云计算技术与工程

容器管理技术

徐敏贤

副研究员
云计算研究中心
中国科学院深圳先进技术研究院
http://www.minxianxu.info/vcc

云想衣裳花想容，春风拂槛露华浓。若非群玉山头见，会向瑶台月下逢。
——（唐）李白

内容回顾

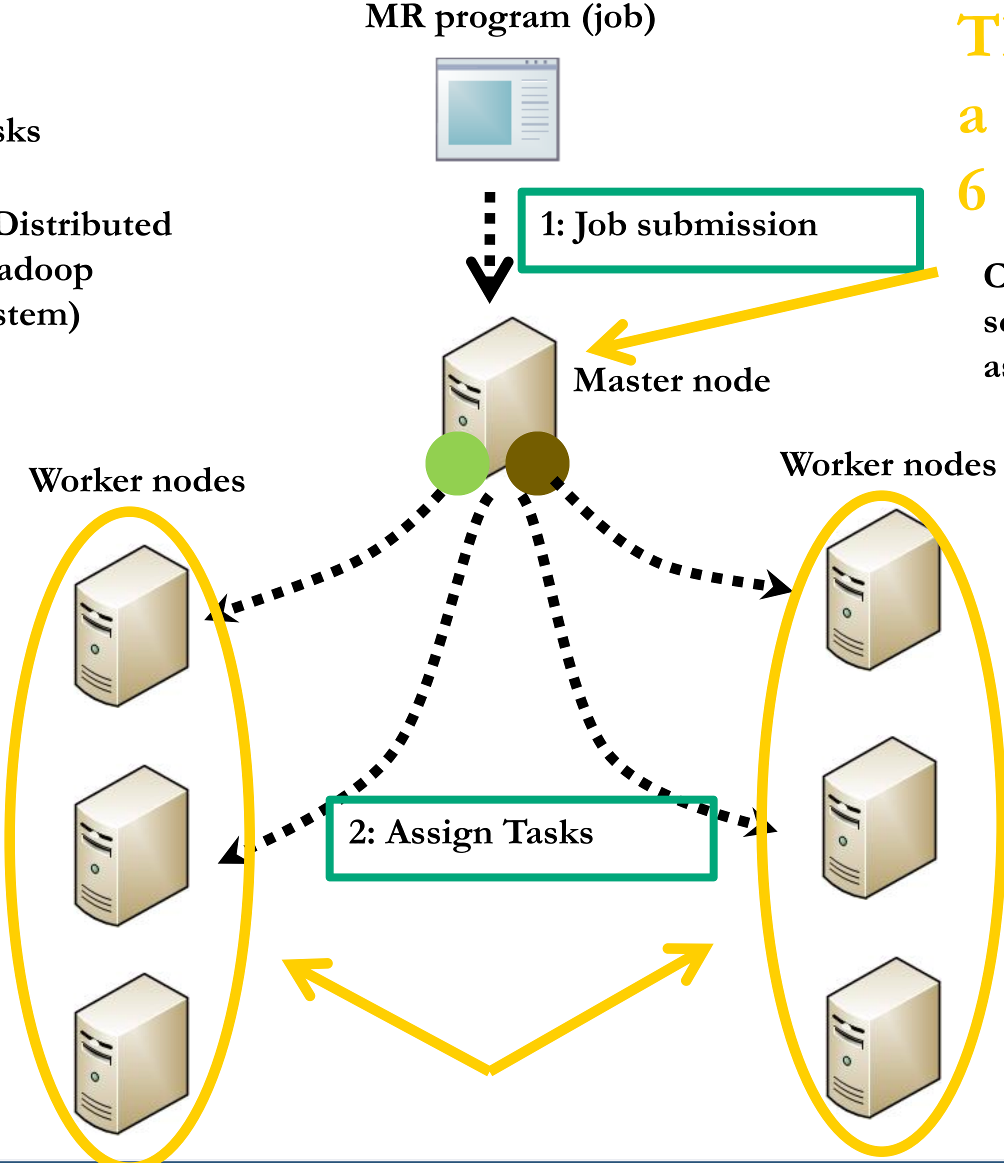
- Q1: MapReduce过程主要包含哪6个步骤?

The execution of a MR job involves 6 steps

Map Tasks

Reduce Tasks

Data is stored in a Distributed File System (e.g. Hadoop Distributed File System)



Control level work, e.g. job scheduling and task assignment

内容回顾

- Q2: MapReduce有哪些有点和缺点?

- 优点:
  - 处理海量数据（>1TB）
  - 上百/上千CPU实现并行处理
  - 移动计算比移动数据更划算
  - 自动实现分布式并行计算
  - 模型抽象简洁，程序员易用
  - 提供状态监控工具
  - 容错性
- 缺点
  - 扩展性受限
  - 单点失效
  - 重启困难

内容回顾

- Q3: Yarn相比MapReduce，做了哪些改进?

YARN基本设计思想是将原MapReduce架构中JobTracker的两个主要功能，即资源管理和作业调度/监控分成两个独立组件，即全局的ResourceManager和与每个应用相关的ApplicationMaster。

- 将JobTracker的两个主要功能分开
  - Global Resource Manager – 集群资源管理
  - Application Master – 作业调度与监控(one per application).
- Tasktracker
  - NodeManager (NM)
- YARN 维持了与现存 MapReduce 应用的兼容性.

内容回顾

- Q4: Spark在MapReduce基础上做了什么改进?

- Spark是UC Berkeley AMP lab所开源的类Hadoop MapReduce 的通用的并行计算框架；
- Spark 基于 map-reduce 算法实现的分布式计算，拥有 Hadoop MapReduce所具有的优点，并在**数据模型上做了扩展（支持迭代、交互、流式查询），**核心思想是提供高效的数据共享抽象；
- Spark不同于MapReduce的是Job中间输出和结果可以保存在内存中，从而不再需要读写HDFS，因此Spark能更 好地适用于数据挖掘与机器学习等需要迭代的map reduce的算法。

目录 Contents

注：本章部分内容参考ANM公司2019年报告

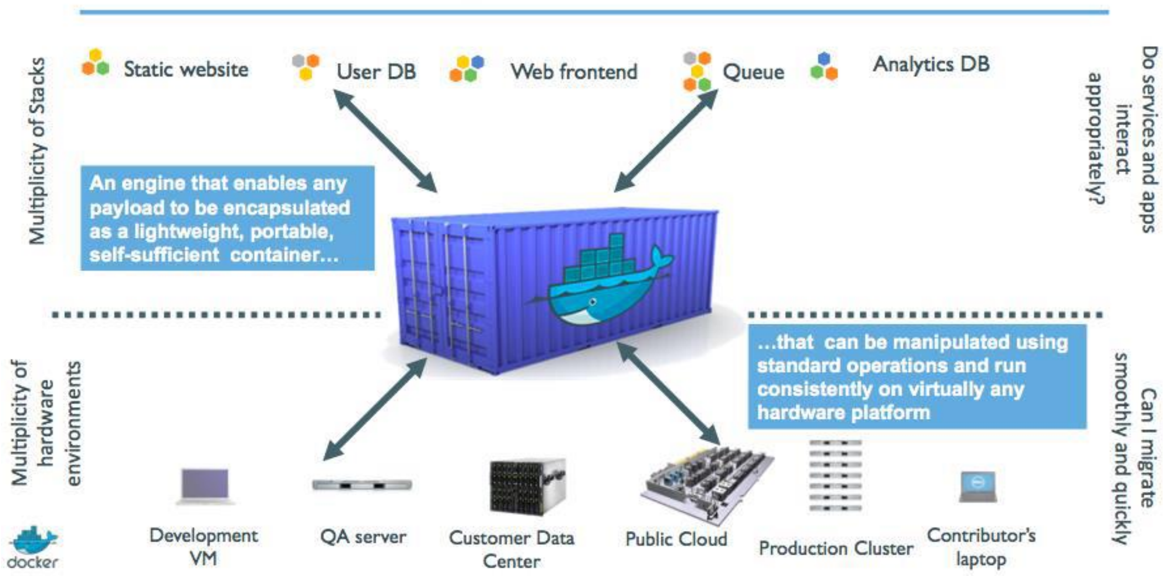
What is a container?

- Shipping containers
  - One size fits most loads
  - Massive efficiency improvements
  - Standardsize and configuration

Source -https://container.training/intro-selfpaced.yml.html#25

What is a container?

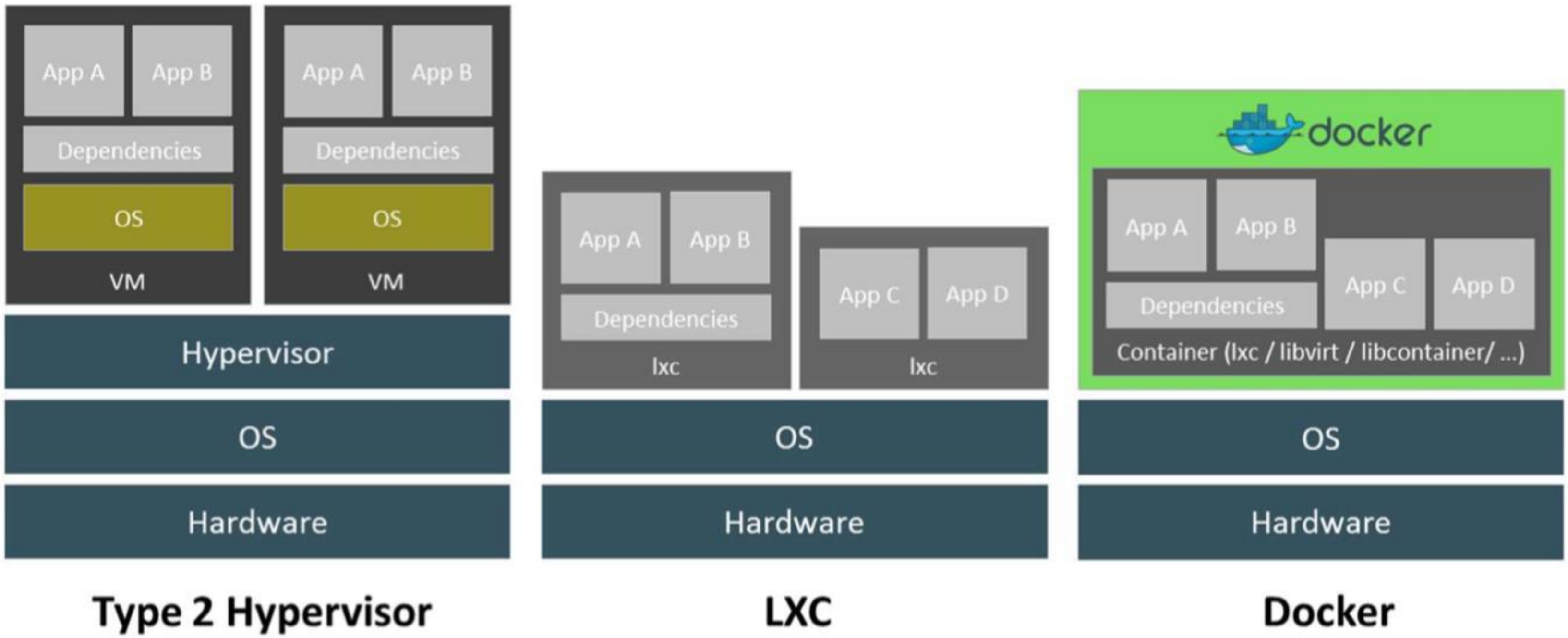
- Containers:
  - Portability for our applications
  - Standard format
  - Includes all the dependencies for the application
  - Workload isolation at runtime



Source -https://container.training/intro-selfpaced.yml.html#25

Way To Innovation

Comparison


App A
App B
Dependencies
OS
VM
App A
App B
Dependencies
OS
VM
Hypervisor
OS
Hardware
App A
App B
Dependencies
lxc
App C
App D
lxc
OS
Hardware
docker
App A
App B
Dependencies
App C
App D
Container (lxc / libvirt / libcontainer/ ...)
OS
Hardware
Type 2 Hypervisor
LXC
Docker

What is a container?

