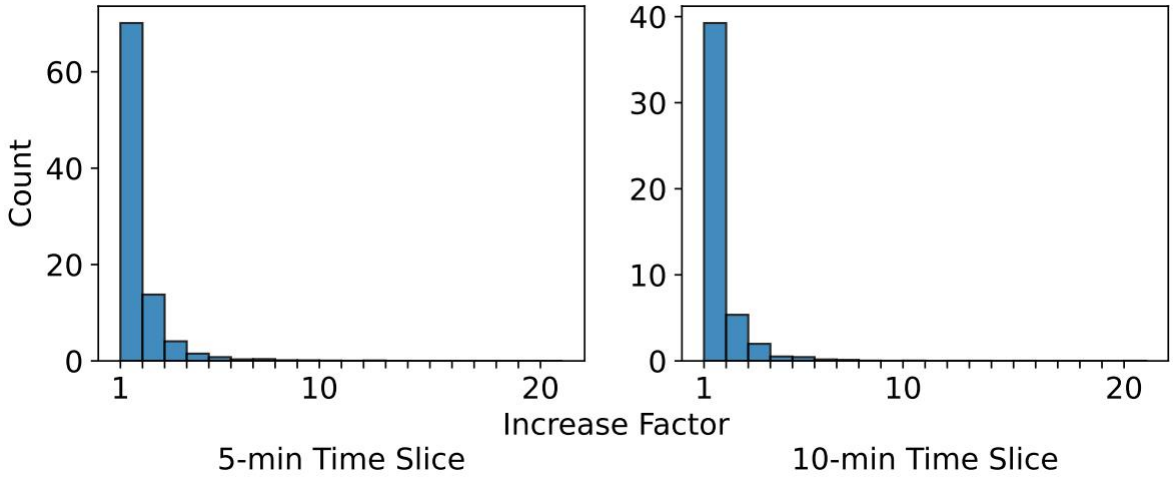

KubeCon
CloudNativeCon
THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
AI_dev
Open Source GenAI & ML Summit
China 2024
Accelerating Serverless AI Large Model Inference with Functionalized Scheduling

Real-World Workload

- **Highly Dynamic Workload**
  - High fluctuation: peak-to-valley ratio > **200x**
  - High burstiness: request volume doubling occur **65x** within 5 minutes



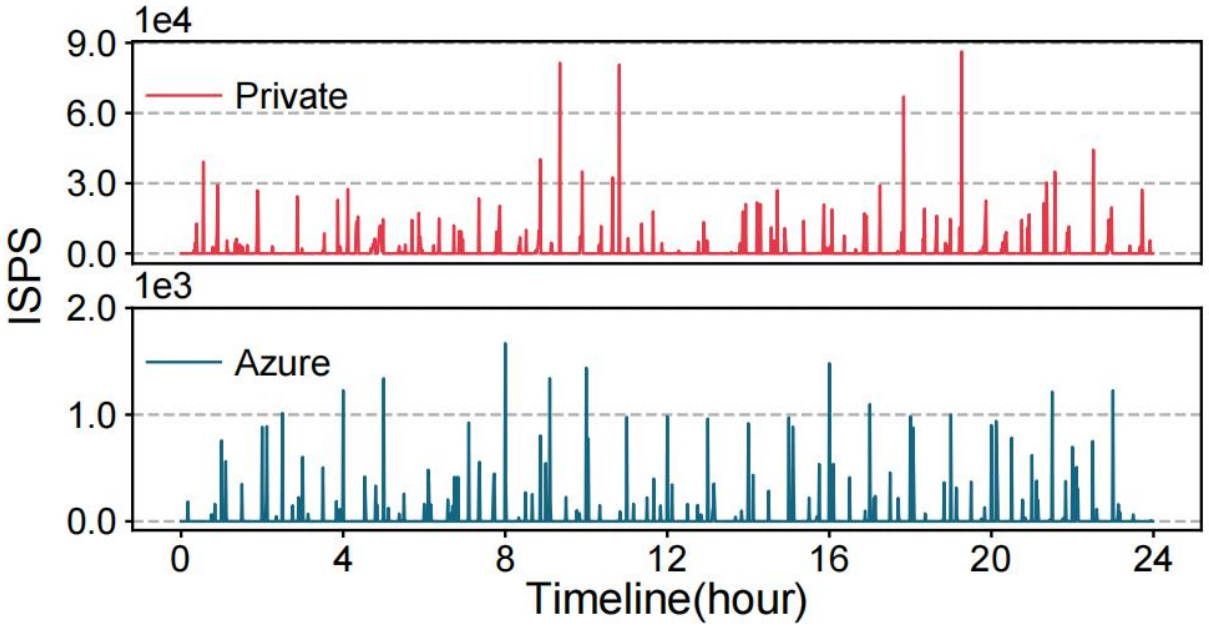
Daily Periodicity Conversation Services



Frequency of Short-term Burstiness

Real-World Workload

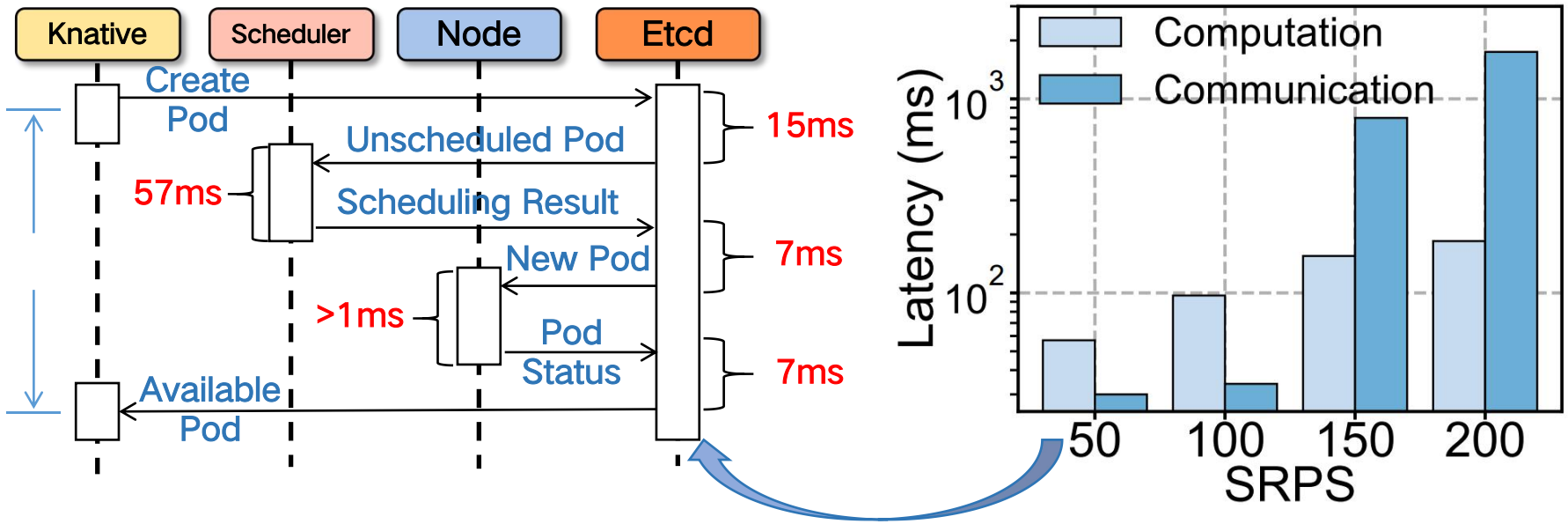
- **Sparsity of Scaling in Serverless**
  - No scaling for over **90.1%** of the time in Azure Function
  - No scaling for over **86.0%** of the time in the private system



