中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

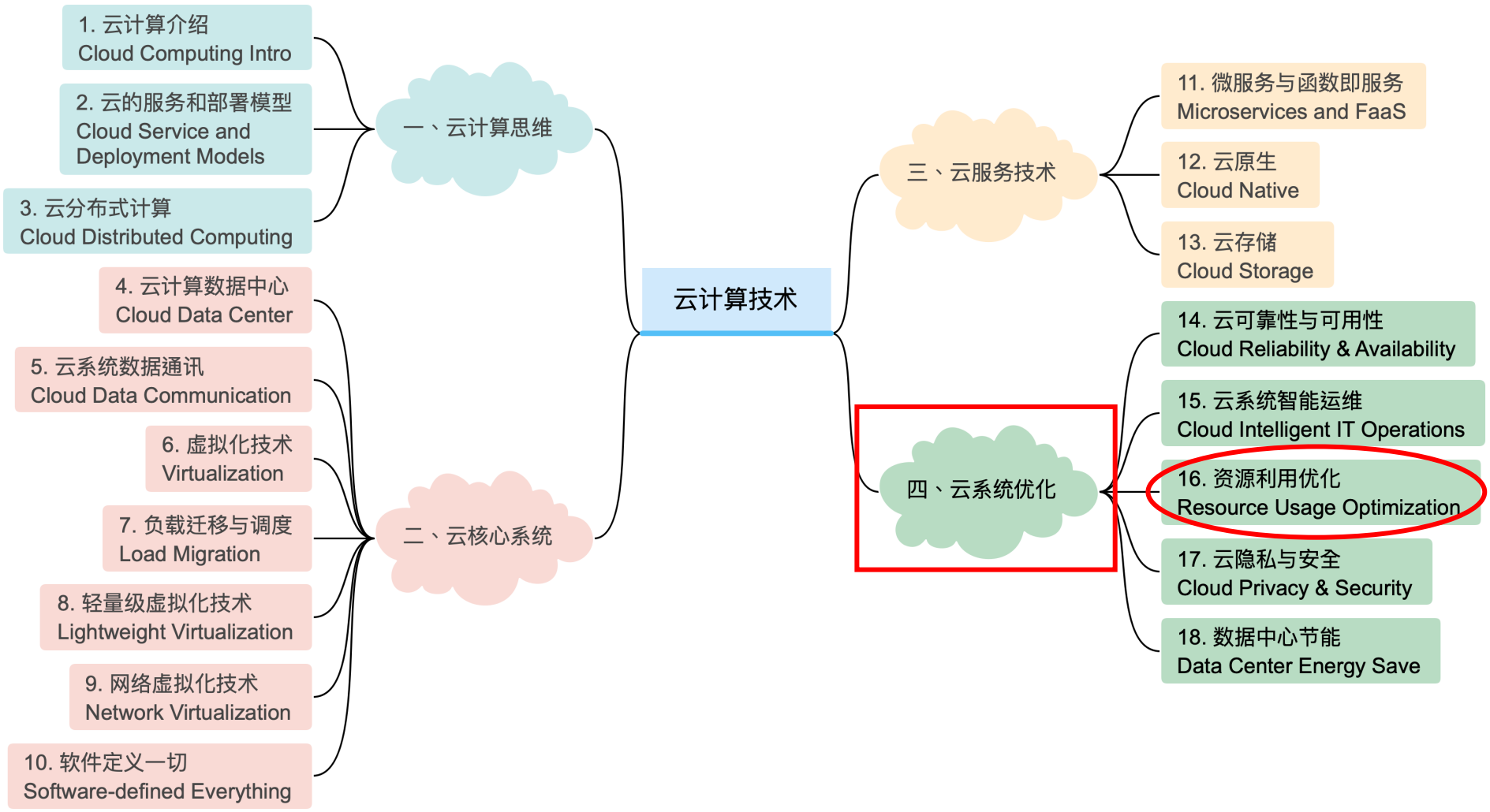
Lecture 16：资源利用优化

SSE316：云计算技术
Cloud Computing Technologies

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn


云计算技术
一、云计算思维
1. 云计算介绍
Cloud Computing Intro
2. 云的服务和部署模型
Cloud Service and
Deployment Models
3. 云分布式计算
Cloud Distributed Computing
二、云核心系统
4. 云计算数据中心
Cloud Data Center
5. 云系统数据通讯
Cloud Data Communication
6. 虚拟化技术
Virtualization
7. 负载迁移与调度
Load Migration
8. 轻量级虚拟化技术
Lightweight Virtualization
9. 网络虚拟化技术
Network Virtualization
10. 软件定义一切
Software-defined Everything
三、云服务技术
11. 微服务与函数即服务
Microservices and FaaS
12. 云原生
Cloud Native
13. 云存储
Cloud Storage
四、云系统优化
14. 云可靠性与可用性
Cloud Reliability & Availability
15. 云系统智能运维
Cloud Intelligent IT Operations
16. 资源利用优化
Resource Usage Optimization
17. 云隐私与安全
Cloud Privacy & Security
18. 数据中心节能
Data Center Energy Save

Today’s topics

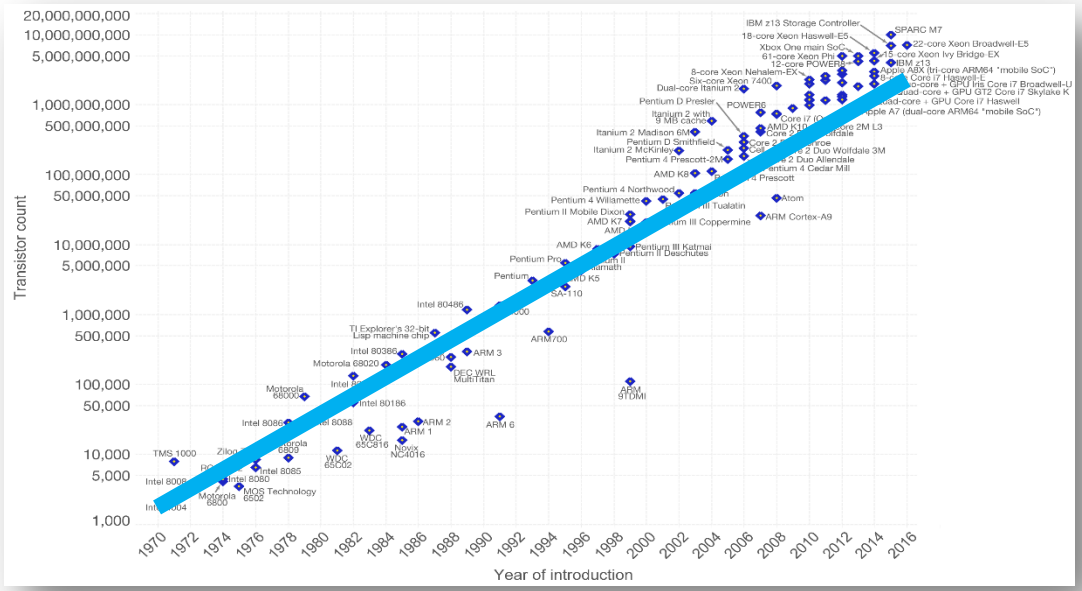
- 数据中心能耗问题
- 资源弹性伸缩
- 请求负载均衡

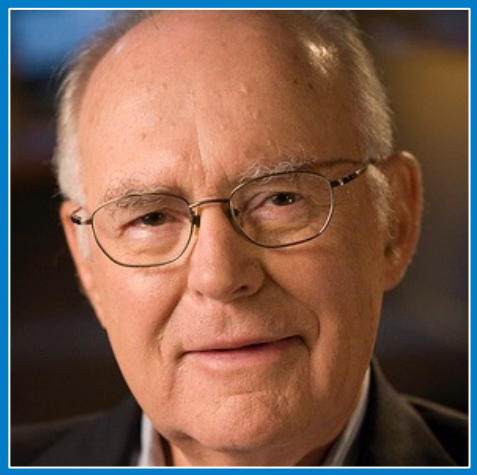
数据中心能耗


功耗视角
数据中心可达…
百万千瓦级
“…或需核电站来满足未来大型计算机系统…”
–《工程与技术杂志》（美国）
Engineering
and Technology