- Compared to a virtual machine containers are
  - Extremely lightweight
  - Often stateless
  - Somewhat OS agnostic
  - Run an Ubuntu flavored container on CentOS
  - Include just the binaries and libraries your app needs, run on any new enough Linux kernel

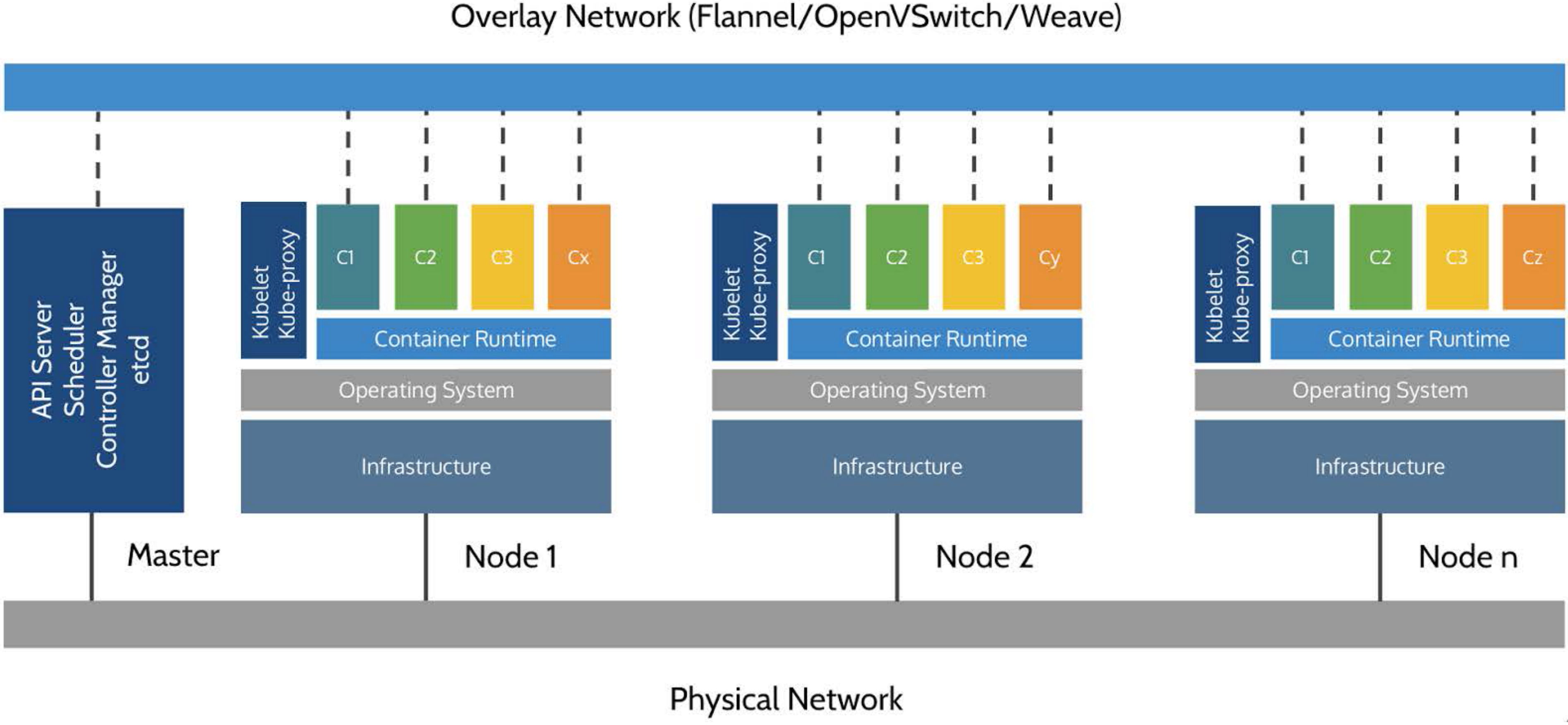
What is Docker?

- Docker
  - Made using containers easy.
  - Originally Linux centric but runs on many platforms today.
  - Includes tools for defining a container, and distributing them (the registry).
  - Has become widely used standard for building and packaging applications as containers.

What is Kubernetes?

- Open source container cluster manager
  - Used as a backend in Google's App Engine
  - Runs on Private and Public Clouds, and even on Bare metal
  - Becoming the de-facto standard for managing and orchestrating container workload clusters

Kubernetes Cluster



Key Definitions

- **Container**–lightweight standardized unit of software
- **Repository**–An online resource for hosting pre-made containers (images)
- **Cluster**–one or more nodes hosting containers
- **Orchestration**–software responsible for assigning containers to nodes in a cluster, and maintaining the health of those containers (i.e. restart dead containers)

How did we get to containers?

Sun microsystems

digital

SiliconGraphics

IBM

1950s 1960s 1970s 1980s 1990s

The explosion of computing on the campus

- As computers have become more affordable and applications have grown in complexity we've seen:
  - Application stability suffers as complexity grows.
  - Application/workload isolation becomes critical.
  - Applications often can't co-exist on the same server(Hi Java!).
  - Servers became affordable enough to dedicate them to a single application.

How did we get to containers?

1990s – 2000s

- Application isolation leads to server sprawl
  - Most servers are 75-90% idle
  - System administrators are stretched thinner and thinner
  - Developers have to wait on system admins to order servers, and install software, sometimes for months

How did we get to containers?

2000s

- Container technology is developed
  - FreeBSD jails are introduced March 2000 in FreeBSD 4.0
  - Solaris zones are introduced in 2005 with Solaris 10
  - Google supports the development of name spaces in the Linux kernel (a key component of containers)
  - Docker is founded in 2010 making Linux containers easy for developers to consume

How did we get to containers?

2010 - today

- What made Docker special?
  - Docker makes it easy for a developer to spin up a container and run it almost anywhere
  - Docker with its registry becomes the application delivery system
- Downsides to plain Docker
  - Docker doesn't address the IT operations pain points around availability and reliability

How did we get to containers?

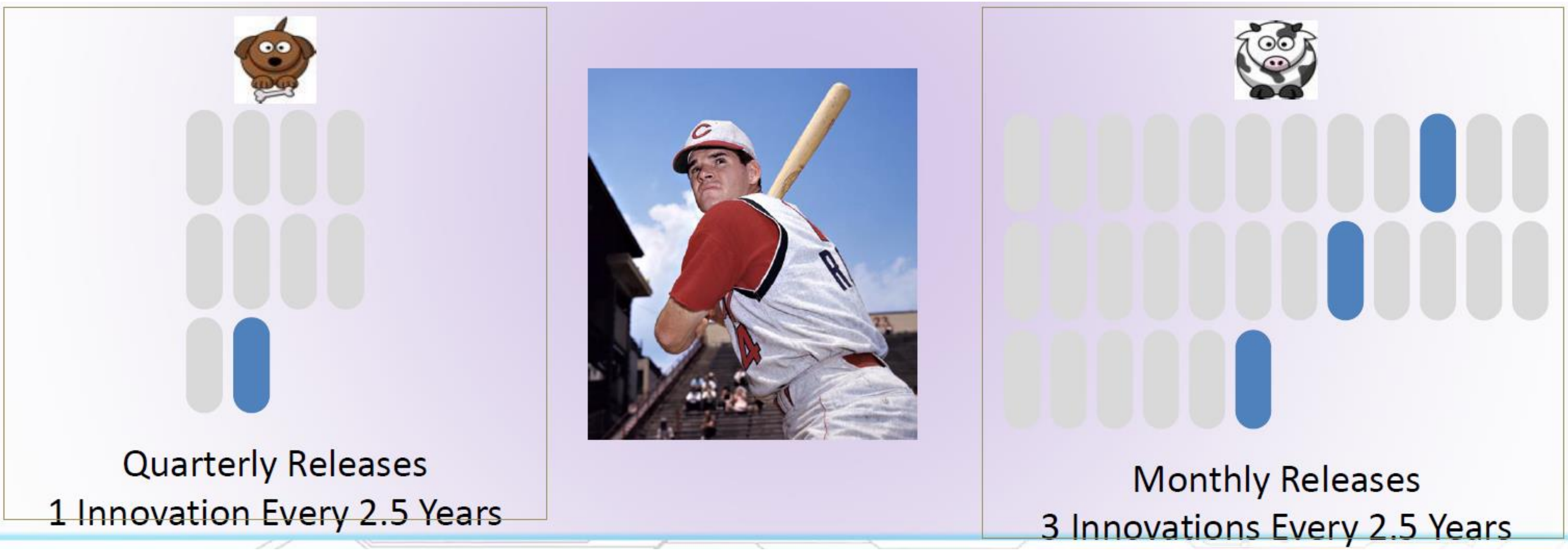
2010 - today

- Google releases Kubernetes in 2013
  - Kubernetes is an open source system based on Google’s internal systems called Borg and Omega
  - Kubernetes is a container management platform that addresses most of IT operations pain points while still providing the agility developers crave

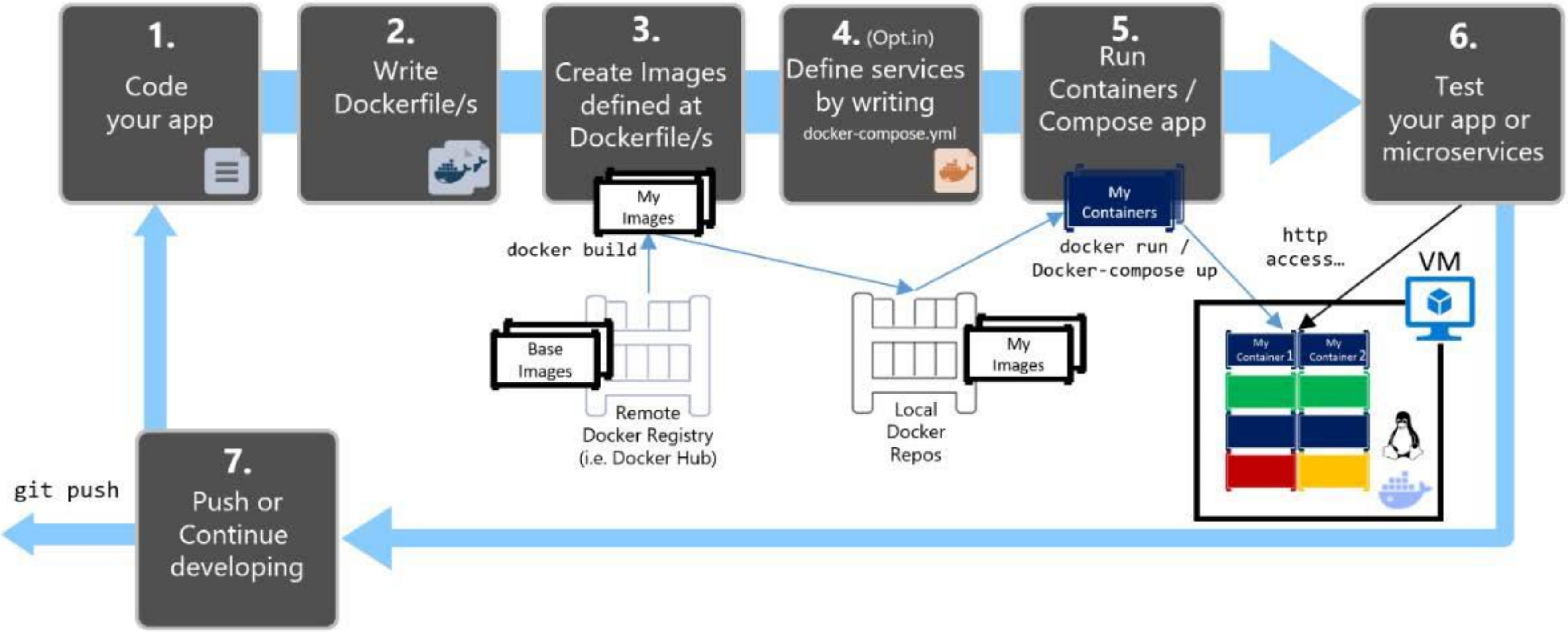
Container use cases

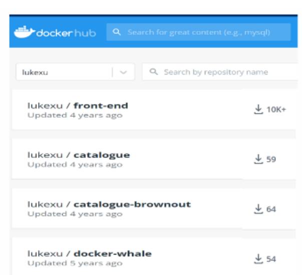
- Containers can be used in many different ways, they broadly divide into two categories:
  - Empowering developers with APIs for consuming infrastructure.
  - Software packaging and distribution.
- This has become the standard application design in the devops/cloud native world. Microservice architectures being the prime example of this design.