The number of scaling request per second throughout the day

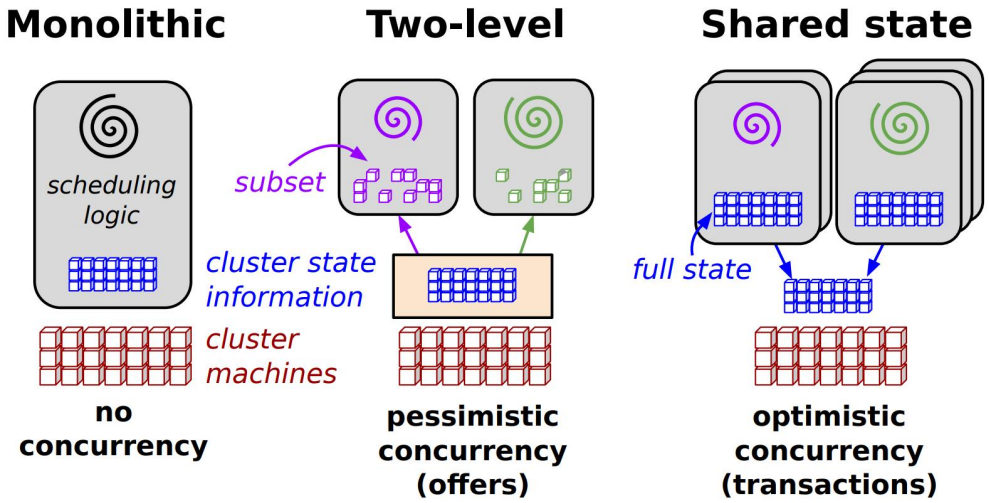
Control Plane

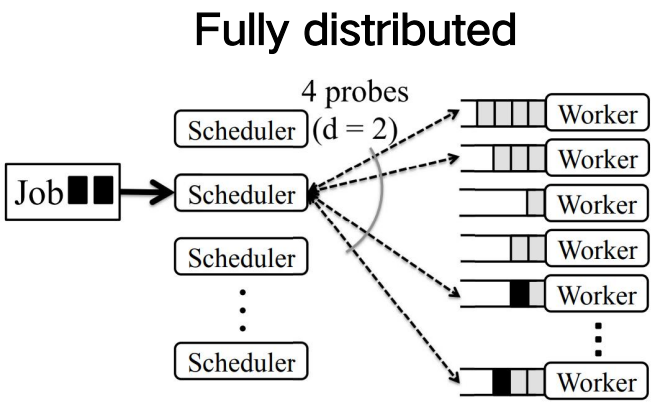
- **Centralized Architecture**
  - Cannot meet the **scalability** request of scheduling
  - **Limited** scheduling throughput

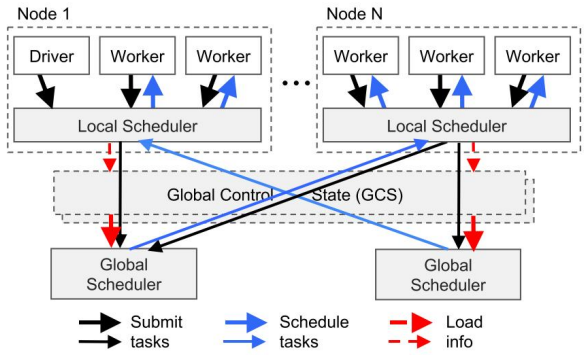


Existing Works

- **Distributed Schedulers**
  - Two-level: Mesos (NSDI 11), Yarn (SoCC 13)
  - Shared state: Omega (EuroSys 13)
  - Fully distributed: Sparrow (SOSP 13)
  - Bottom-up: Ray (OSDI 18)





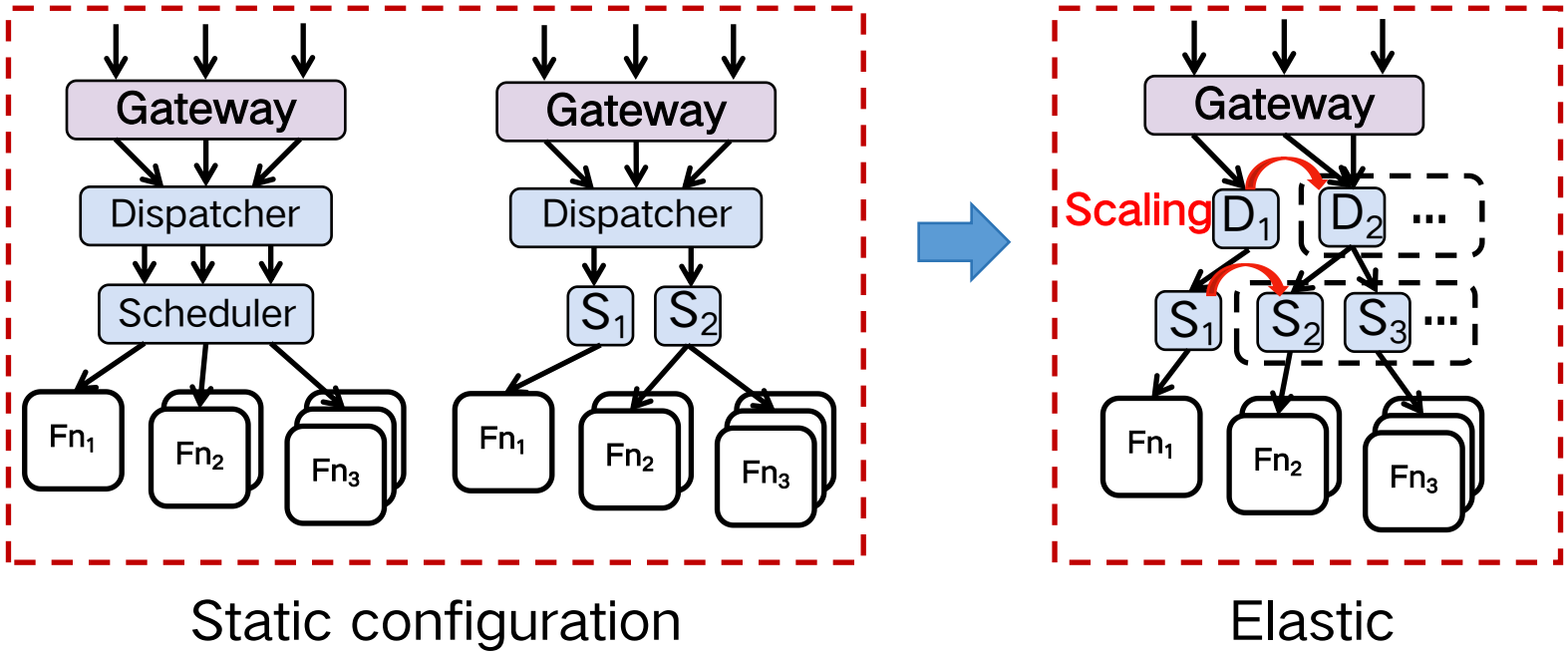


Existing Works

- **Limitations**
  - Only solved the bottleneck of centralized scheduling
    - the **bottleneck** can easily **shift** to other scheduling components
  - Inefficient scheduling capability when resources are scarce
    - **scheduling conflicts and failures**
  - Over-provisioning or throughput bottlenecks
    - **static configuration** of control plane is not suitable for fluctuating loads

Our Solution

- **Functionalized Control Plane**
  - Decouple the control plane into functions and dynamically adjust the number of instances for each function based on the request load