计算的发展就是不断增加资源

摩尔定律 (1971-2016)



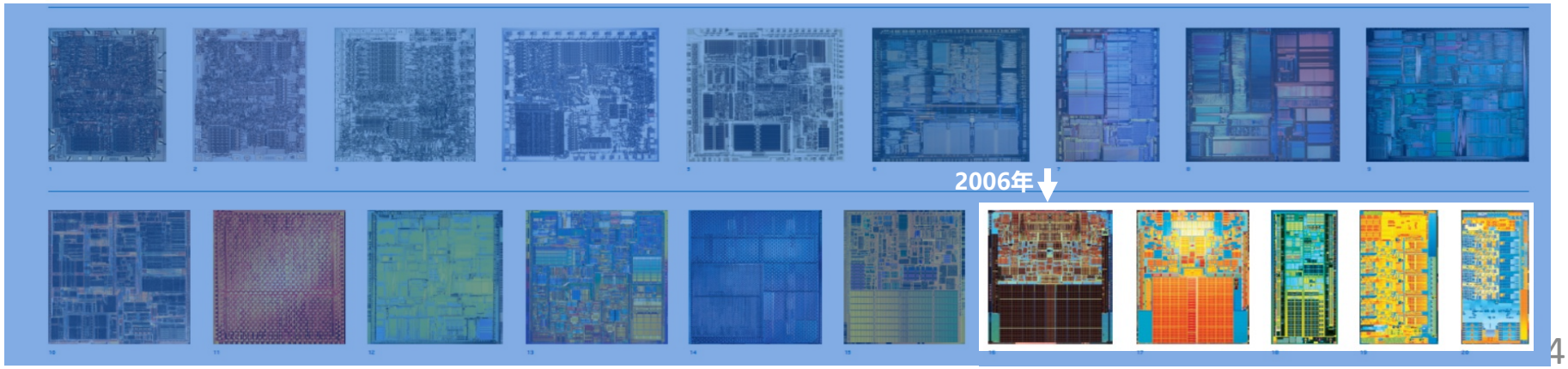


每过一代，CPU上的晶体管数目应当翻一倍

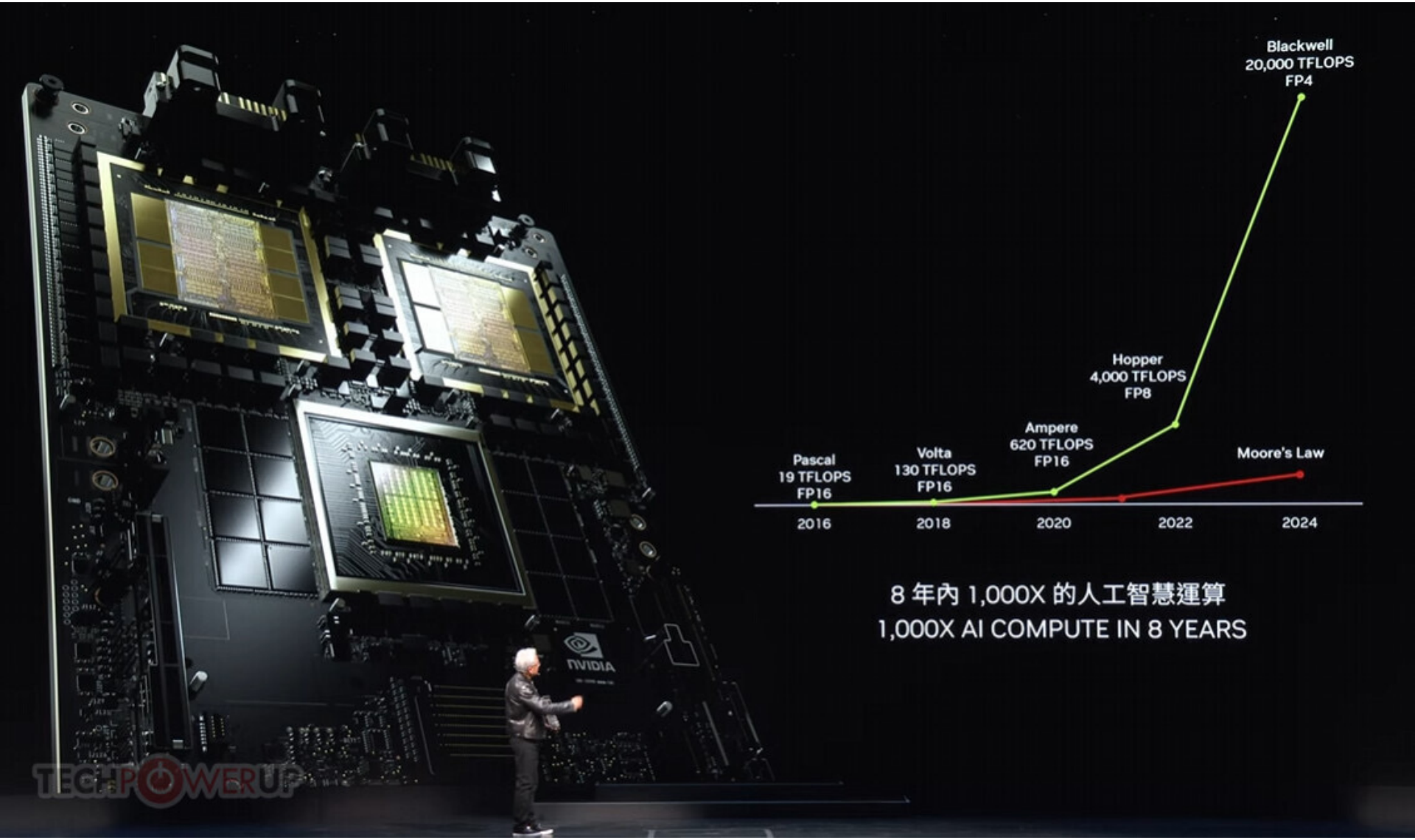
Intel Chips

Throughout Intel's history, new and improved technologies have transformed the human experience.

Decades of Intel chips, including the 22nm 3rd generation Intel® Core™ processor with its revolutionary 3-D Tri-Gate transistors, illustrate Intel's unwavering commitment to delivering technology and manufacturing leadership to the devices you use every day. As you advance through the chart, the benefits of Moore's Law, which states that the number of transistors roughly doubles every couple of years, are evident as Intel increases transistor density and innovates the architecture designs that deliver more complex, powerful, and energy-efficient chips that transform the way we work, live, and play.