- Empowering Developers
- At Bats Matter: The VC 10% Rule



Inner-Loop development workflow for Docker apps





- Containers and software distribution
  - Pre-built containers (images) are distributed via open and private registries.
  - Registries are collections of downloadable pre-built images.
  - Hub.docker.com(public)
  - Amazon Elastic Container Registry (private)
  - Google cloud container registry (private)

Container Architecture

- Containers and software distribution
  - Pre-built containers (images) are distributed via open and private registries.
  - Registries are collections of downloadable pre-built images.
  - Hub.docker.com(public)
  - Amazon Elastic Container Registry (private)
  - Google cloud container registry (private)

Container architecture –stateless design

- Kubernetes' orchestration engine provides the ability to dynamically scale your environment up and down.
- Scaling dynamically is *much harder* when there is local state in the container.

Containers and Networking

- Container Networking can refer to some very different use cases.
  - How do we network our containers together?
  - Running containers on our network infrastructure.

Networking for containers

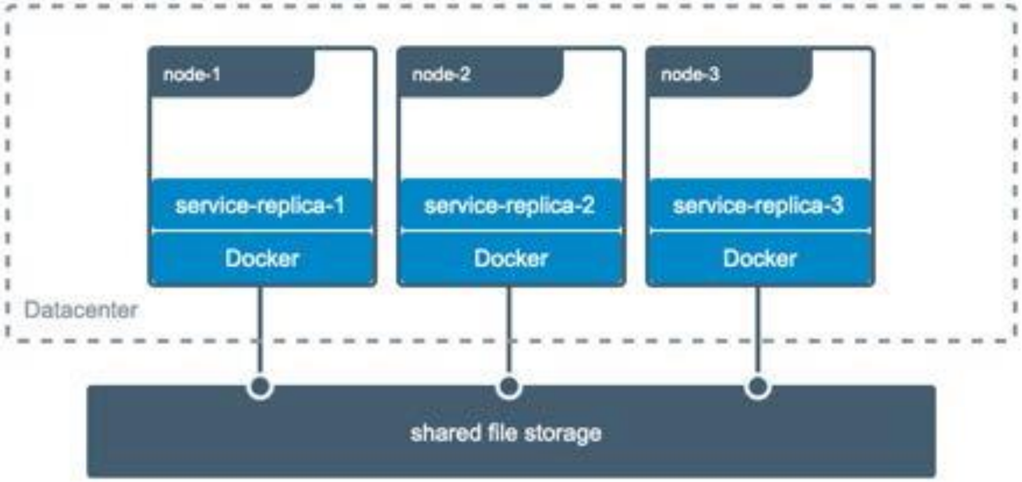
- Do we need to share a layer 2 segment between containers on different nodes?
  - VXLAN overlays
- Do we need IP routing?
  - BGP, OSPF, static
- Do we need to provide IP services for containers?
  - Load balancing, NAT, DNS, DHCP
- Both open source and vendor solutions are available to enable advanced networking capabilities.
  - E.g. Contiv, Weave, Flannel, Vmware NSX-T, Cisco ACI, etc.

Containers on switches

- Linux based switch operating systems with the ability to run containers as a service:
  - Cisco IOS XE
  - Arista EOS
  - Cumulus Linux
- Example use cases
  - Performance monitoring (PerfSONAR)
  - Configuration management (Puppet agent)
  - Troubleshooting tools (tcpdump, Wireshark)

Containers and Storage

- Containers were originally stateless.
- Volumes were added afterwards to add stateful storage.
- Docker and Kubernetes both support volumes.
- There are many different kinds of volumes to chose from.
- Not all volumes can be shared with multiple containers/pods.
- There are multiple opensource and commercial solutions available.
  - E.g. Cisco Hyperflex HCI integration, NetApp, Amazon S3 integration, native open source.



An example of commercial Kubernetes

Kubernetes-as-a-Service

Setup

- Deploy Kubernetes clusters on HyperFlex IaaS (VMware)
- Container Networking (Contiv / ACI)
- Persistent storage (Flex Driver)
- Layer-4 and Layer-7 load balancing
- High availability

Consume

- Authentication with Active Directory
- Role based access control
- Communication between containers and external VMs / BMs
- UI – Kubernetes, API
- Security (policies, encryption)

Manage

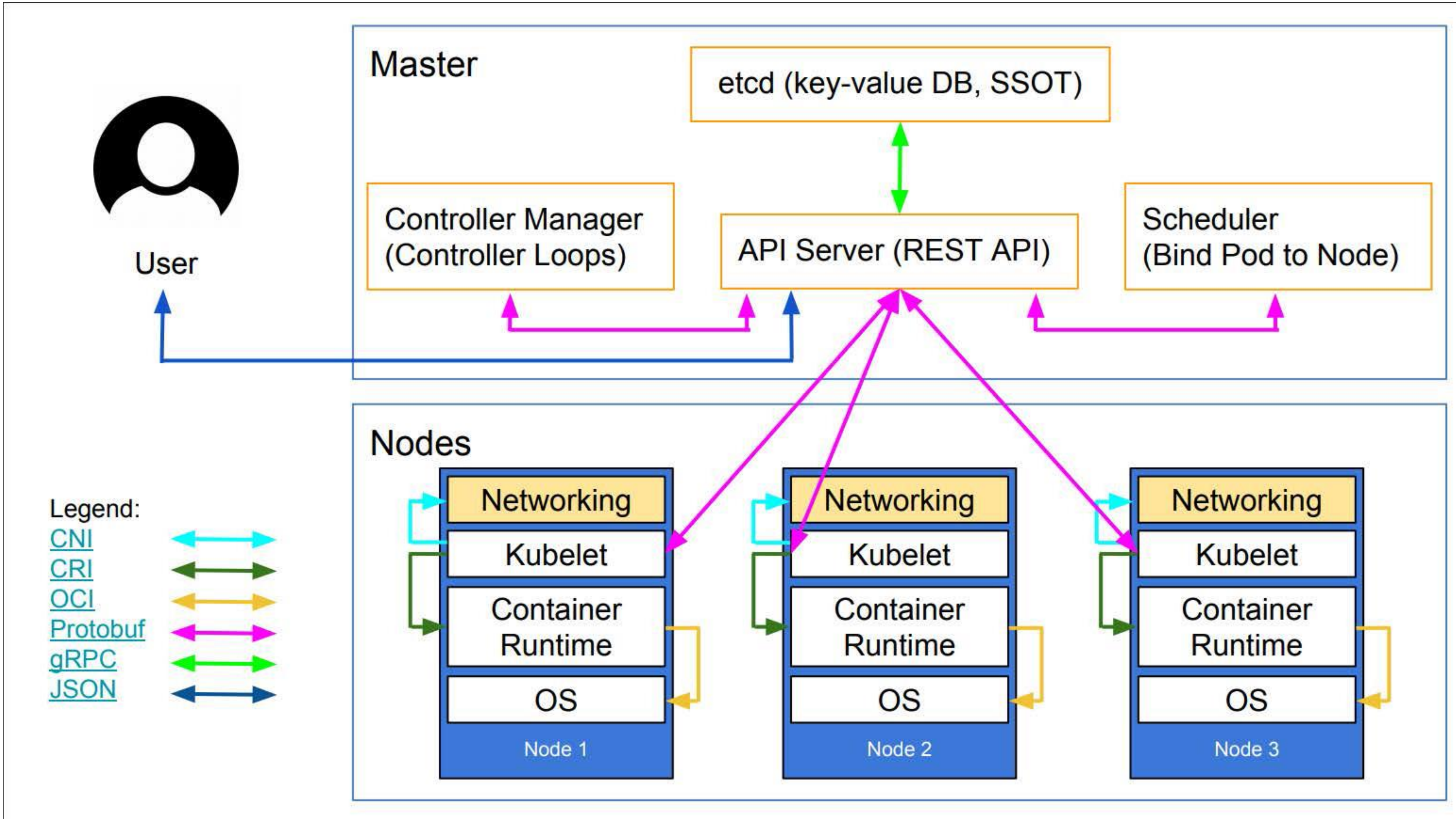
- Add / remove Kubernetes nodes
- Lifecycle management (OS updates, Kubernetes upgrades)
- Monitoring (Prometheus)
- Logging (EFK)

Kubernetes key concepts -Nodes

- **Nodes**
  - Worker machine in Kubernetes
  - Can be physical or virtual
  - Used to be called a minion

- **Master Node**
  - Runs the k8s control plane
  - Can be replicated for high availability

Kubernetes Control Plane



Kubernetes Control Plane

- The control plane is all about maintaining your desired state. For example:
  - If a pod should have 3 copies of a container, it will make sure it always has *exactly* 3.