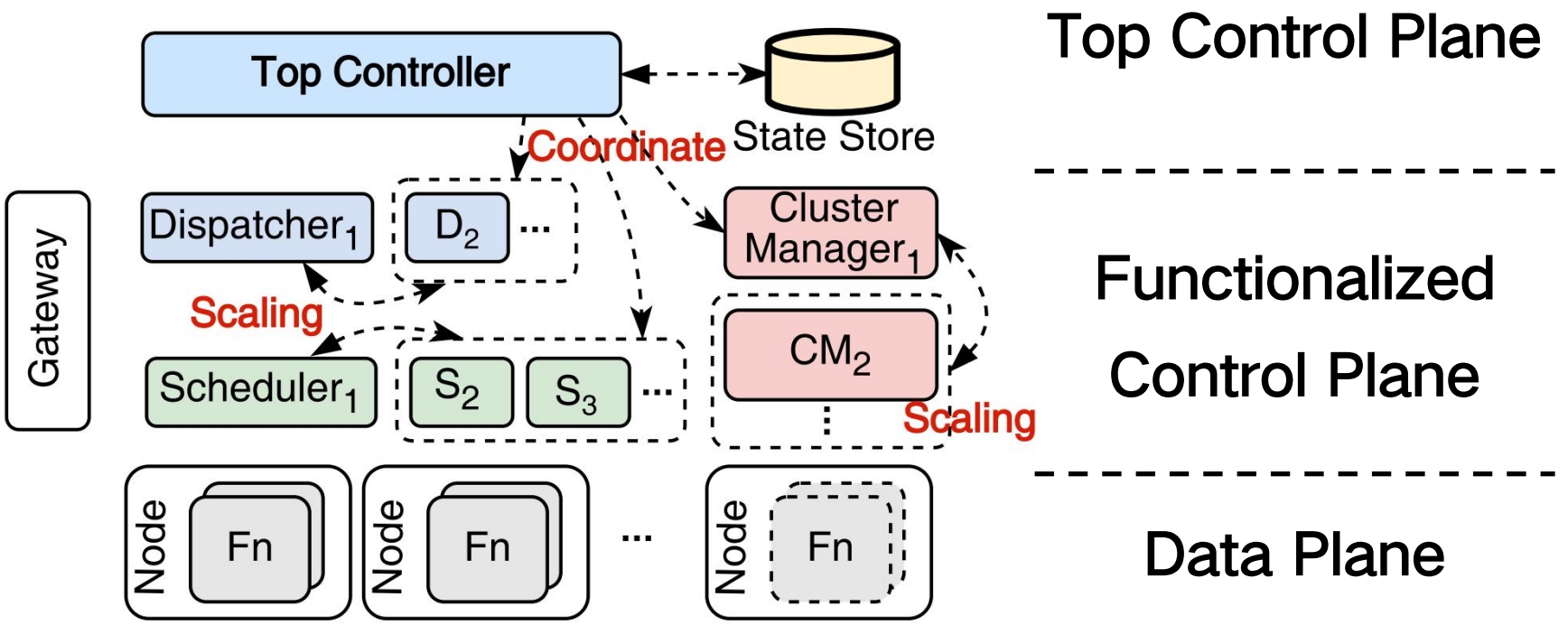


Challenges

- How to **decouple** the control plane into functions?
- How to ensure **fast scheduling** of function instances?
- How to **scale** each control function based on load?

Our Solution

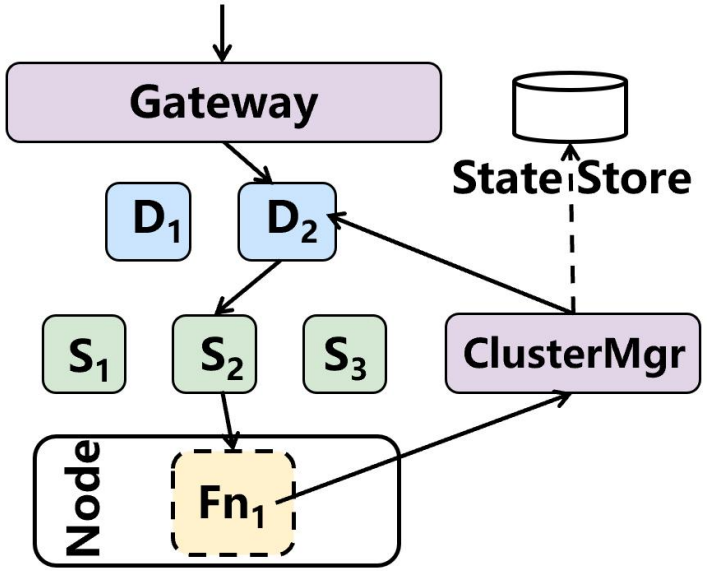
- **Elastic Serverless Control Plane**



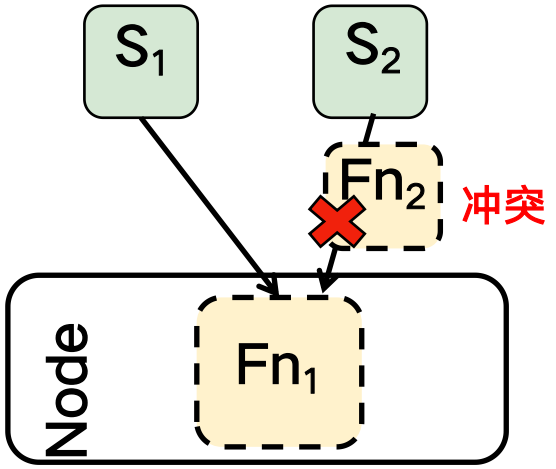
- **Control Functions**

Component	Definition	Description
Dispatch Func	func Dispatch(request)->InstanceAddr	Dispatch requests to available instances
	func ModInstance(funcName,address) ->StatusCode	
	func Coornadite(actions)->StatusCode	
Schedule Func	func Schedule(funcConstarints)->StatusCode	Select a node for the newly launched instance
	func ModNode(nodeStatus,address) ->StatusCode	
	func Coornadite(actions)->StatusCode	
ClusterMgr Func	func Update(nodeStatus)->StatusCode	Synchronize node status

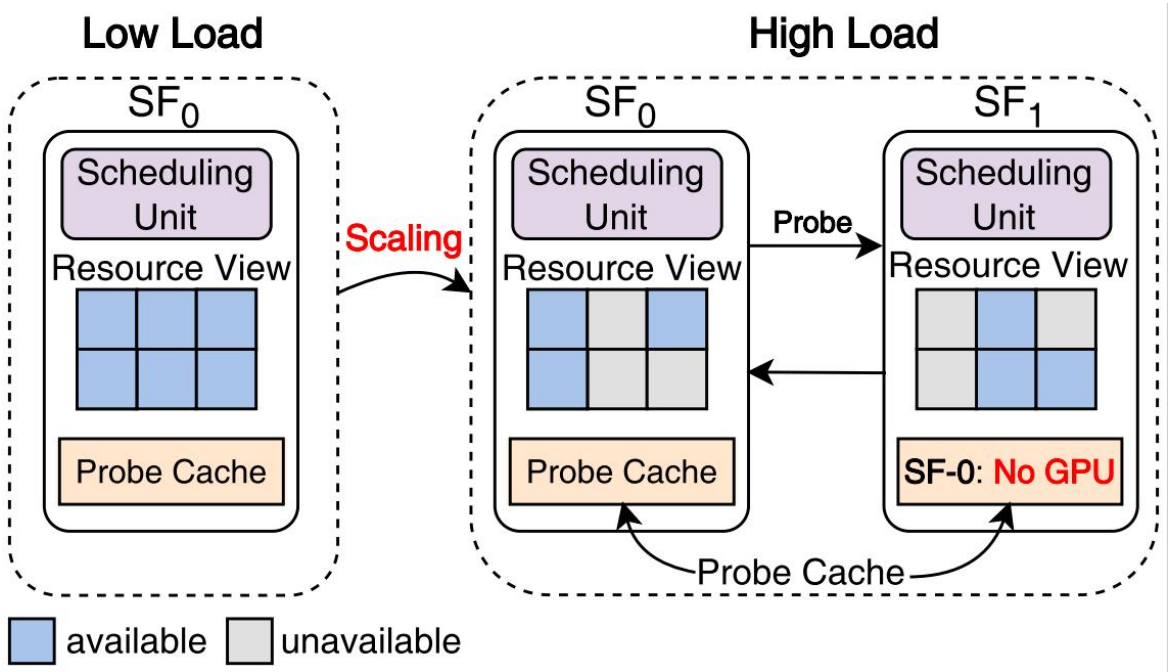
- **Control Functions**
  - Computational bottleneck
    - each control function can independently scale to improve throughput
  - Communication bottleneck
    - control functions maintain the addresses of downstream functions to guarantee direct connections
    - remove the state store from the critical path of state synchronization