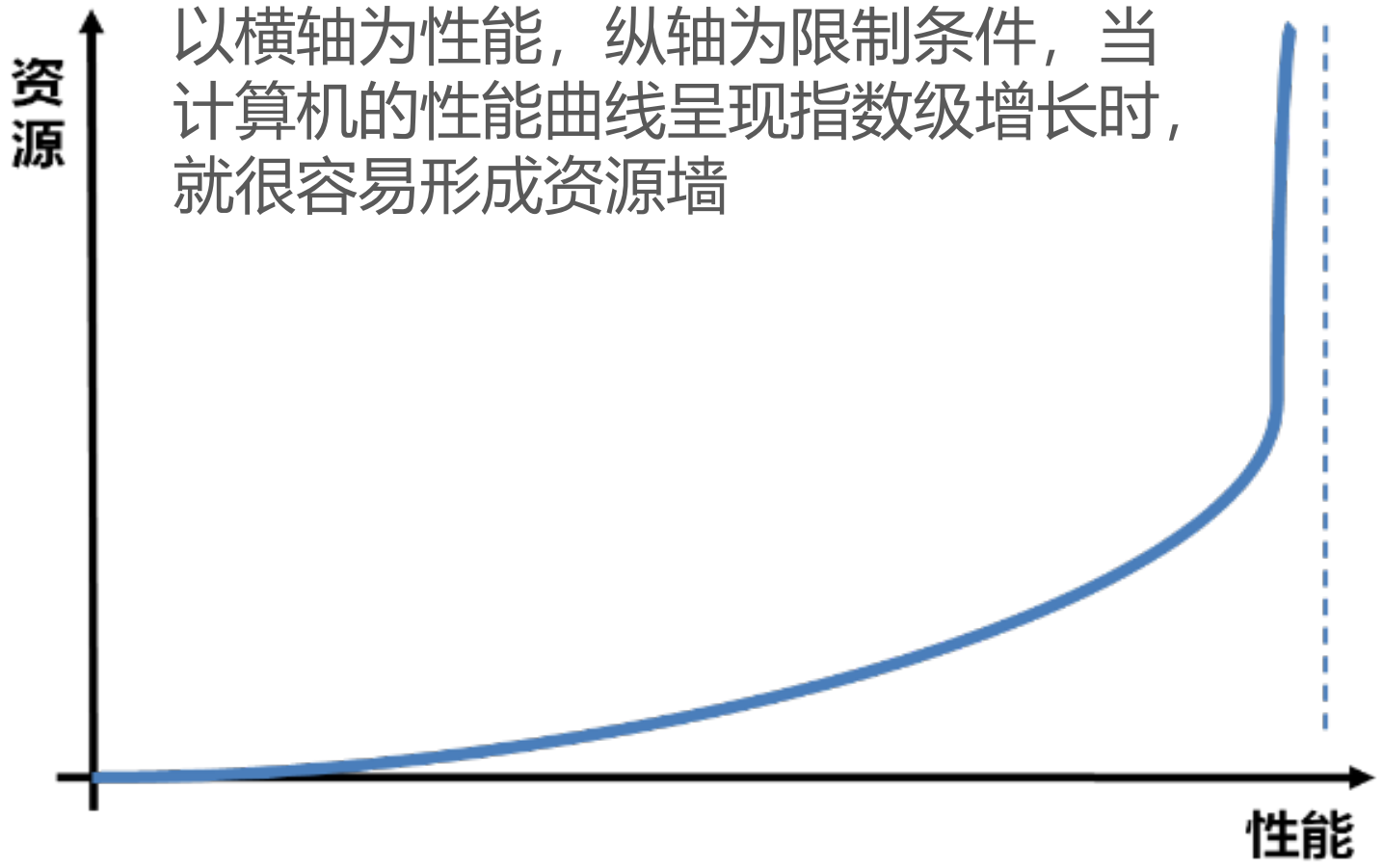
Nvidia ComputeX 2024



https://www.thepaper.cn/newsDetail_forward_27606547

访存/功耗墙

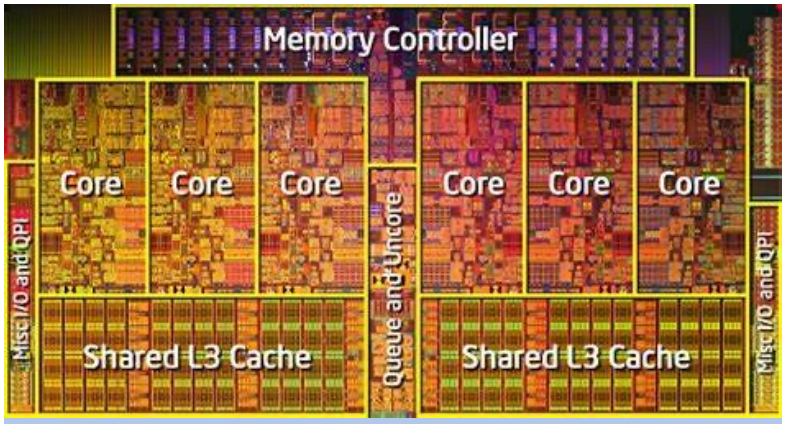
“墙”：限制计算机性能的无形资源瓶颈



以横轴为性能，纵轴为限制条件，当计算机的性能曲线呈现指数级增长时，就很容易形成资源墙

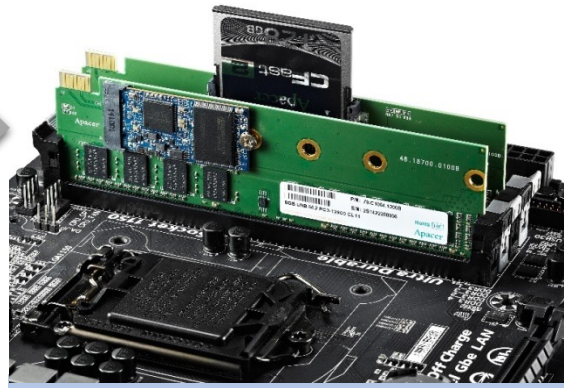
访存/功耗墙

“内存墙”：包括内存容量和访存带宽问题



访问芯片上的缓存速度极快 t_c

访存墙

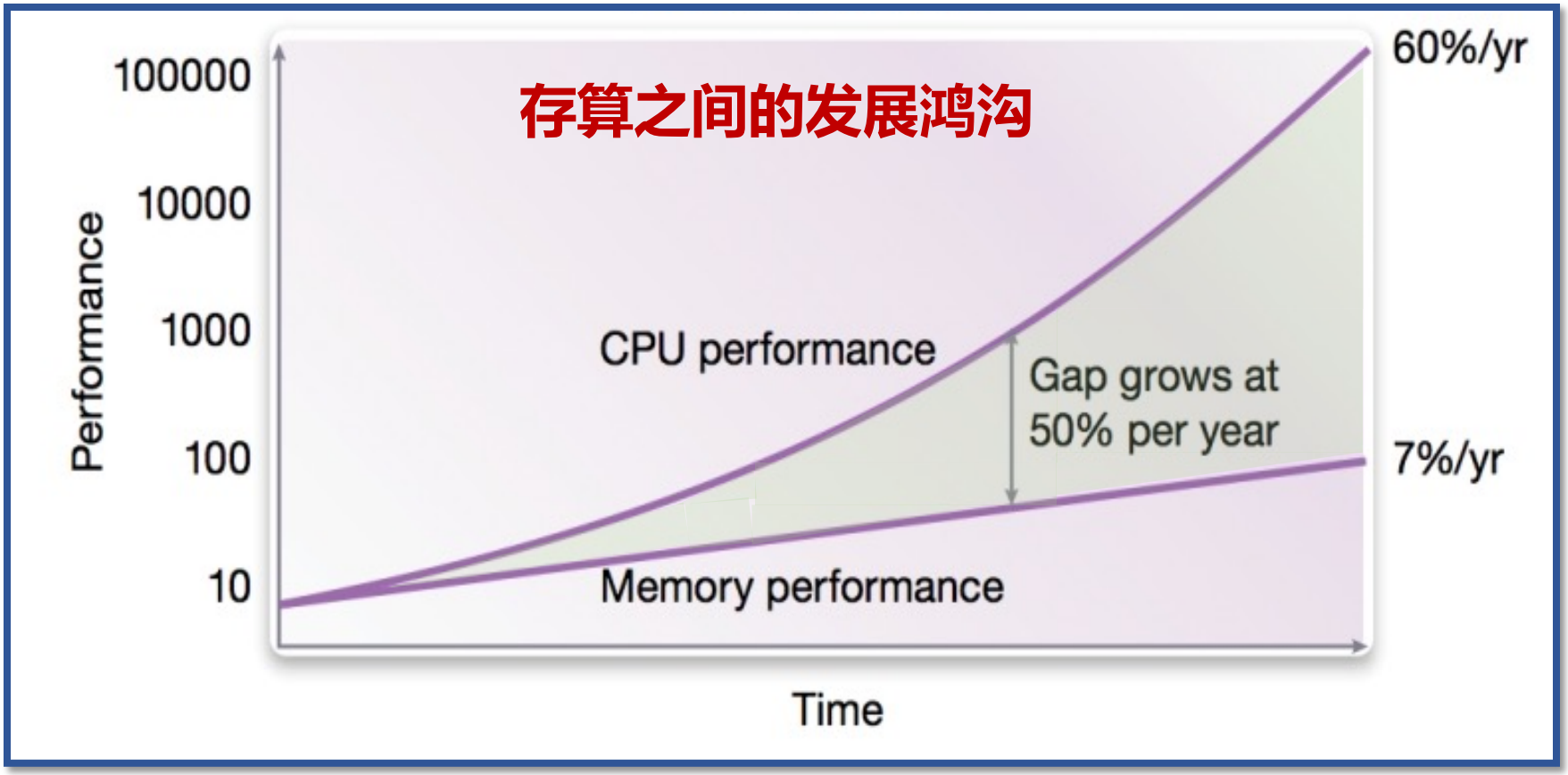