Kubernetes key concepts –Objects and Names

- **Objects**
  - Persistent entities in k8s system
  - Containers, resources, policies

- **Names**
  - Every object has a unique name which is client provided
  - Every object also receives a unique UID automatically

Kubernetes key concepts – Namespaces and Labels

- **Namespaces**
  - Multi-tenancy construct that allows for the reuse of names by different tenants
  - Also a technology inside the Linux kernel for isolating processes
- **Labels**
  - Non-unique strings assigned to objects to group them by role or function
- Examples
  - Tier=production
  - App=web_server
- Policies can reference labels

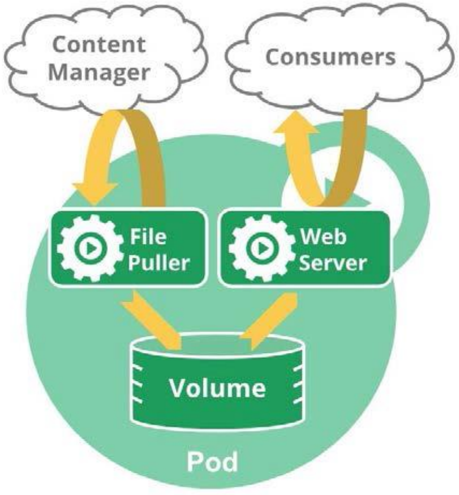
Kubernetes key concepts – Containers and images

- **Images**
  - The collection of binaries, libraries, and other files needed to run an application

- **Container**
  - A running application with its associated resources (i.e. the image)
  - Short lived and usually stateless

Kubernetes key concepts –Pods

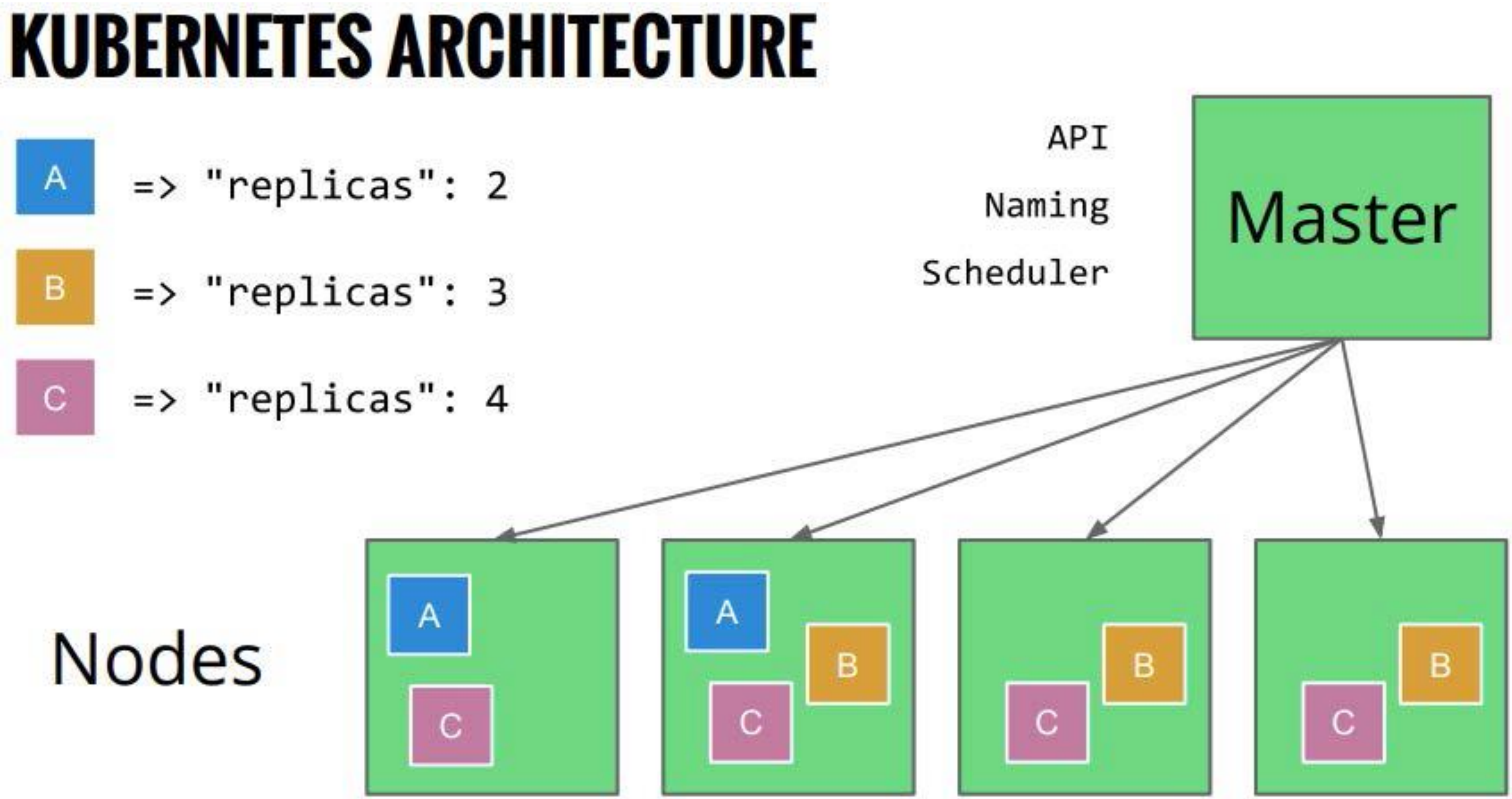
- **Pods**
  - The smallest schedulable resource in k8s
  - One or more containers live in a pod.
  - All containers within a pod always run on the same Node
  - All containers within a pod see the same files and can communicate via IPC, shared files, network stack on localhost
  - Each Pod has its own unique intra-cluster IP
  - Note: each container in the Pod shares the pods IP.



Kubernetes key concepts – Containers and images

- **Replicas**
  - Multiple copies of a pod.
  - Normally scheduled on different nodes.
  - Remember containers are normally stateless.

Kubernetes Architecture



Kubernetes key concepts –Services and Volumes

- **Services**
  - Any time we expose a Pod to the network outside of the Pod we call that a service
  - Services can have 3 types of connectivity
    - Intra-cluster only
    - Node TCP/UDP port
    - Externally Load Balanced
- **Volumes**
  - Storage that is associated with a pod
  - Over 15 different types of volumes tailored to different use cases

Kubernetes key concepts –CNI

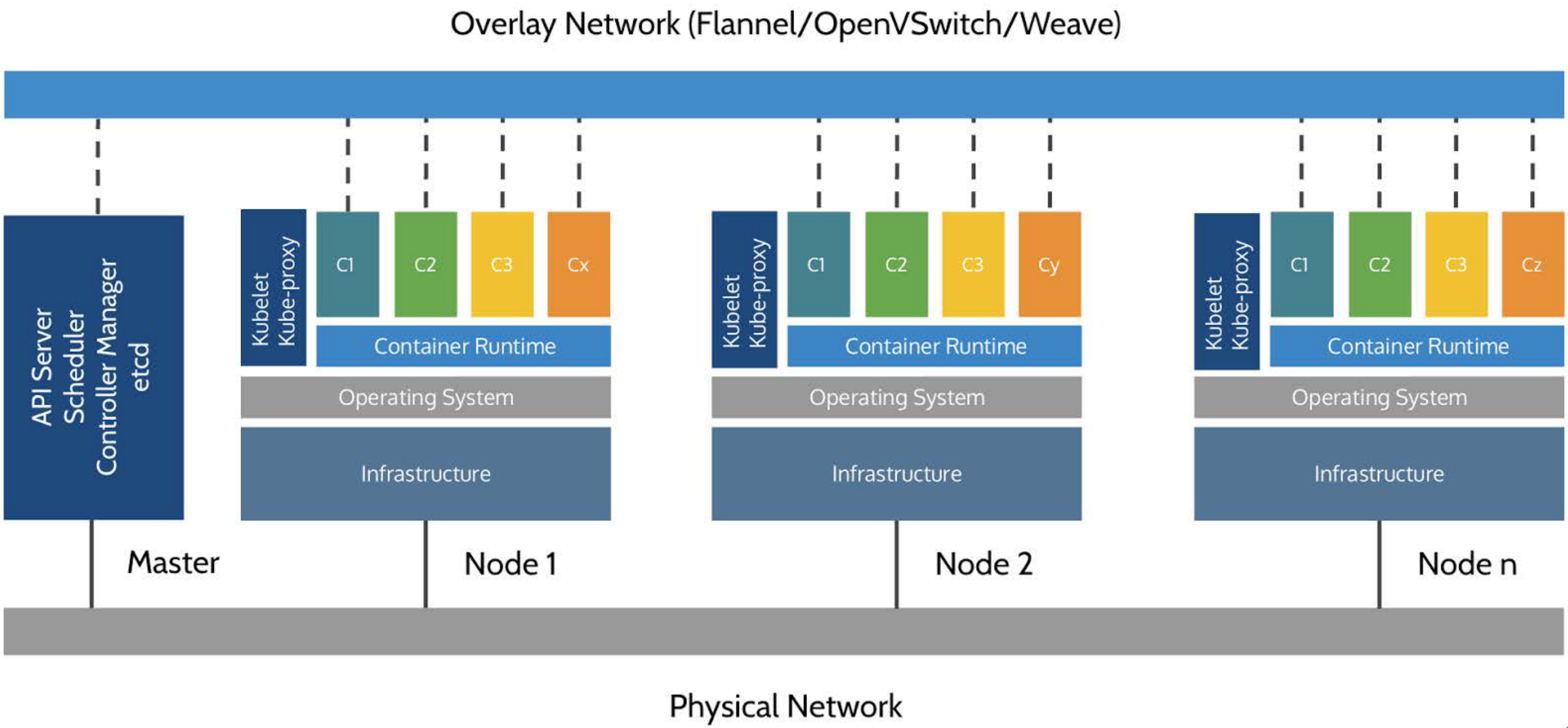
Container Network Interface (CNI)

- A generic plugin interface that allows for a wide variety of networking configurations.
- Anything from simple NAT to full routing with BGP is possible.
- Use VLANs, VXLAN, Genevefor Layer 2 between pods.

Network plugin

- A program that is run by the container management system.
- Creates a virtual ethernet interface for each container namespace and configures the cluster network.

Kubernetes Cluster



Contiv–an example CNI plugin

Rich Policy Framework
Set bandwidth and isolation policies in a multi-tenant environment.

Multi-Platforms
Docker, Kubernetes, OpenShift and more.

Multi-Infrastructure
VMs, containers, and bare metal.

Enterprise Grade
Rigorously tested for the cloud.

Networking Support
Layer 2, Layer 3, BGP, ACI

Open Source
Contiv is available through the Apache 2 License and our code is available on GitHub.

Kubernetes –Lets build a pod

- This is the specification for a pod
  - Every spec must call out the apiVersion, kind, a name, and spec.
  - The specification is similar to what we see in docker.
  - This will download a MongoDB binary, and all of the Linux support binaries and librarys needed for MongoDB to run in the container

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: db
  labels:
    type: db
    vendor: MongoLabs
spec:
  containers:
  - name: db
    image: mongo:3.3
    command: ["mongod"]
    args: ["--rest", "--httpinterface"]

```

Kubernetes –Lets start a pod

- Now let's create the pod
  - The pod specification is saved in db.yml
  - We'll use the kubectl command to create the pod
  - "kubectl create –f db.yml"