- **Efficient Scheduling**
  - Multiple schedulers can cause a large number of conflicts
  - Handling conflicts
    - introduce centralized component (e.g., Ray) can result in significant scheduling overhead
    - re-probe at node level (e.g., Sparrow) can lead to a large number of re-scheduling, still limiting the throughput

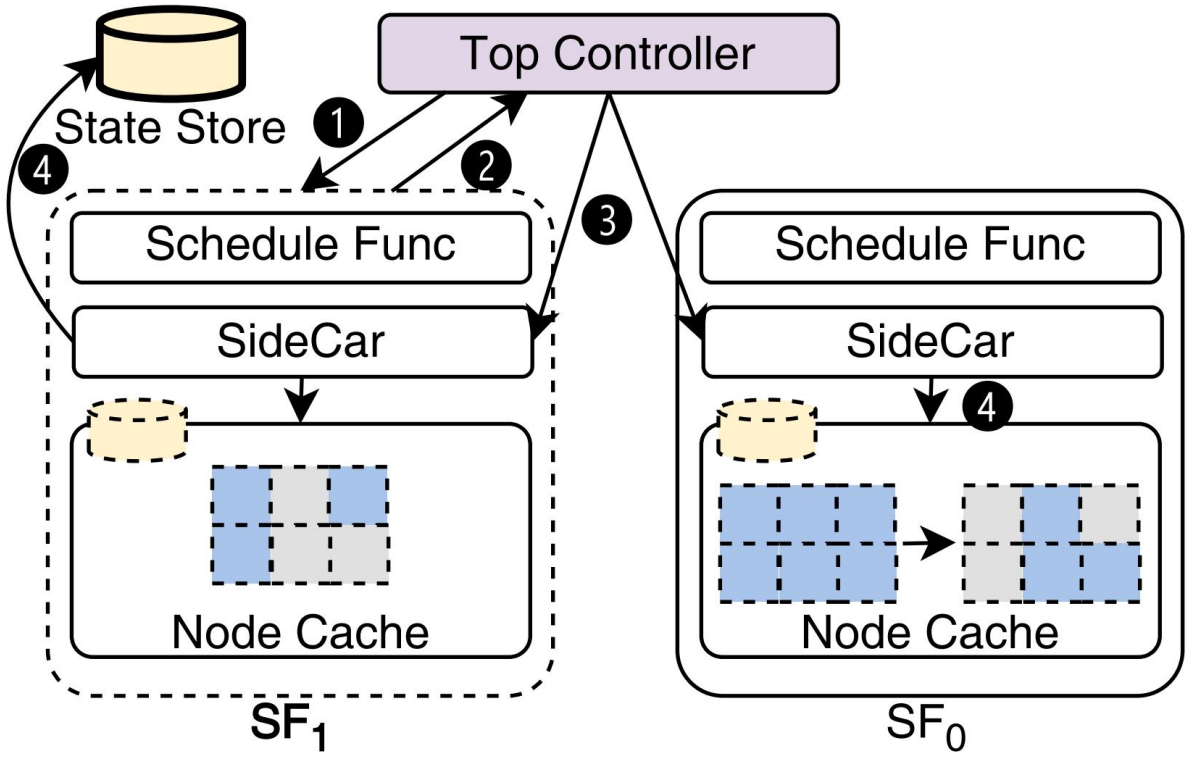


- **Efficient Scheduling**
  - Dynamic partitioning and probe

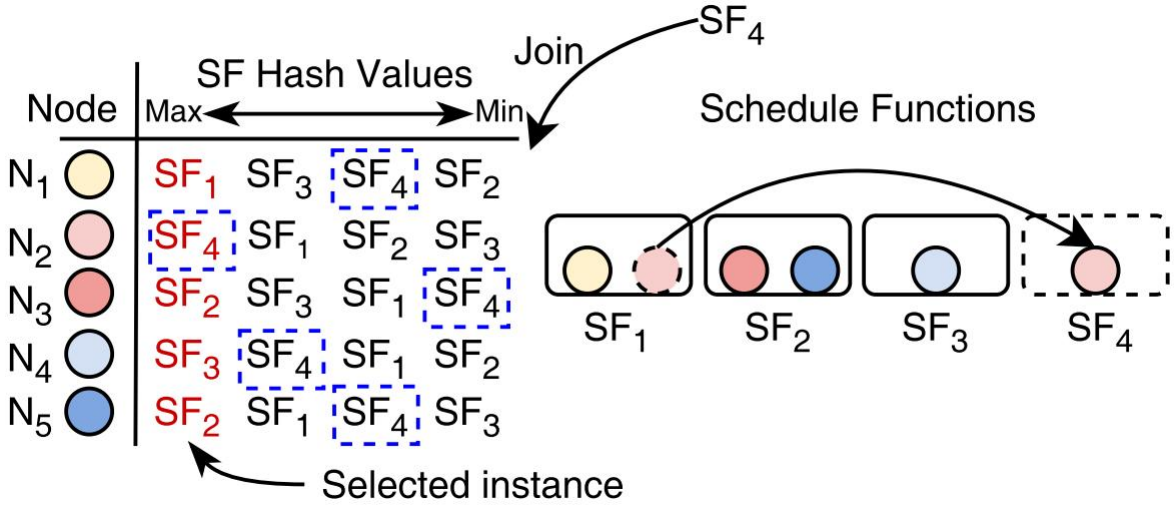


- **Scaling Control Plane**
  - Metric
    - Scaling Request Per Second (SRPS)
    - The max throughput of each control function: T_{DF}, T_{SF}, T_{CMF}
  - Decision
    - Dispatch Func instance periodically reports RPS to the top controller
    - Top controller determines the target number of instances for each control function based on the maximum throughput
    - Unified decision-making to avoid redundant scaling

- **Scaling Control Plane**
  - Scaling process
    - select available nodes for new CF instances and initiating
    - register with top controller
    - update instances partitioning
    - fetch state cache from state store



- **Scaling Control Plane**
  - Reallocating partitions
    - partition balance
    - minimal partition migration
  - Rendezvous hashing algorithm
    - hash all key-server (e.g., Node-SF) combinations with a hash function
    - assign the key to the server with the largest hash value
    - maintain the “first choice” invariant when adding and removing servers