访问内存速度较慢 t_m

$$t_{avg} = p \times t_c + \underbrace{(1-p) \times t_m}_{\text{内存速度限制}}$$

访存/功耗墙

造成“墙”的原因往往在于技术发展速度不均衡

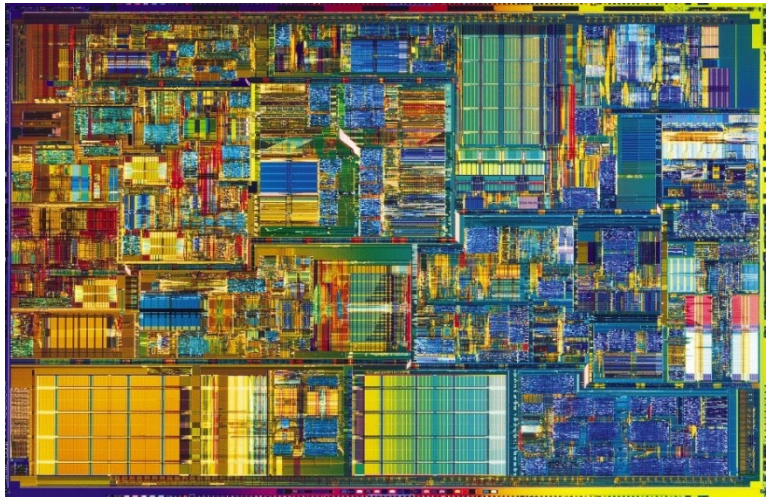


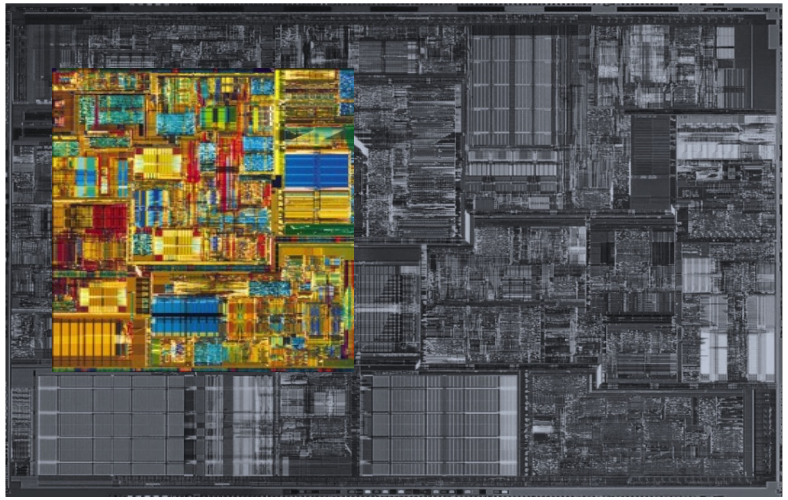
暗硅效应

Michael Taylor, Steven Swanson, Jack Sampson
美国加州大学圣地亚哥分校（UCSD）

暗硅效应

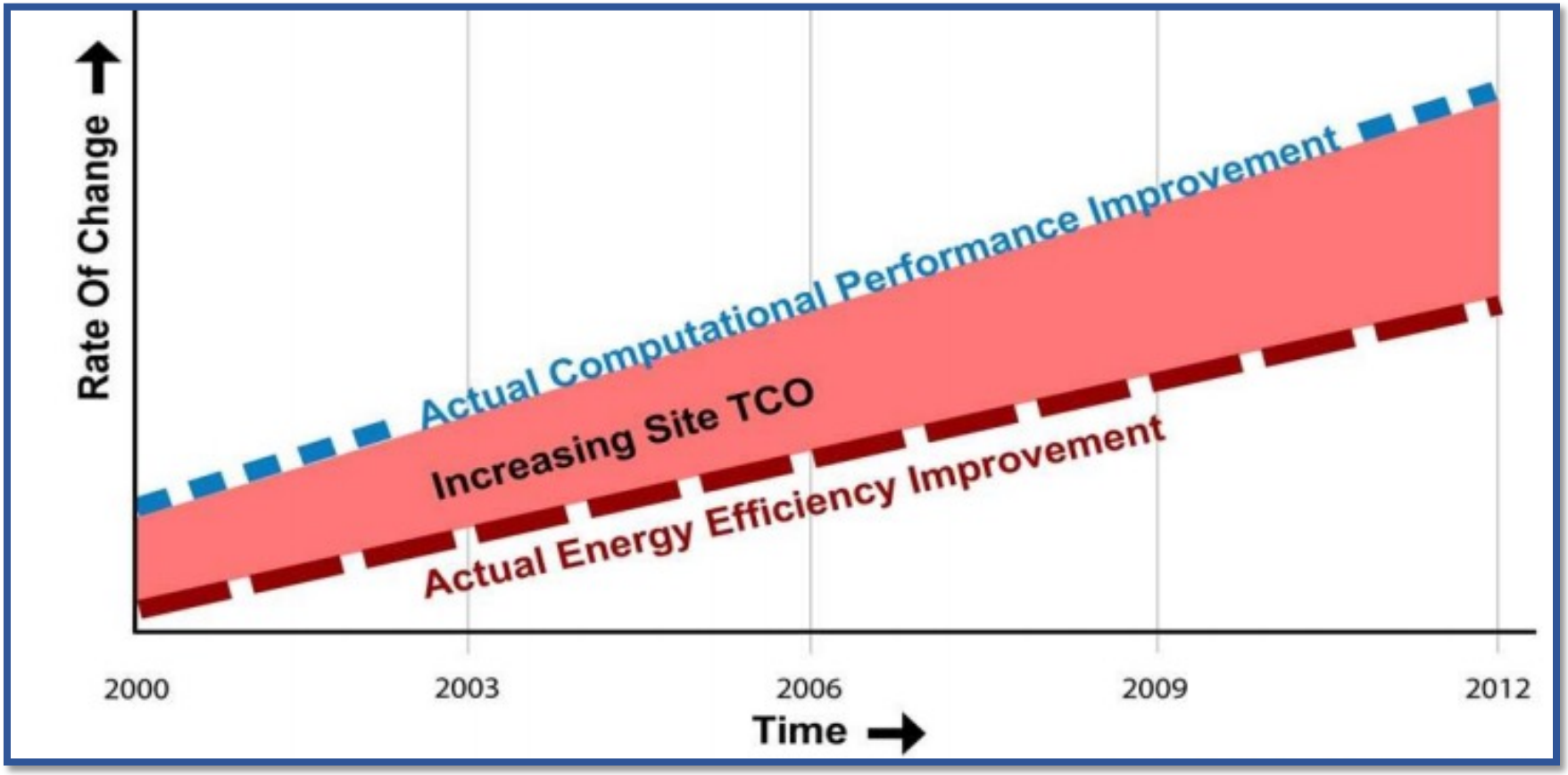
在摩尔定律发展下，芯片上可供使用的晶体管数目(在片上功耗限制下)将呈指数级下降！