```
→  k8s-specs git:(master) cat pod/db.yml
apiVersion: v1
kind: Pod
metadata:
  name: db
  labels:
    type: db
    vendor: MongoLabs
spec:
  containers:
  - name: db
    image: mongo:3.3
    command: ["mongod"]
    args: ["--rest", "--httpinterface"]
→  k8s-specs git:(master) kubectl create -f pod/db.yml
pod/db created
→  k8s-specs git:(master)
```

Kubernetes –the kubectl command

- The **kubectl** command is
  - Our primary tool for interacting with k8s
  - Developers can also write code to interact with the API if desired, but ops will mostly use kubectl or the dashboard

Resource list

- Minikube–Kubernetes on your laptop
- Docker also runs great on a laptop
- Victor Farcic'sKubernetes live class at safarionline(runs every 2-3 months)
  - The free trial is long enough to attend the class

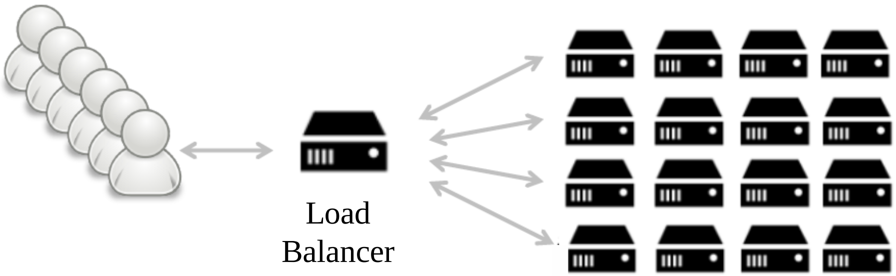
Key Takeaways

- Container technology like Docker and Kubernetes empower developers to be self service, improving turn around time between releases.
- Kubernetes provides IT operations tooling for building highly available and reliable container environments.
- Containers is a new software packaging and distribution format we will all need to know at some level.

Efficient Autoscaling Approach for Microservice-based Cluster to Ensure Quality of Services

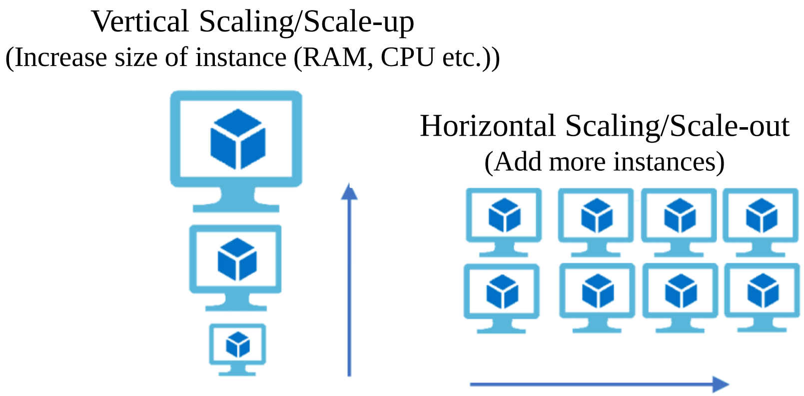
Autoscaling

- Monitor app performance or server load
  - [AWS AutoScale, Google Autopilot]
- Adjust resource given to app
  - Add or remove to meet performance goal
  - Feedback-based control loop
  - Adjust resource given to app
- Benefits
  - Lower cost
  - Automation
  - Reliable performance
  - Fault tolerance
  - Service availability



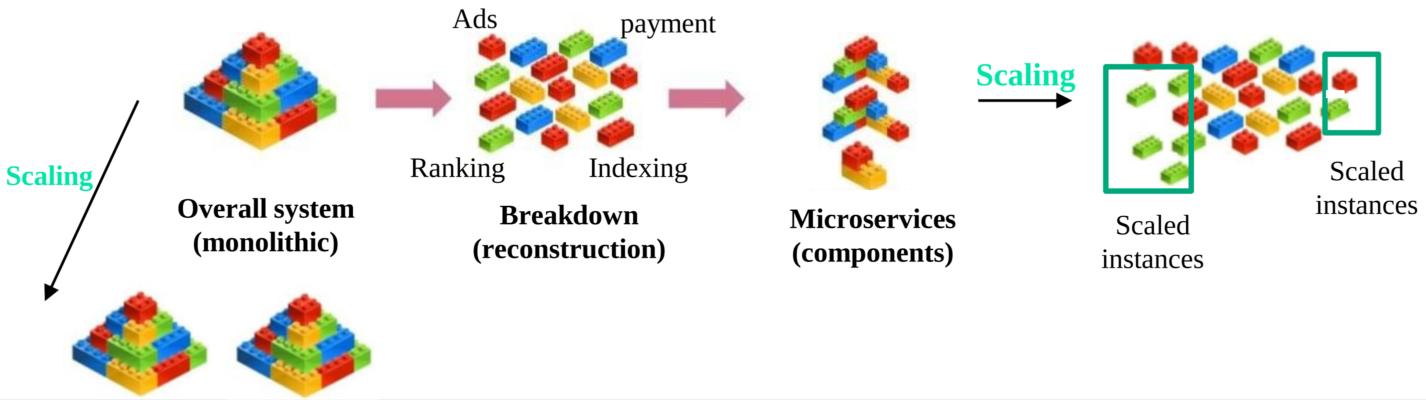
Autoscaling with virtual machines

- Traditional monolithic applications are deployed in virtual machines
- Tradeoffs
  - Monitor performance or resource utilization?
  - Key parameters for controller design?
  - Autoscale within (scale-up) or across servers (scale-out)?
  - Reactive or predictive control?



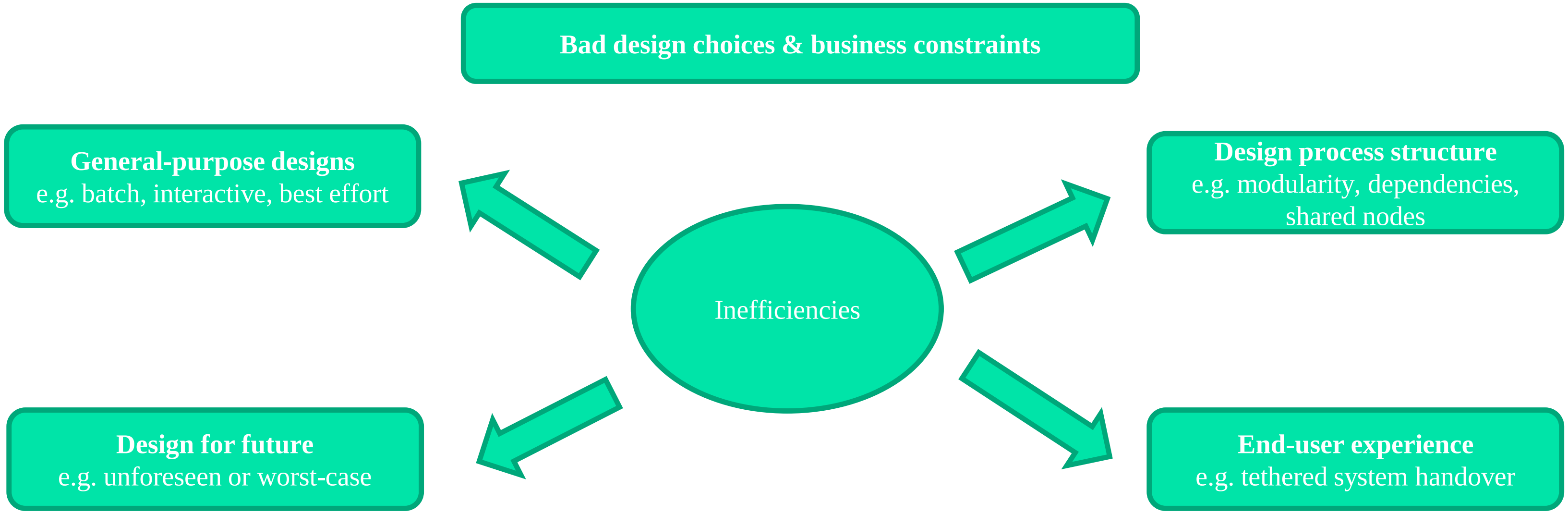
Autoscaling with microservices

- Microservice decomposes applications into a group of **lightweight, fine-grained and self-contained** units/components
- Vertical scaling
  - Increase or decrease the capacity of a single instance/pod
- Horizontal scaling
  - Adding or removing instances of a resource



Source of inefficiency

- Goal:
  - Can we improve resource efficiency while assuring microservice application QoS guarantees?
  - Potential: 30%-50% efficiency; ￥10Ms in cost saving in large-scale cluster



Optimization approaches to assure QoS

- Scaling techniques
  - E.g., single scaling, hybrid scaling, multi-facet scaling
- Match resources to work
  - E.g., turn-off/dial-down unused components
- Overprovisioning
  - E.g., allocate resources based on peak resource usage
- SLO allocation
  - E.g., determine the minimum SLO allocation for applications
- Special purpose solutions
  - E.g., GPUs, ASICs, …
- Tradeoff some other metrics
  - E.g., utilization & tail latency & queueing time
- Cross layers for efficiency
  - Coordinate management across rack/cluster, affinity rules, interference-aware scheduling

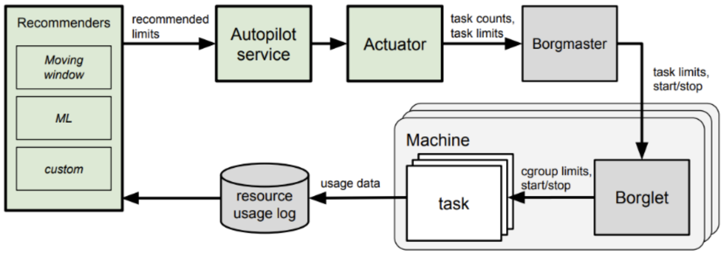
Challenges

- **Variance in workloads**
  - Highly dynamic, multi-dimensional, bursts, unpredictable events
- **Extra large-scale**
  - E.g. hard to manage **5k** services with **10k** nodes and **10 million** cores
- **Complicated dependency**
  - Different calling modes, bottleneck identification, a full function can across **tens of nodes**
- **Latency-sensitive**
  - Microservice units can be processed in **several milliseconds**
- **Frequent scale-in and scale-out**
  - Operations of add/remove/upgrade applications components always happen

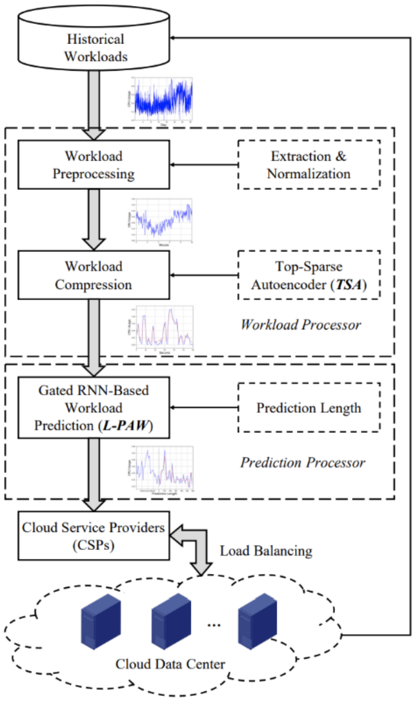