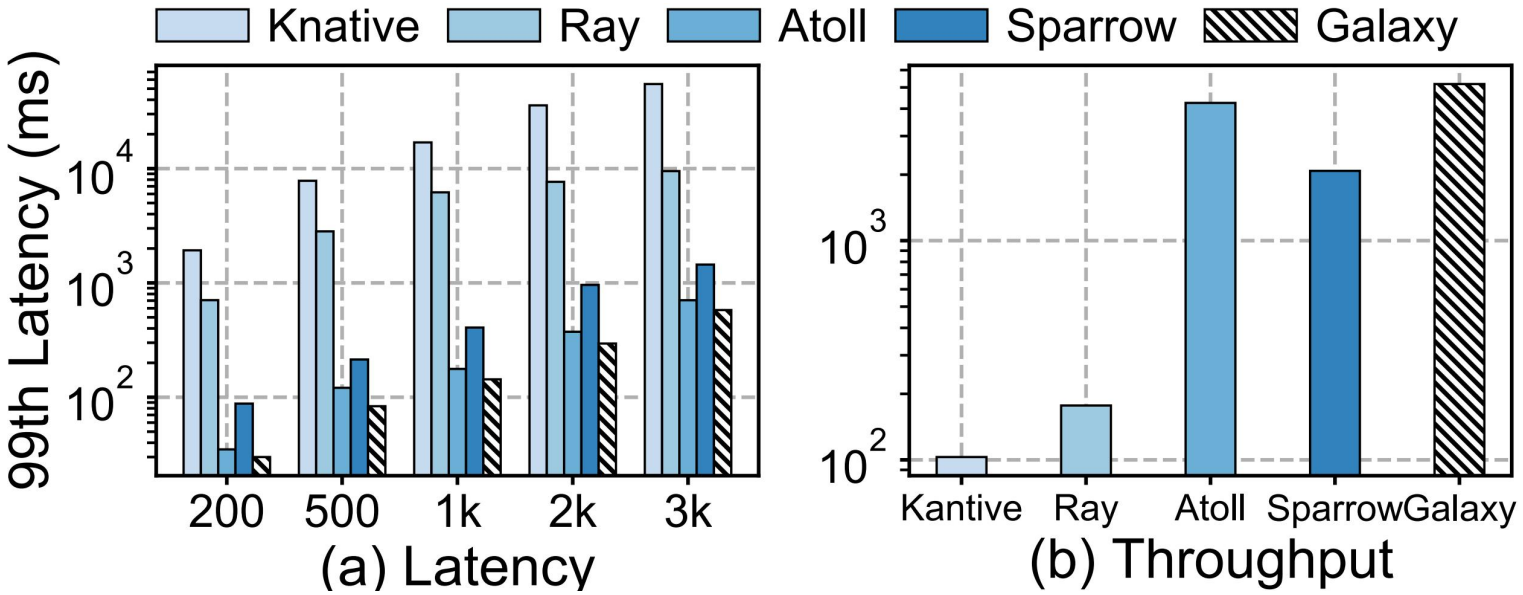


Evaluation

- **Setup**
  - 210 nodes cluster
  - Comparison systems
    - Knative: centralized architecture
    - Sparrow: fully distributed scheduler
    - Ray: bottom-up scheduler
    - Atoll: two-level scheduler

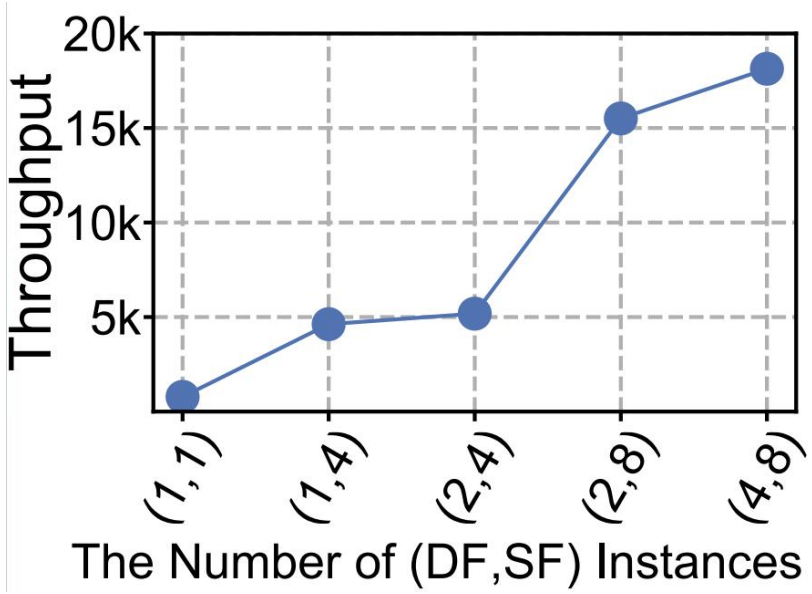
Evaluation

- **Overall Performance**
  - Scheduling latency: reducing 87.9%~95.3%
  - Throughput: increasing by 1.2x~29.3x



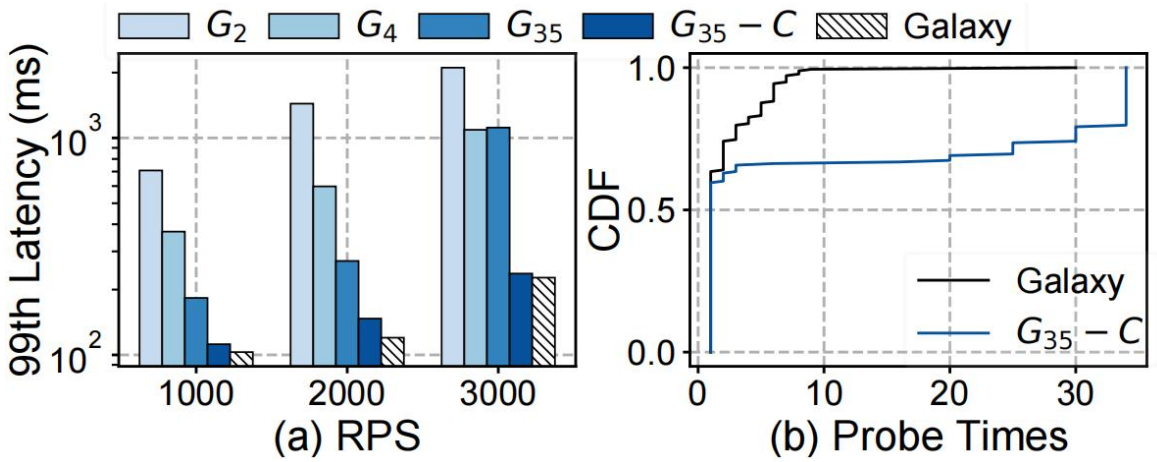
Evaluation

- **Scalability**
  - Scheduling 20,000 concurrent instances within 1s
  - Scaling instances for a single control function leads to limited improvement