访存/功耗墙

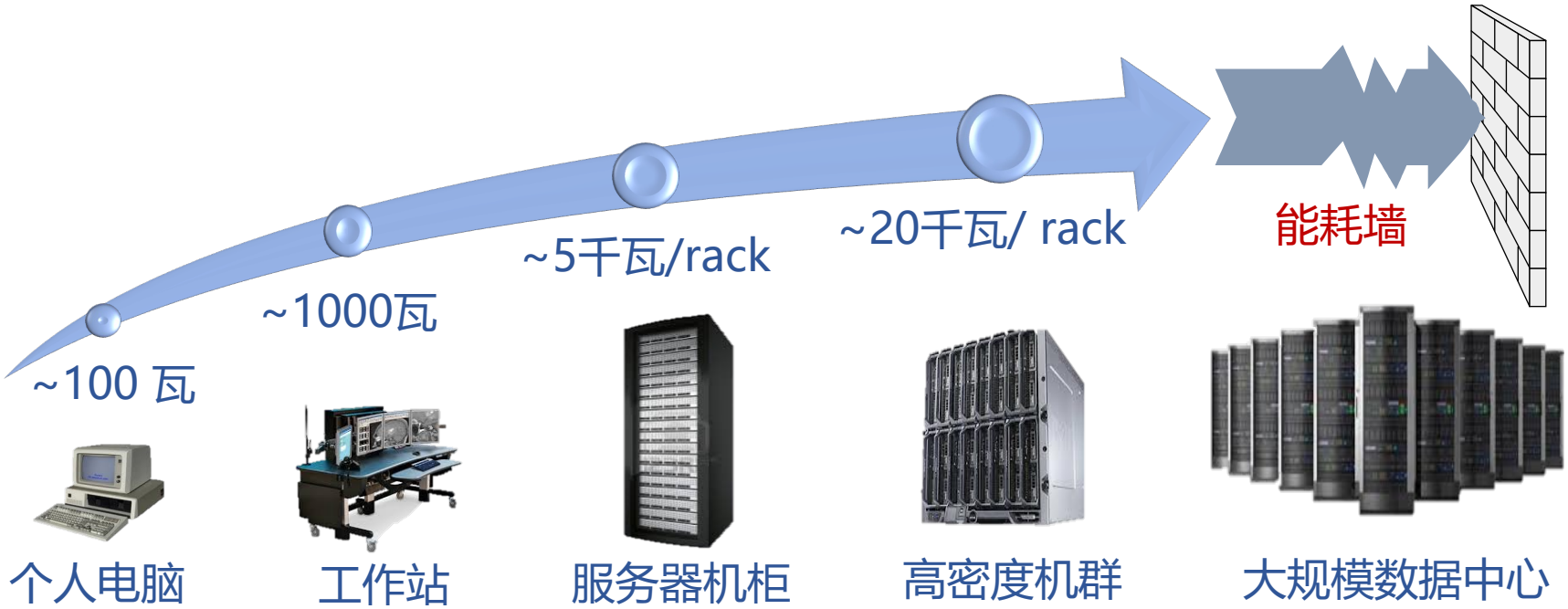
“功耗墙”：依然源于技术发展的不均衡



Source： Ken Brill, The Economic Meltdown of Moore’s Law and the Green Data Center, LISA, 2017

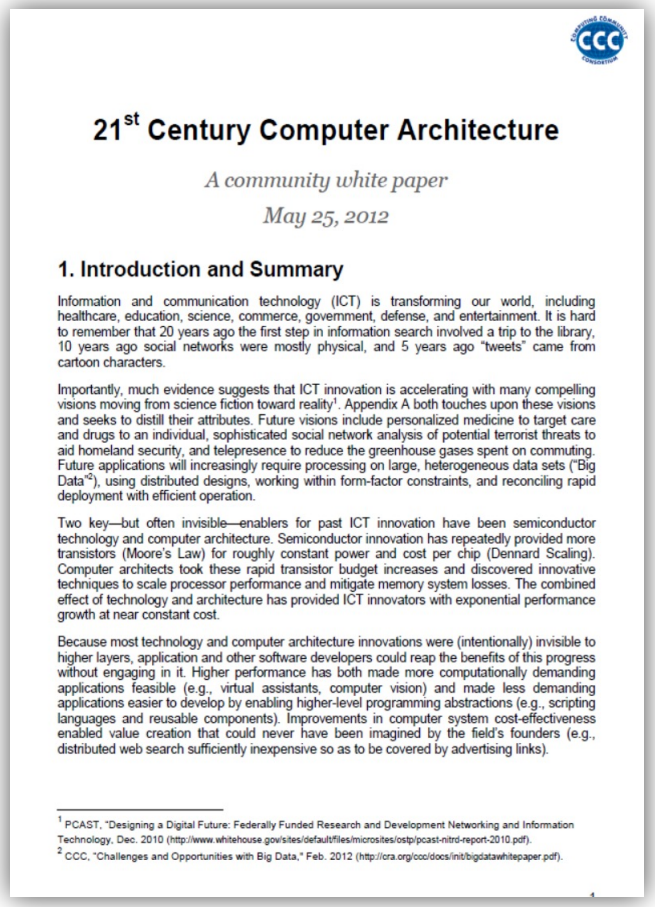
访存/功耗墙

访存/功耗墙


~100 瓦
~1000瓦
~5千瓦/rack
~20千瓦/ rack
能耗墙
个人电脑
工作站
服务器机柜
高密度机群
大规模数据中心

访存/功耗墙

“能耗优先”成为关键设计理念



21st Century Computer Architecture

A community white paper

May 25, 2012

1. Introduction and Summary

Information and communication technology (ICT) is transforming our world, including healthcare, education, science, commerce, government, defense, and entertainment. It is hard to remember that 20 years ago the first step in information search involved a trip to the library, 10 years ago social networks were mostly physical, and 5 years ago "tweets" came from cartoon characters.

Importantly, much evidence suggests that ICT innovation is accelerating with many compelling visions moving from science fiction toward reality[1]. Appendix A both touches upon these visions and seeks to distill their attributes. Future visions include personalized medicine to target care and drugs to an individual, sophisticated social network analysis of potential terrorist threats to aid homeland security, and telepresence to reduce the greenhouse gases spent on commuting. Future applications will increasingly require processing on large, heterogeneous data sets ("Big Data"[2]), using distributed designs, working within form-factor constraints, and reconciling rapid deployment with efficient operation.