Z. Zhong, M. Xu* et al., Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions, ***ACM Computing Surveys (CSUR)***, Vol. 54, No. 10, 2022 (**citations>100**).

Variance in workloads

- Solution: load/utilization prediction
  - Key in avoiding excess capacity and QoS violations
- How far ahead do you need to predict?
- Example prediction techniques
  - Moving window average [EuroSys'20]
  - Auto-encoder + deep learning [TPDS'20]
  - Sage [ASPLOS'21]



Autopilot [EuroSys'20]

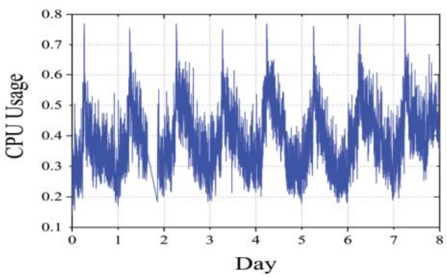


L-PAW [TPDS'20]

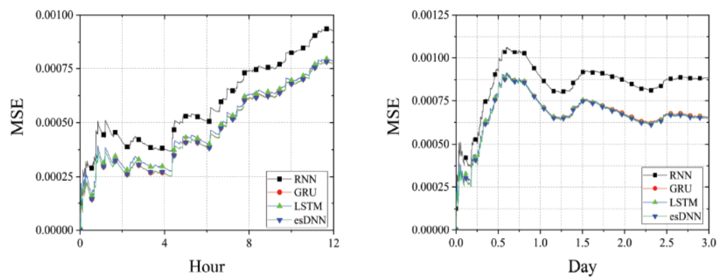
Way To Innovation

Our Solution: Workloads Prediction based on Supervised Learning and GRU

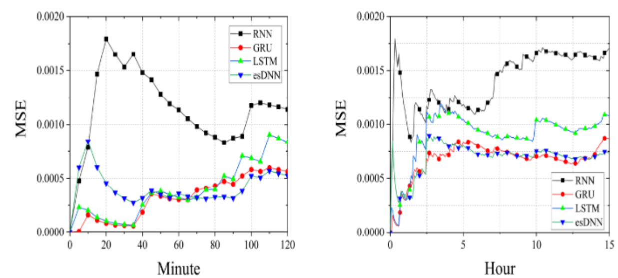
- A sliding window for Multivariate Time Series Forecasting to convert data into supervised learning
- Deep Neural Network algorithm based on GRU for workloads prediction
- Comprehensive experiments with Alibaba and Google Cloud data centers for performance evaluations



Alibaba Trace Periodical trends with 8 days in 2018 with 4000 machines data



esDNN MSE with Alibaba: <0.001



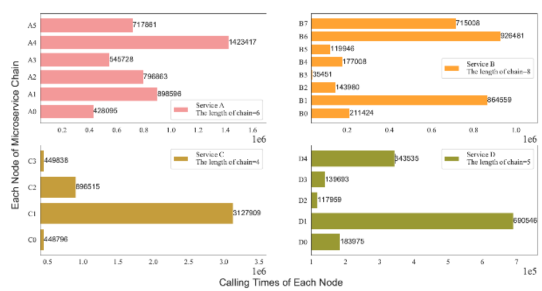
esDNN MSE with Google: <0.002

Remarks: trade-offs concern. Balancing accuracy and efficiency (e.g., training time & inference time) with prediction algorithms in large-scale and realistic environment.

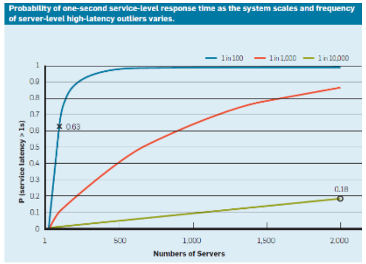
M. Xu et al., esDNN: An Efficient Approach for Multivariate Workload Prediction in Cloud Computing Environments, ***ACM Transactions on Internet Technology (TOIT)***, Vol. 22, Issue 3, 2022.

Extra Large Scale

- Tail at Scale
  - Larger cluster → more prone to high tail latency
  - WeChat: **3k** services over **20k** machines [SoCC’18]
  - Alibaba: more than **10k** nodes with **10 million** cores in 2019
- How to scale microservice with large-scale cluster?
- Existing techniques
  - PEMA [HPDC’22]
  - SHOWAR [SoCC’21]
  - GNN+DQN [SoCC’22]



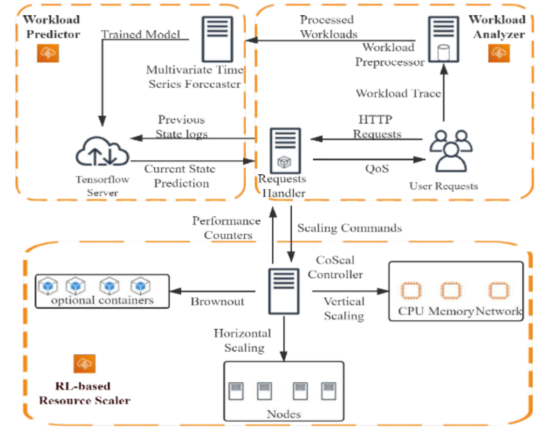
Our analysis of calling times with Alibaba trace [SPE 2023]



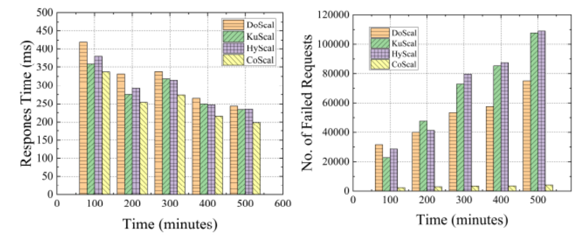
Probability of response time as system scales [Commun. ACM’ 13]

Our Solution: Multi-faceted Scaling of Microservices with RL

- A performance analysis of horizontal scaling, vertical scaling, and brownout for microservices to investigate trade-offs.
- A reinforcement learning based scaling algorithm for decision making to optimize the performance of microservices.
- CoScal reduces response time by 19%-29% and reduce failures significantly.



  - Reward (QoS): $$R_{qos}(rt) = \begin{cases} e^{-\left(\frac{rt - RT_{max}}{RT_{max}}\right)^2} & , rt > RT_{max} \\ 1 & , rt \le RT_{max} \end{cases}$$
  - Reward (utilization): $$R_{util}(u_k) = \begin{cases} \frac{\sum_{k=1}^{K} U_k^{max} - u_k}{K} + 1 & , u_k \le U_k^{max} \\ \frac{\sum_{k=1}^{K} u_k - U_k^{max}}{K} + 1 & , u_k > U_k^{max} \end{cases}$$
  - Final reward: $$r(s_t, a_t) = \frac{R_{qos}(rt)}{R_{util}(ut)}$$

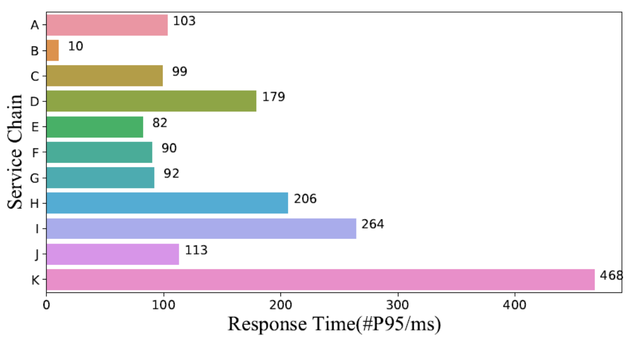


Remarks: scalability concern. Solution space of RL-based **MUST** be carefully designed, otherwise it is infeasible to be applied to large-scale system.

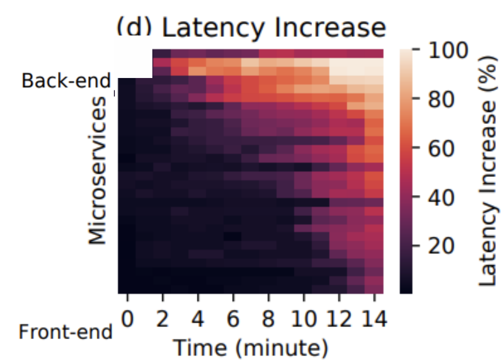
M. Xu et al., CoScal: Multi-faceted Scaling of Microservices with Reinforcement Learning, ***IEEE Transactions on Network and Service Management (TNSM)***, Vol. 19, Issue 4, 2022

Latency Sensitive

- Microservices are more sensitive to latency than traditional applications
- Latency propagation from back-end microservices to the front-end microservices are not synchronous
- How to assure microservices with various QoS via right sizing?
- Existing techniques
  - Parslo [SoCC'21]
  - Seer [ASPLOS'19]



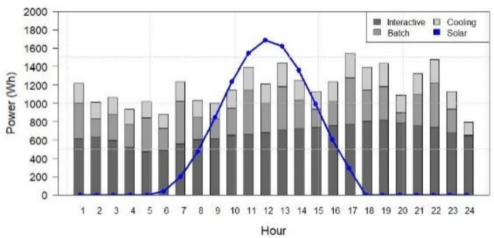
Our analysis with response time of different service chains in Alibaba trace [SPE 2023]

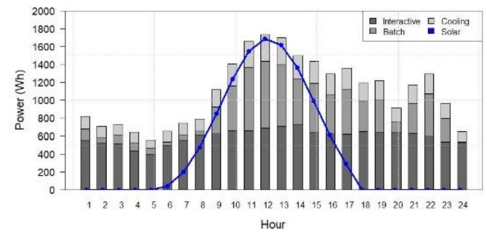


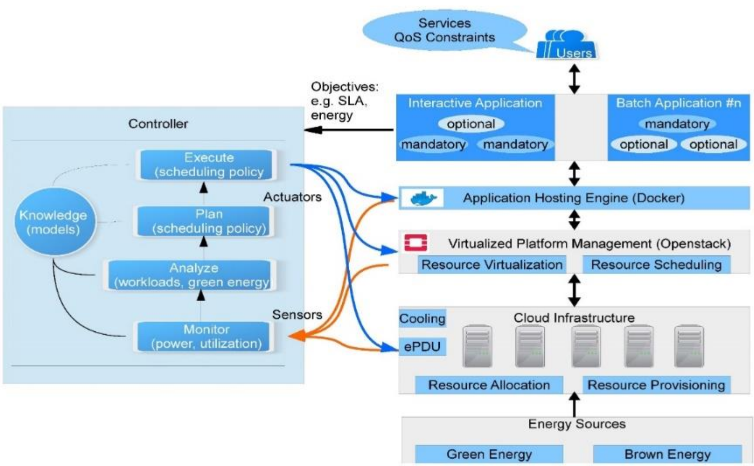