Evaluation

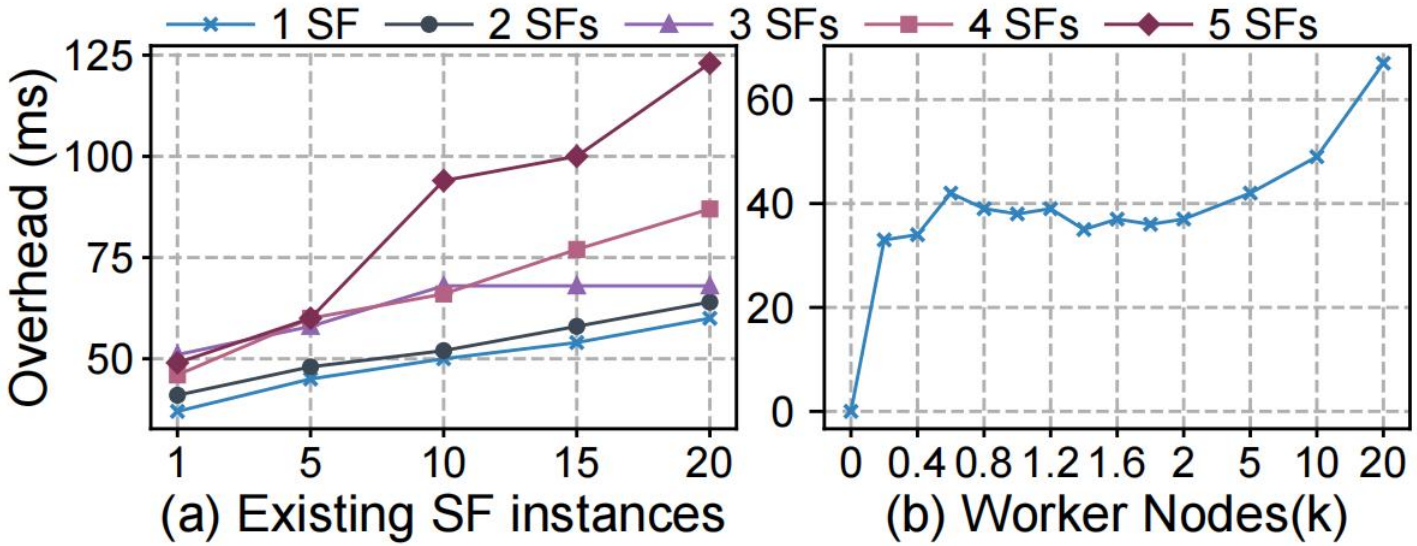
- **Improvement Analysis**
  - Scaling control functions effectively reduces latency
  - Connection caching can further reduce the overhead by 78.7%
  - Probe caching can reduce the probe overhead by 75.9%



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 $G_{35}-C$

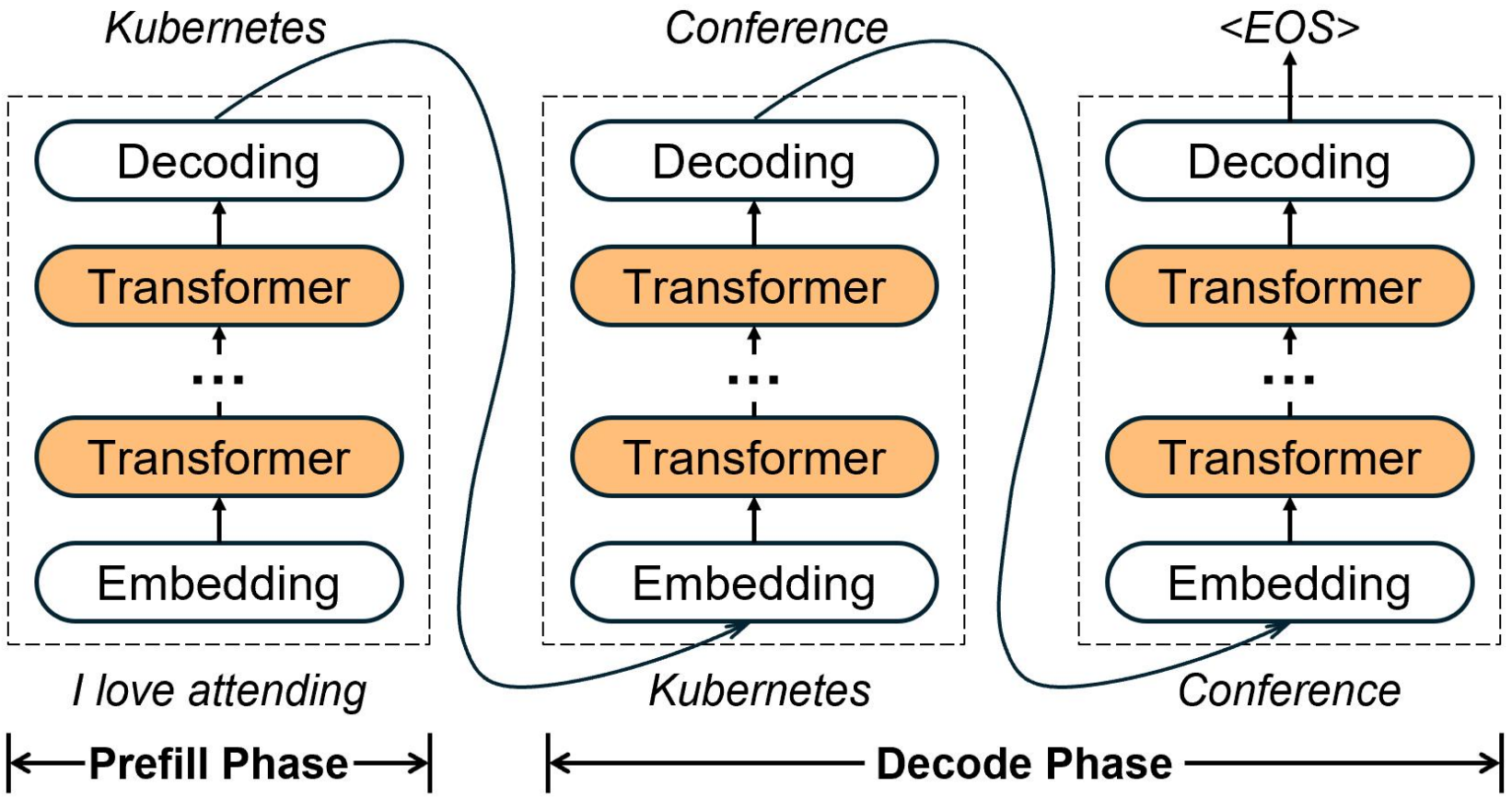
Evaluation

- **Overhead**
  - Main factors: existing instances, new instances and cluster size
  - Reallocation overhead: > 30ms