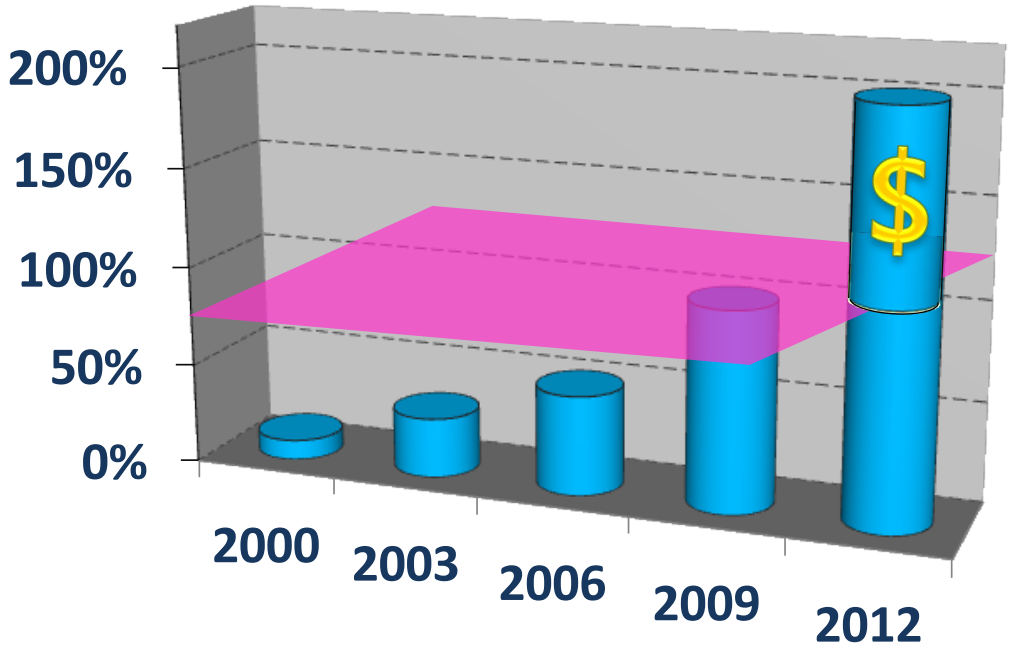
Two key—but often invisible—enablers for past ICT innovation have been semiconductor technology and computer architecture. Semiconductor innovation has repeatedly provided more transistors (Moore's Law) for roughly constant power and cost per chip (Dennard Scaling). Computer architects took these rapid transistor budget increases and discovered innovative techniques to scale processor performance and mitigate memory system losses. The combined effect of technology and architecture has provided ICT innovators with exponential performance growth at near constant cost.

Because most technology and computer architecture innovations were (intentionally) invisible to higher layers, application and other software developers could reap the benefits of this progress without engaging in it. Higher performance has both made more computationally demanding applications feasible (e.g., virtual assistants, computer vision) and made less demanding applications easier to develop by enabling higher-level programming abstractions (e.g., scripting languages and reusable components). Improvements in computer system cost-effectiveness enabled value creation that could never have been imagined by the field's founders (e.g., distributed web search sufficiently inexpensive so as to be covered by advertising links).

[1] PCAST, "Designing a Digital Future: Federally Funded Research and Development Networking and Information Technology, Dec. 2010 (http://www.whitehouse.gov/sites/default/files/microsites/ostp/pcast-nitrd-report-2010.pdf).

[2] CCC, "Challenges and Opportunities with Big Data," Feb. 2012 (http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf).

云数据中心三年期总体能耗成本（占整个IT设备购置成本比例） [1]



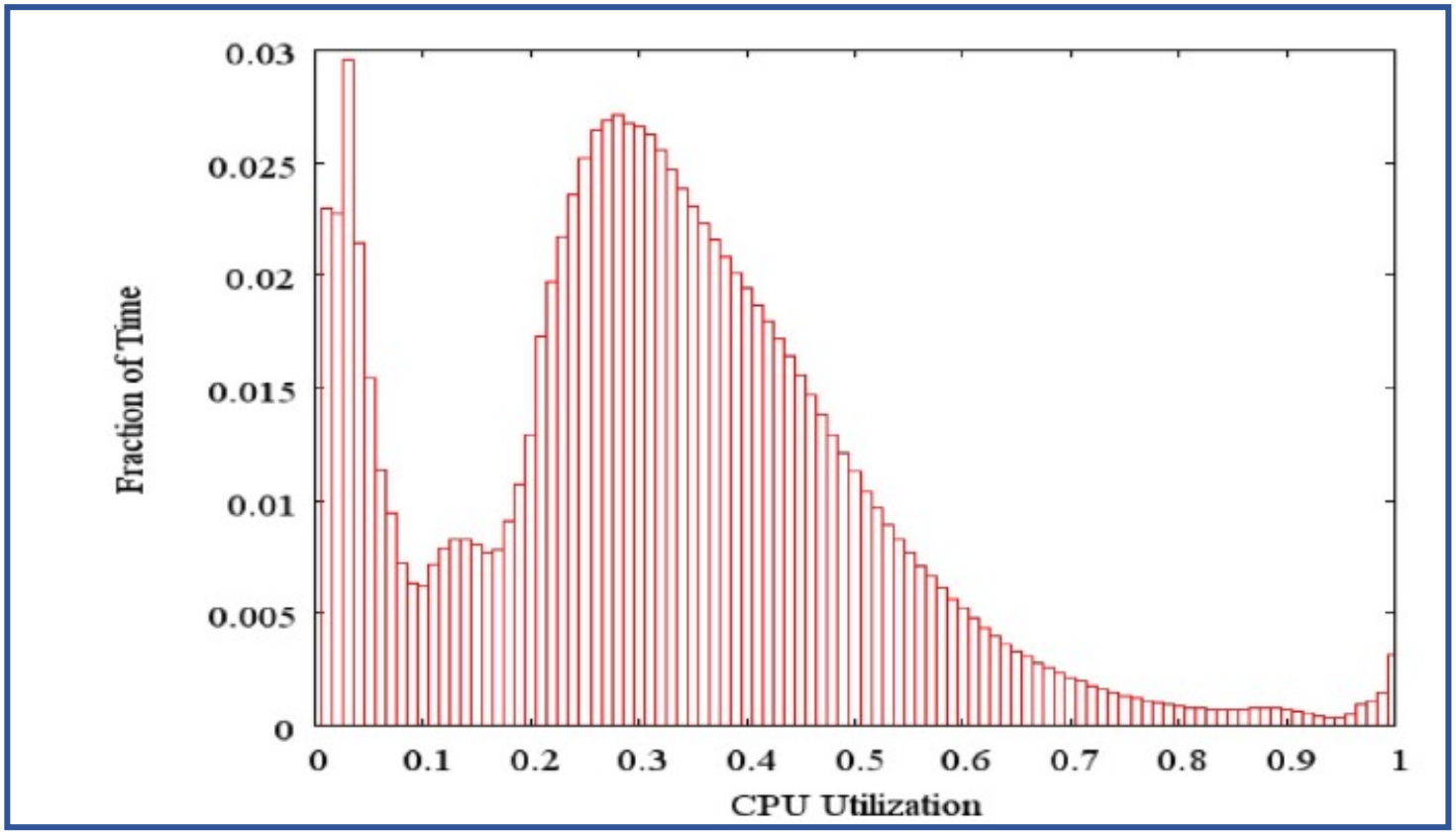
[1] The Future of the Data Centre, http://www.information-