SHOWAR [SoCC'21]

Our Solution:
A Self-adaptive Approach for Managing Co-located Workloads

- Providing a perspective model for multi-level adaptive resource scheduling to manage workloads and renewable energy
- Proposing a self-adaptive approach for interactive workloads and batch workloads to ensure their QoS
- Workloads Model
  - Two types of workloads: $d(t) = \sum_{m} a_m(t) + \sum_{n} b_n(t)$







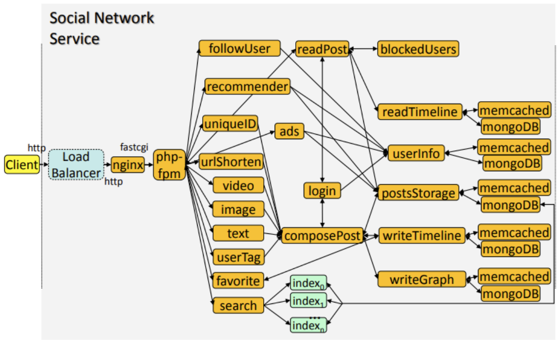
VM consolidation + host scaling + green-aware scheduling

Remarks: priority concern. Comprehensive scaling priorities can be assigned to latency sensitive microservices.

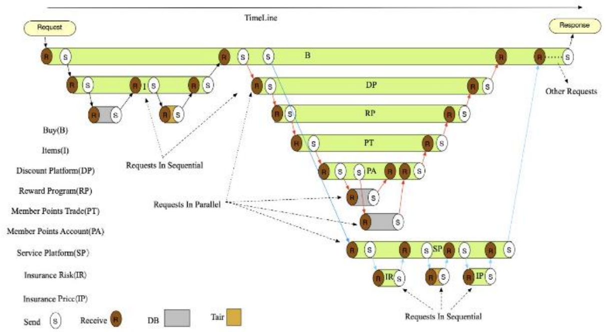
M. Xu et al., A Self-adaptive Approach for Managing Applications and Harnessing Renewable Energy for Sustainable Cloud Computing, ***IEEE Transactions on Sustainable Computing (TSUSC),*** Vol. 6, No.4, 2021

Complicated Dependency

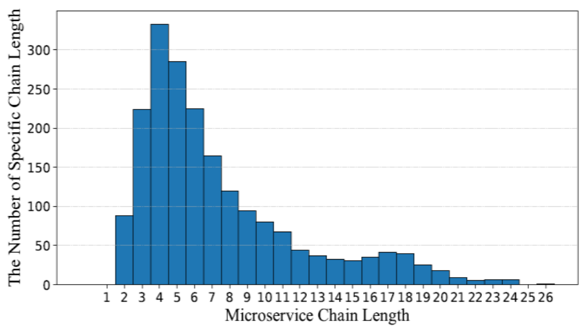
- How to identify key dependencies and scale right amount of resources?
- Existing Examples:
  - FIRM [OSDI'20]
  - Sinan[ASPLOS'21]
  - Alibaba Trace Characterization [SoCC'21, SoCC'22]



Dependency graph of Social Network Application [ASPLOS'19]



Our analysis of requests process with Alibaba trace

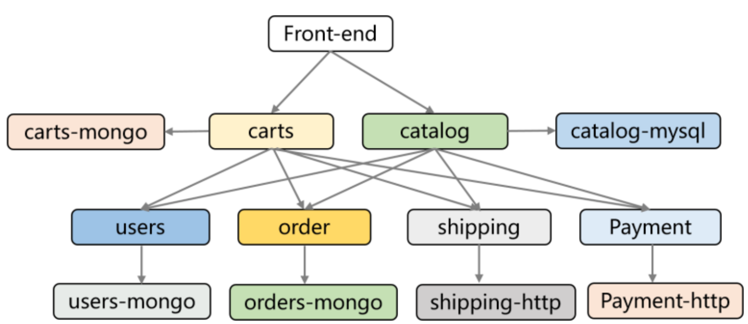


Our analysis of microservice chain with Alibaba trace

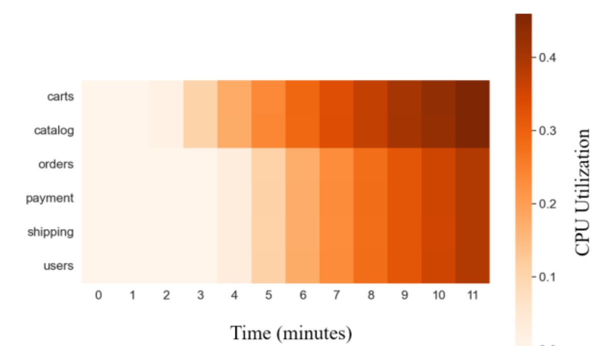
M. Xu et al., Practice of Alibaba Cloud on Elastic Resource Provisioning for Large-scale Microservices Cluster, ***Software: Practice and Experience (SPE)*** 2023.

ChainsFormer (ICSOC 2023)

- To efficiently manage the dynamic nature of microservice chains and adapt to changes quickly, we propose *ChainsFormer*, a chain latency-aware resource provisioning framework for microservices cluster based on chain feature analysis.
- Benefits
  - *ChainsFormer* dynamically scales CPU and memory resources to microservices to ensure high-quality service.
  - *ChainsFormer* utilizes online telemetry data to capture the system state.
  - *ChainsFormer* leverages ML and RL to adapt to variances in the system and reduce the need for manual efforts.



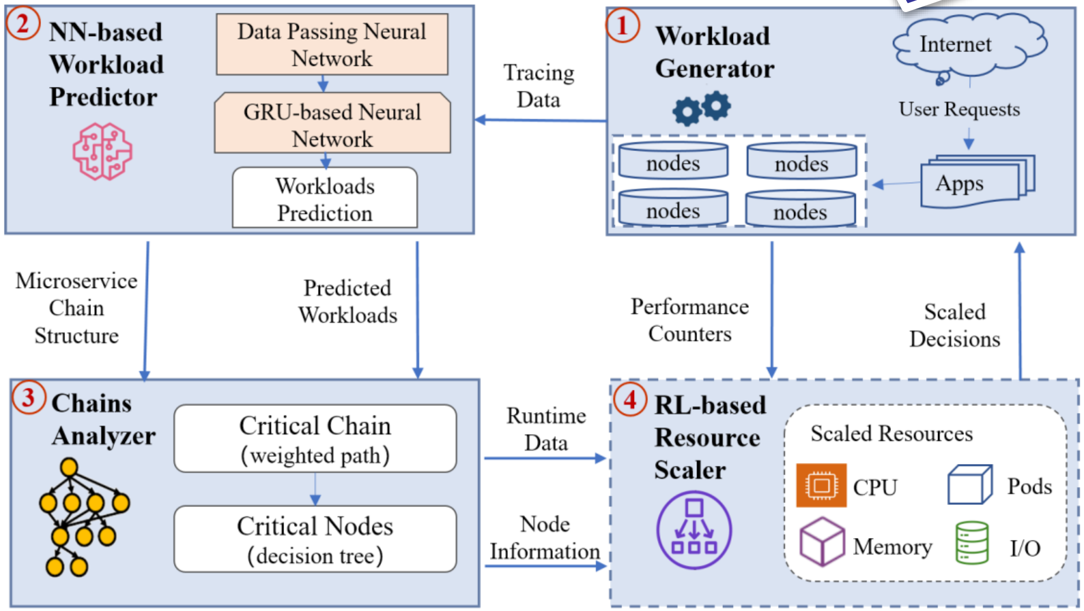
Sock Shop application structure



Inconsistent resource usage propagation

ChainsFormer Framework

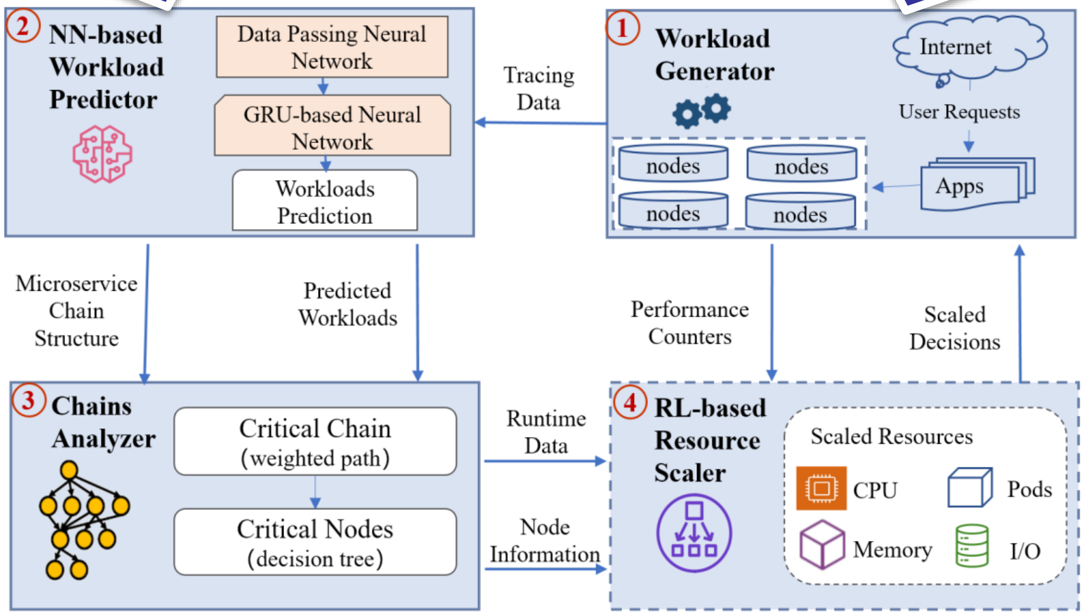
Workload Generator records the number of requests and extracts the tracing data and performance counters.



ChainsFormer Framework

Neural network-based prediction algorithm to estimate future workloads

Workload Generator records the number of requests and extracts the tracing data and performance counters.

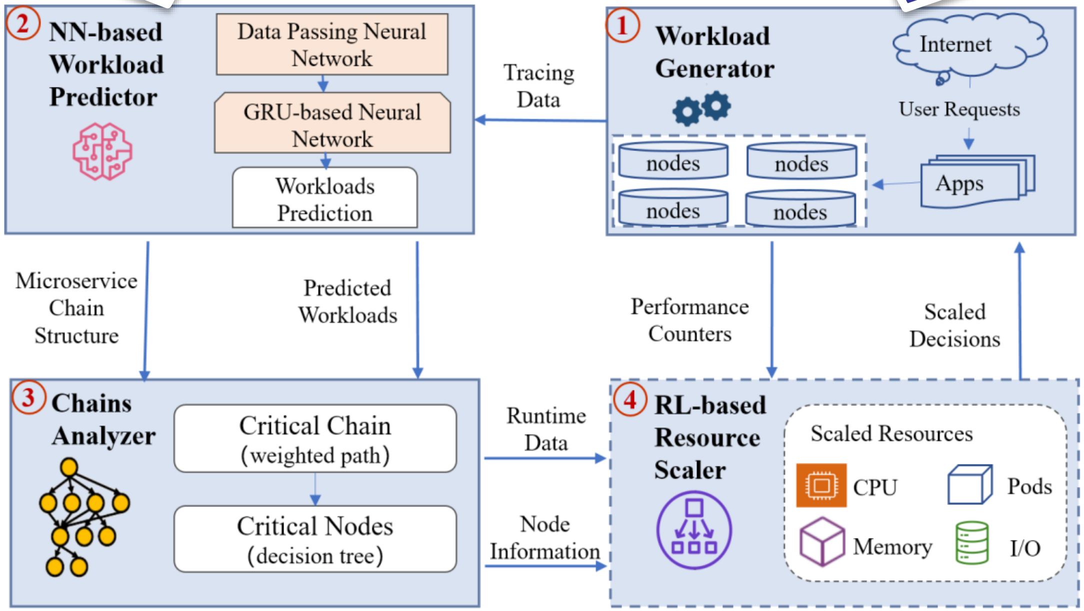


Way To Innovation

ChainsFormer Framework

Neural network-based prediction algorithm to estimate future workloads

Workload Generator records the number of requests and extracts the tracing data and performance counters.

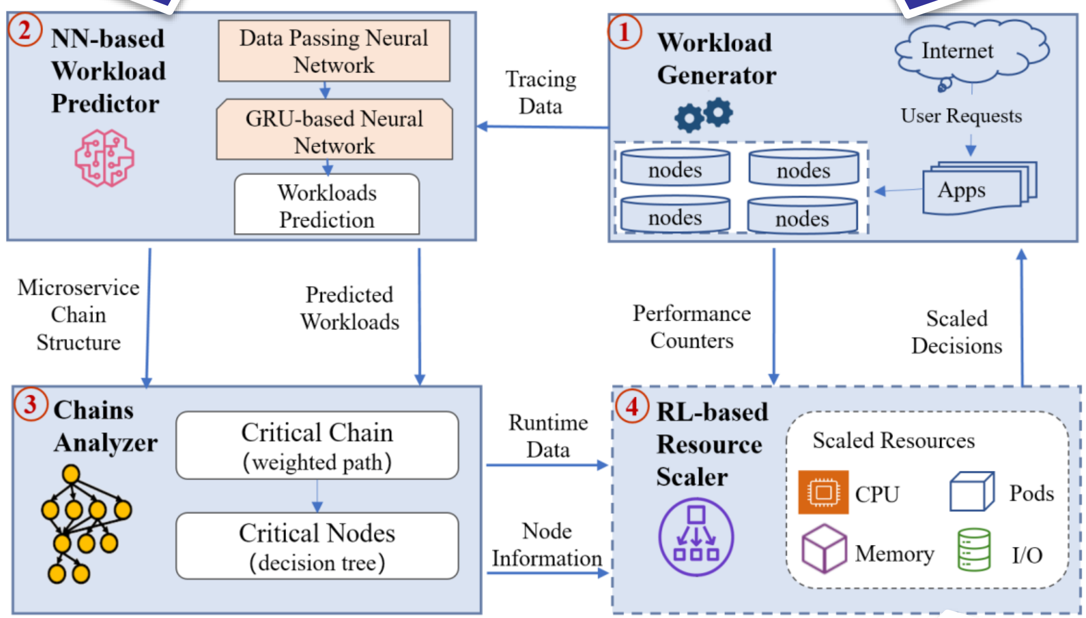


Dynamically identify critical chains and locate critical nodes that result in SLO violations

ChainsFormer Framework

Neural network-based prediction algorithm to estimate future workloads

Workload Generator records the number of requests and extracts the tracing data and performance counters.

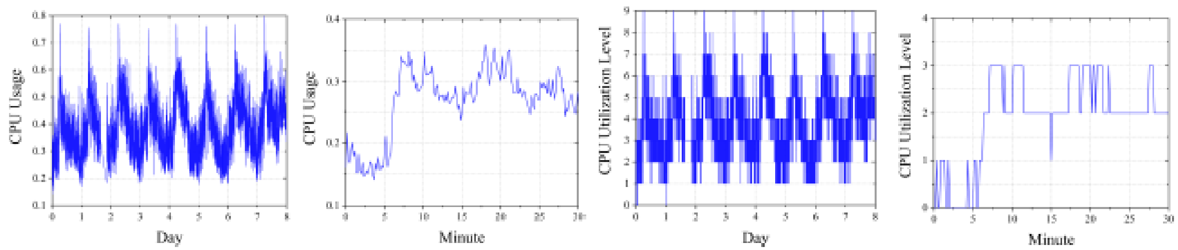


Dynamically identify critical chains and locate critical nodes that result in SLO violations

Make scaling decisions automatically to provision resources for critical nodes

Workload Generator Module

- Workload Generator module is responsible for processing the raw workload trace to make fit with other modules.
  - E.g. timestamp, user id, workloads distributions
- To reduce the state space of our RL model, we process the workloads by dividing the workloads into a number of levels
  - using CPU utilization levels to represent the number of workloads
  - same scaling actions can be applied to the same level to reduce action space