Large Language Model

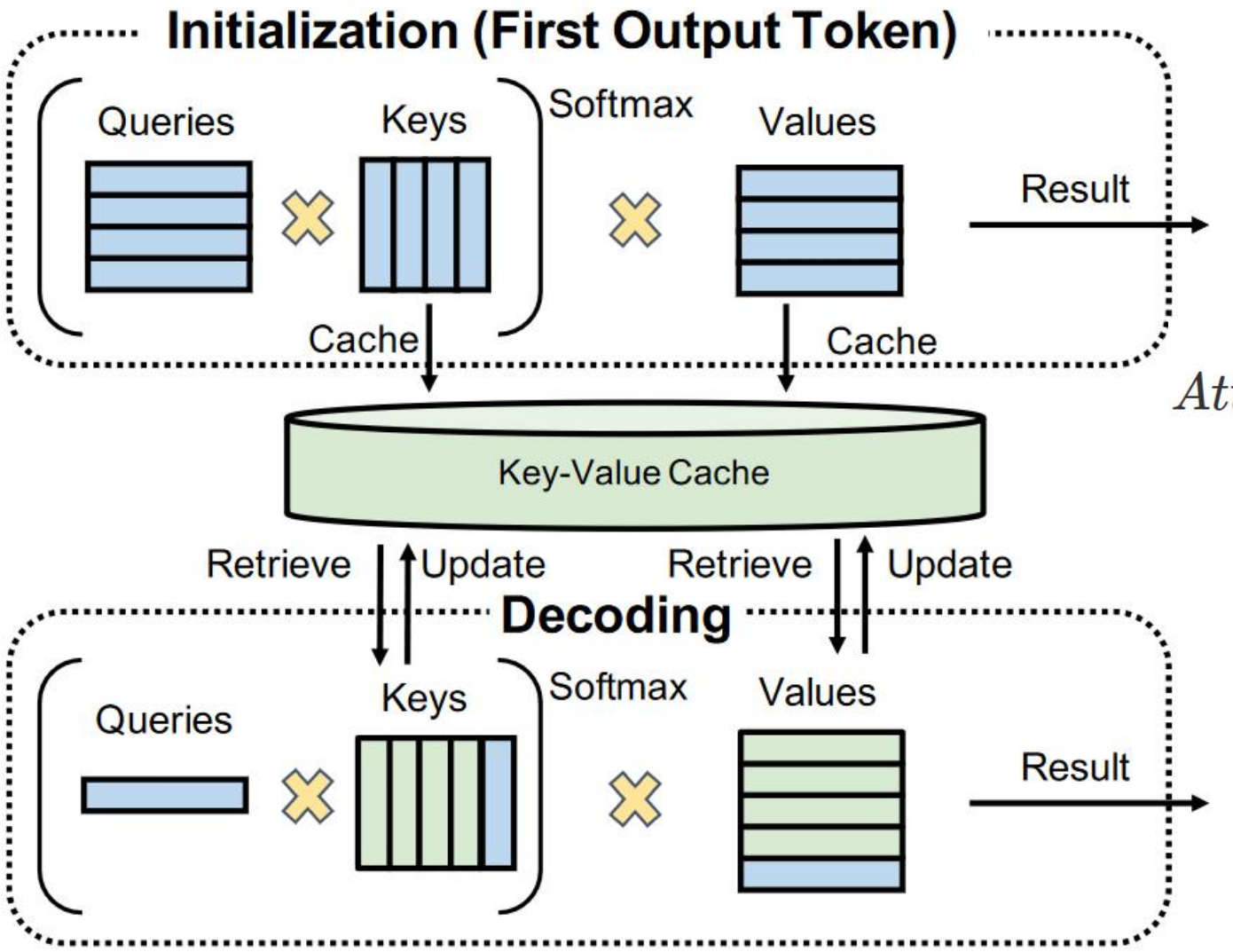
- **Auto-regressive Model**



Prefill & Decode

- **Prefill**
  - the initial step in LLM inference
  - process the input tokens in parallel within a single iteration to generate the first new token
  - compute-bound
- **Decode**
  - generate output tokens autoregressively one at a time
  - each sequential output token needs to know all the previous iterations' output states
  - memory bound

KV Cache



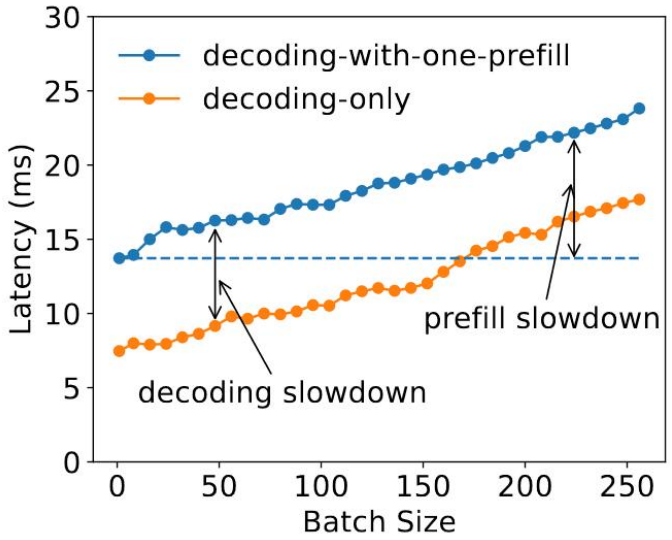
- **Self-Attention**

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

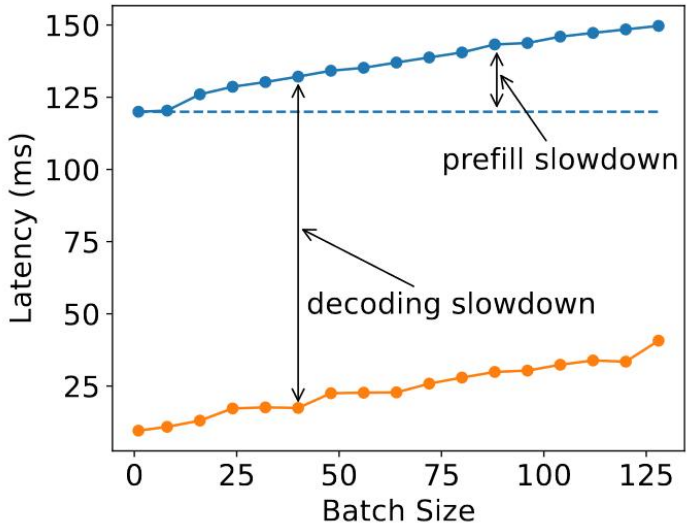
- **KV cache**
  - newly calculated tensor
  - reused tensor

Prefill & Decode

- **Prefill-decoding interference**
  - adding a single prefill job to a batch of decoding requests significantly slows down both processes