提出 “Energy First” ⇔ 同样也是成本问题

利用率问题

数据中心资源闲置问题

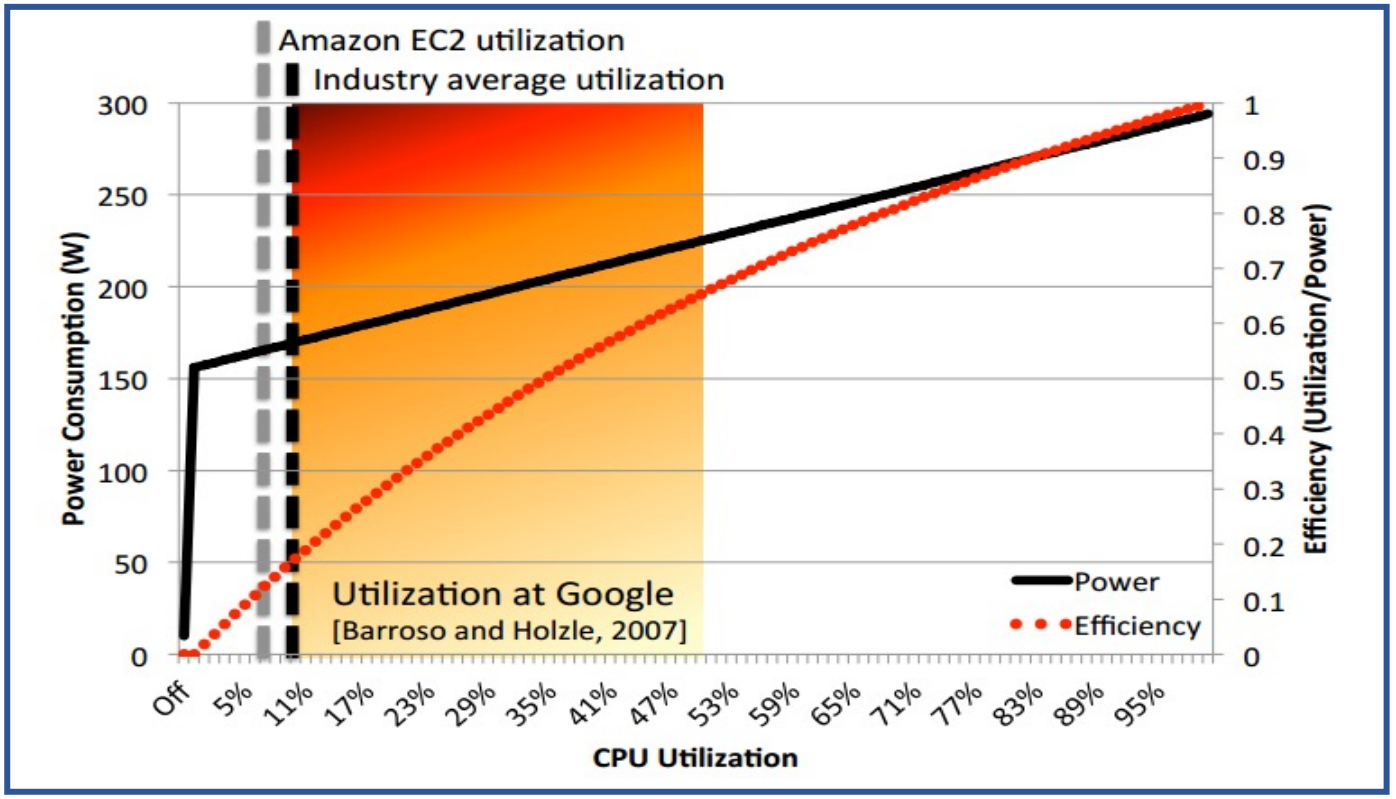
Google 5000台服务器半年的服务器负载分布



低使用率是许多数据中心面临的一个关键问题

利用率问题

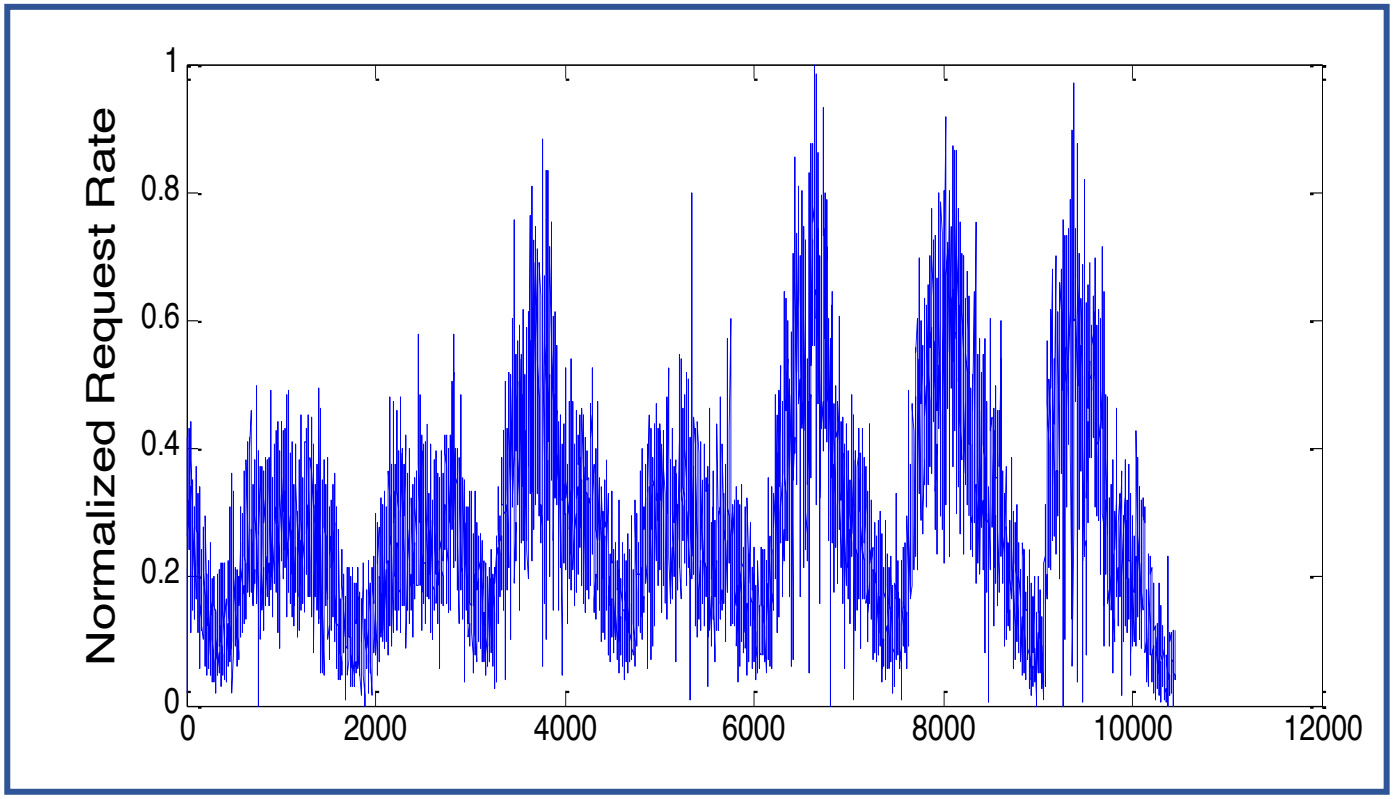
数据中心资源闲置问题



早期服务器利用率仅在30%-60%

利用率问题

数据中心资源闲置问题



早期服务器利用率仅在30%-60%

如何优化数据中心的资源利用？

优化资源利用的挑战

由于云基础设施的规模以及云系统与大量用户之间不可预测的交互，云资源管理非常具有挑战性，需要复杂的决策和多目标优化决策

外部因素

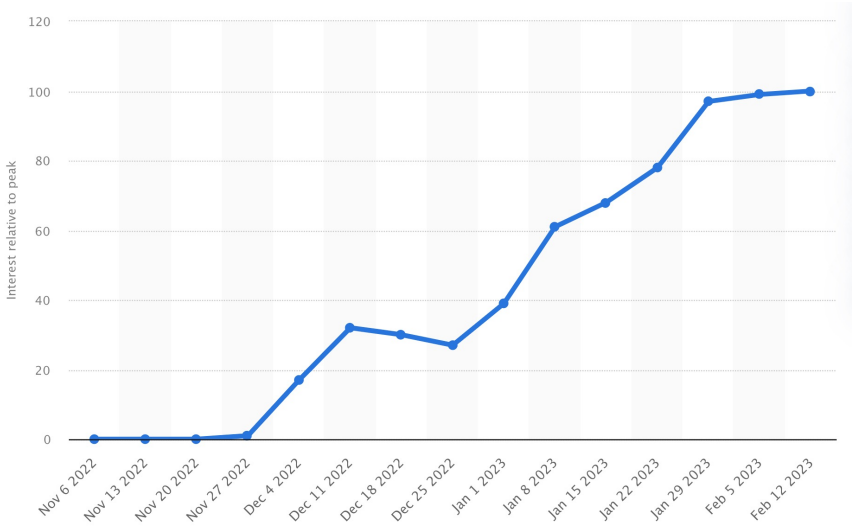
- ✓ 资源被超额订购
- ✓ 用户群体庞大，工作负载的类型和强度不可预测

内部因素

- ✓ 规模庞大，无法获得全局信息
- ✓ 系统软硬件的异构性
- ✓ 不同组件的故障率

优化资源利用的挑战

- 用户请求难以预测



- 过度配置资源

确保SLA

资源过度配置

经济损失

资源利用优化策略 – 弹性伸缩