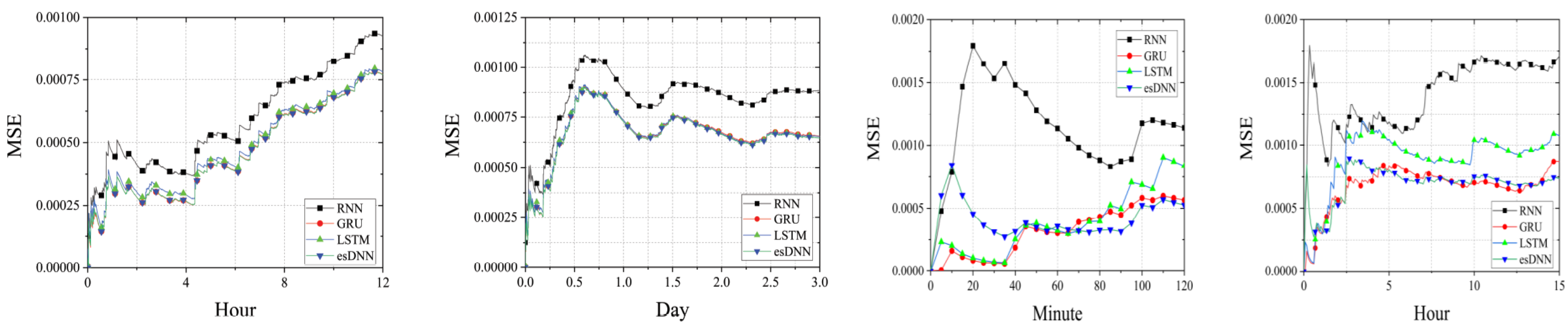


(a) (b) (c) (d)

Fig. 3: (a) Original Alibaba per-day workloads. (b) Original Alibaba per-minute workloads. (c) Converted Alibaba per-day workloads. (d) Converted Alibaba per-minute workloads.

Neural Network-based Workload Predictor Module

- The Workload Predictor can accurately forecast the future workloads in system, and provides information for the RL-based Resource Scaler module to dynamically scale the number of pod replicates.
  - Deep Neural Network algorithm based on GRU for workloads prediction
  - Comprehensive experiments with Alibaba and Google dataset



esDNN MSE with Alibaba: <0.001

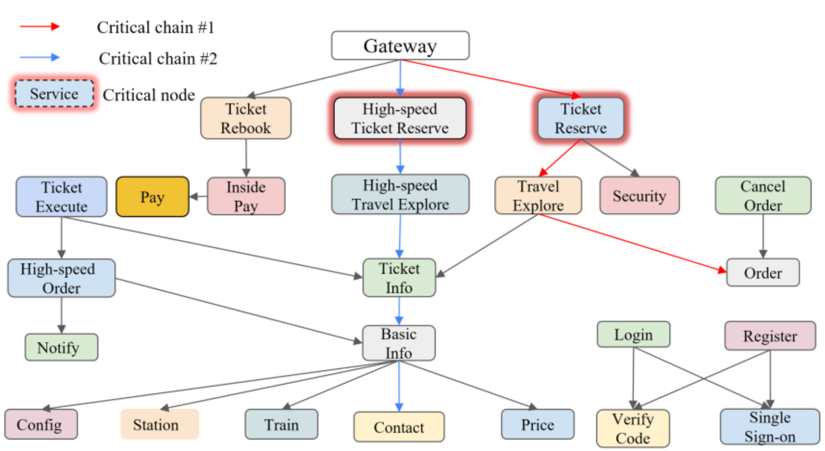
esDNN MSE with Google: <0.002

Remarks: trade-offs concern. Balancing accuracy and efficiency (e.g., training time & inference time) with prediction algorithms in large-scale and realistic environment.

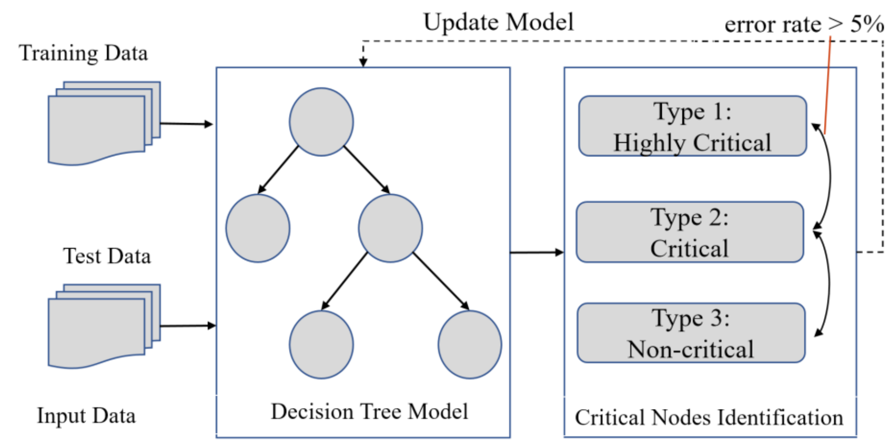
M. Xu et al., esDNN: An Efficient Approach for Multivariate Workload Prediction in Cloud Computing Environments, ***ACM TOIT*** 2022.

Chains Analyzer Module

- Chains Analyzer module is responsible for dynamically identify critical chains and locate critical nodes that result in SLO violations
  - Critical chain: the one with the longest end-to-end latency
  - Critical nodes: the ones have substantial impacts on critical chain
- Weighted longest path algorithm to find the critical chain
  - Low complexity!
- Critical nodes are identified with decision tree
  - Updated quickly!



Dynamic critical chains in TrainTicket



Decision tree for critical node identification

RL-based Resource Scaling Module

- RL-based Resource Scaling module uses RL techniques to determine the optimal scaling actions
  - Exploring a large solution space
  - Responding to dynamic status changes
  - Hybrid scaling with both vertical and horizontal scaling
- SARSA algorithm is applied to learn the policy that maximizes the expected cumulative reward in terms of response time and utilization
  - Fast convergence by updating Q-values based on the current policy

Reward of resource utilization

Utilization threshold

$$R_u(u_k) = \begin{cases} \frac{\sum_{k=1}^{K} U_k^{max} - u_k}{K} + 1, & u_k \leq U_k^{max}, \\ \frac{\sum_{k=1}^{K} u_k - U_k^{max}}{K} + 1, & u_k > U_k^{max}, \end{cases}$$

$$R_q(rt) = \begin{cases} e^{-\left(\frac{rt - RT_{max}}{RT_{max}}\right)^2}, & rt > RT_{max}, \\ 1, & rt \leq RT_{max}, \end{cases}$$

Reward of response time

Acceptable response time

Experiment Settings

- ## Experimental environment
  - Dataset: Alibaba 2018 trace
  - Application: TrainTicket
  - Jaeger for monitoring requests distritbution
  - Kubernetes cluster with 5 nodes
- ## Baselines
  - CF (ChainsFormer) : our approach
  - KS: used by native Kubernetes
  - AUTO: derived from Google Autopilot
  - FIRM [OSDI 20]: with RL and ML
- ## Metrics
  - Request per second (RPS)
  - Number of failures
  - Average response time

Experiments Analyses

- Good Performance is achieved by our approach
  - CF optimized the requests per second up to 8.1% compared to the baselines
  - CF can reduce the number of failures by 8.3% compared to FIRM
  - CF optimizes response time by 1.4% to 26.6% compared to the baselines.

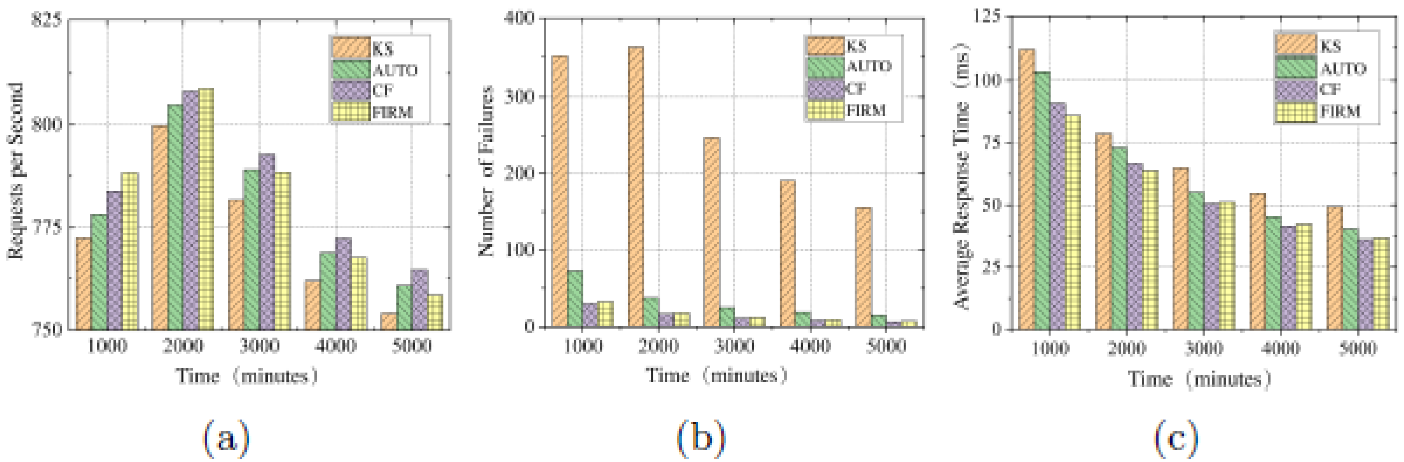


Fig. 5: Comparison of (a) requests per second, (b) number of failures, and (c) average response time.

Potential Research Directions

- Match resources to work (SLOs) with autoscaling
  - Beating the end-to-end SLO is no better than meeting it
  - Lower latency than the SLO does not help
- Accurate and novel metrics
  - Interference-aware scenario
  - Queueing time or hardware counters can be efficient to measure performance
- Efficiently use ML/DL approaches
  - Balancing trade-offs
- Unknown/scale-in/scale-out apps management
  - Especially in public clouds