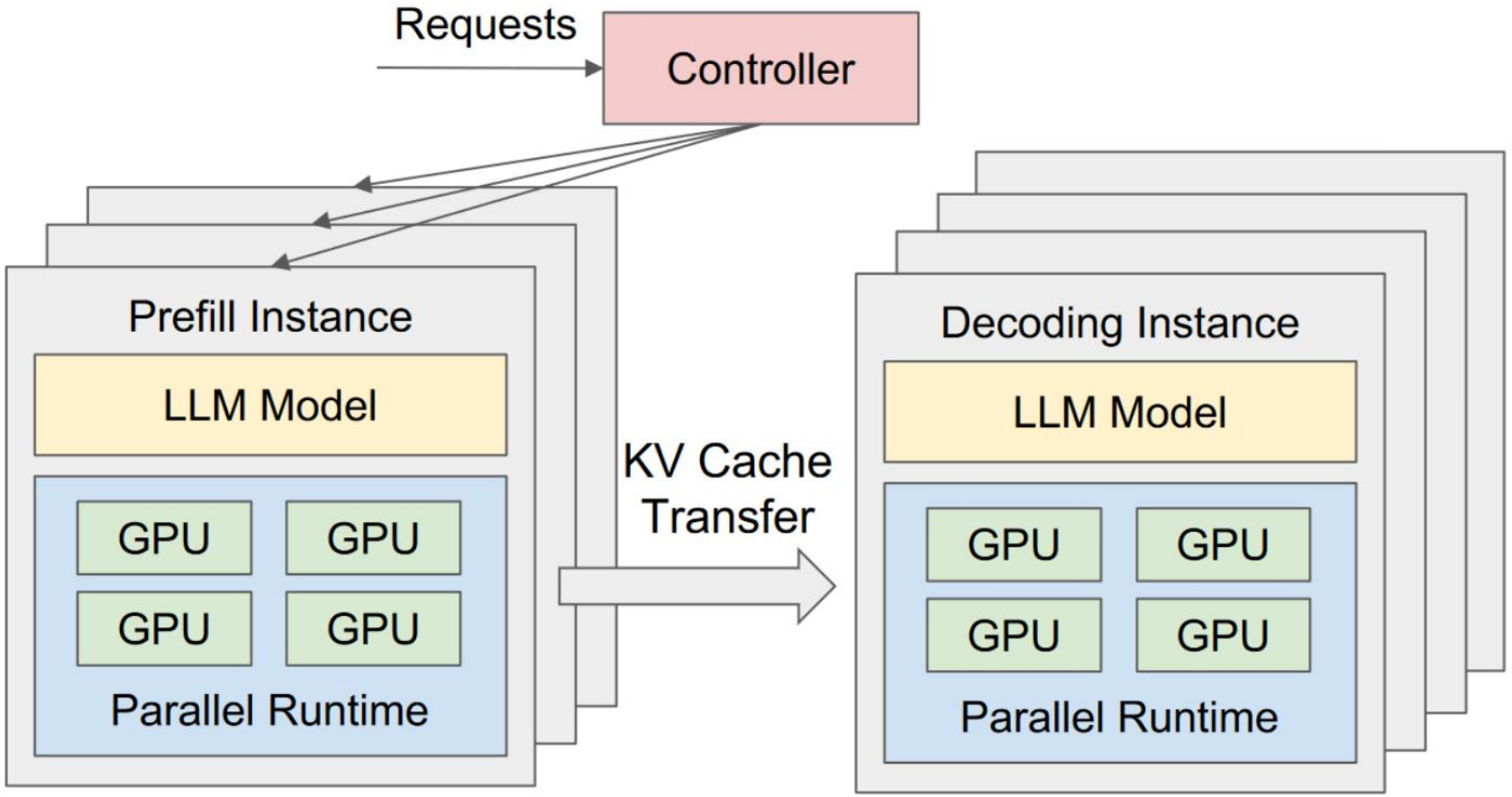
(a) Input length = 128



(b) Input length = 1024

Prefill & Decode

- **Disaggregating Prefill and Decode**

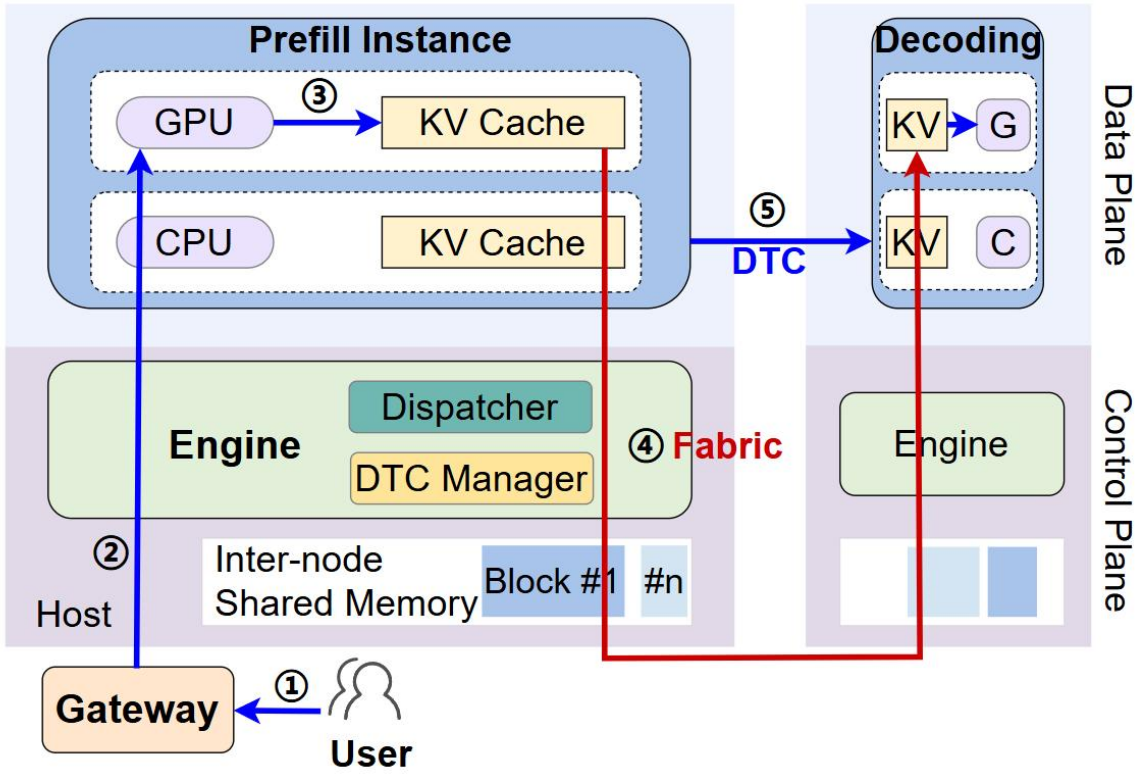


- **Notable Overhead of Transferring KV Cache**
  - The size of KV cache increases linearly with context length
  - Even exceeding model size under long context

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

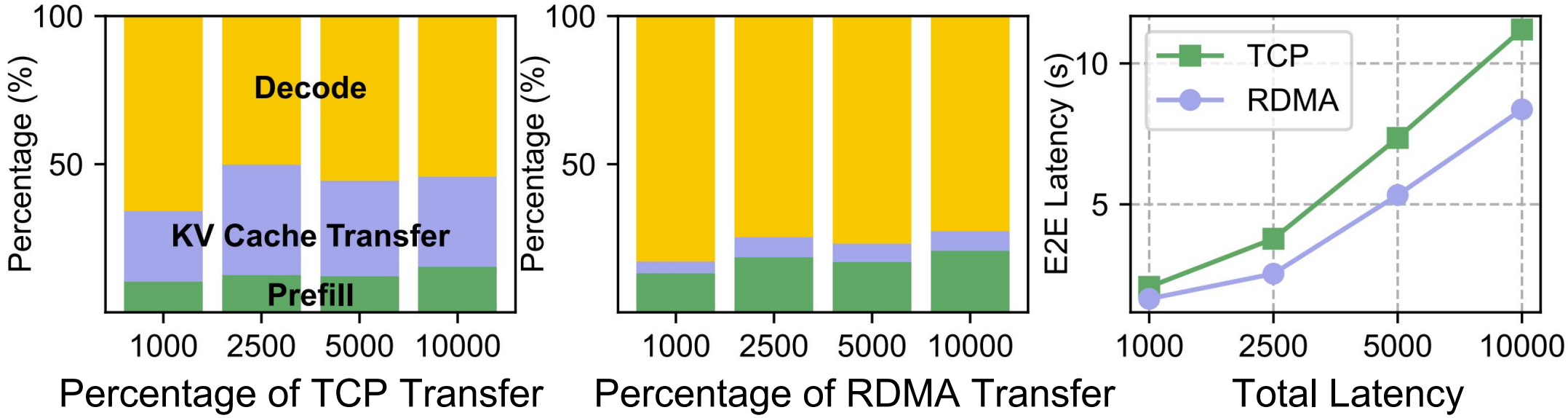
LLaMA2-13B, KV Cache size with context length

- **KV Cache Transfer**
  - Engine
    - data forwarding
    - DTC establishment
    - transfer method selection
  - Direct Transfer Channel
    - full-duplex connection using RDMA
    - avoiding redundant copies
    - local GPU → local CPU → remote CPU → remote GPU



Evaluation

- **Performance of Llama-3-70B with various token size**
  - Transfer: speed up **6x~8.1x**, time percentage **4%~6.8**
  - Total latency: speed up **20.5%-32.9%**



Conclusion

- Problem: Significant overhead of scheduling and KV Cache transfer in LLM inference with Serverless
- Solution: Functionalized control plane and RDMA-based direct transfer
- Result: Increase scheduling throughput by 29x, speed up KV Cache transfer by 8.1x