资源弹性伸缩（Autoscaling）是一种云资源管理策略，它根据实际工作负载需求
动态调整计算资源的数量

资源利用优化策略 – 负载均衡

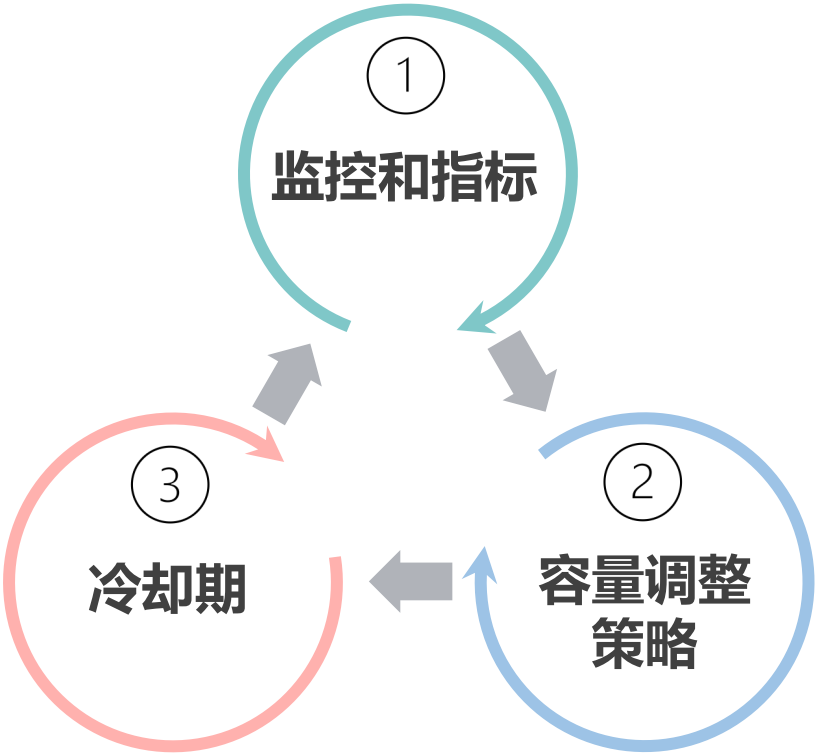
负载均衡（Load Balancing）是一种策略，
通过在多个计算资源之间分配工作负载，
实现性能优化和可靠性提升

资源弹性伸缩
Resource Autoscaling

资源弹性伸缩

根据实际工作负载需求**动态调整计算资源的数量**

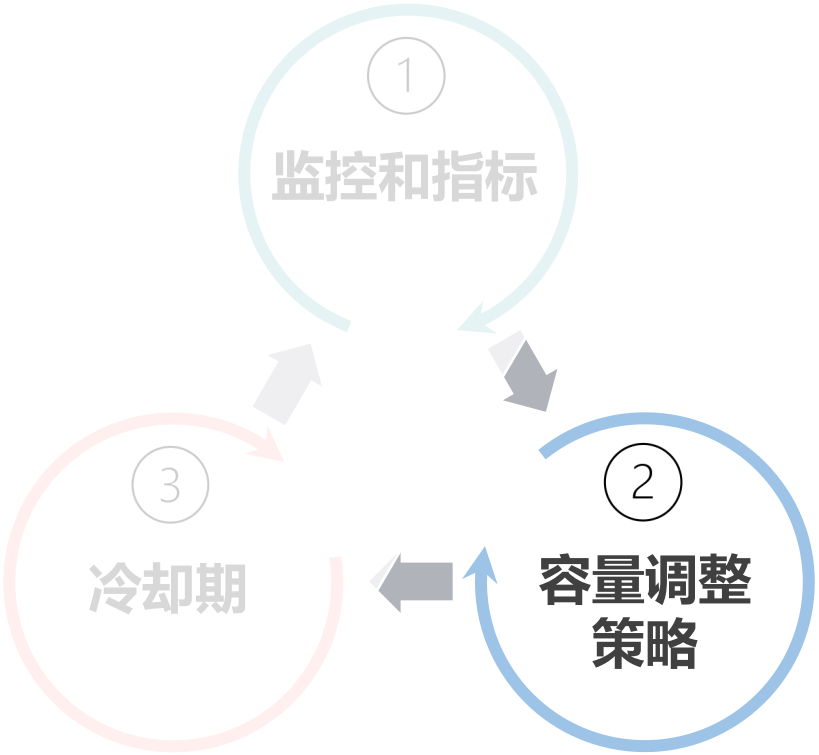
确保资源始终与需求匹配，**维持高性能**同时**降低资源浪费和成本**



监控和指标

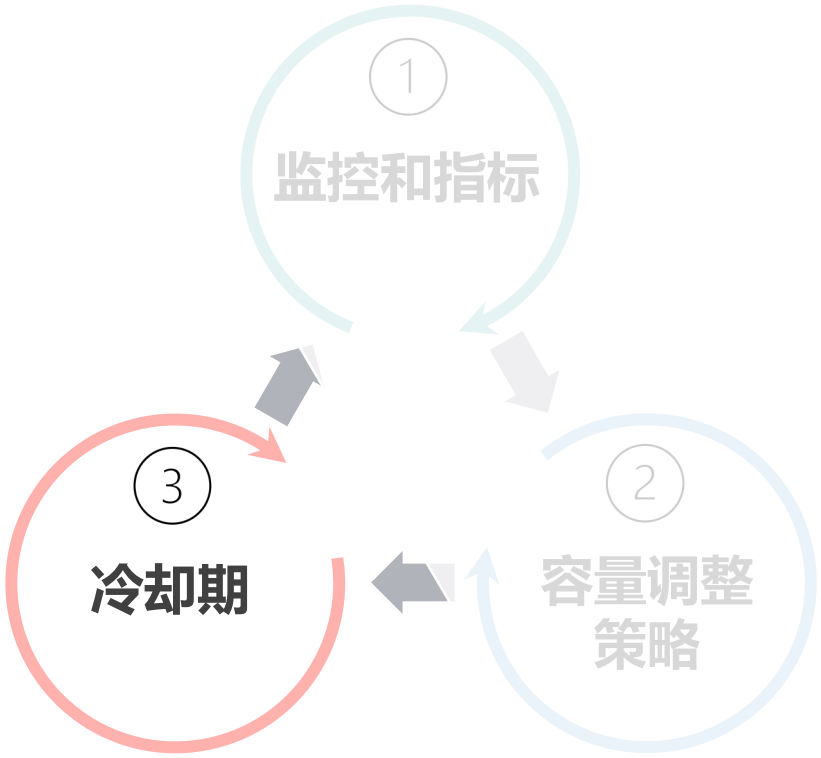
- 弹性伸缩依赖于实时监控和指标，以确定**何时需要增加或减少资源**
- CPU 利用率、内存使用情况、磁盘 I/O、网络流量等
- 通过收集和分析这些指标，**实时评估资源需求并调整**

资源弹性伸缩策略



- 资源弹性伸缩策略定义了**何时以及如何扩展或缩减资源**
- 策略可以是**固定的**，也可以根据时间和工作负载模式进行**动态调整**

冷却期



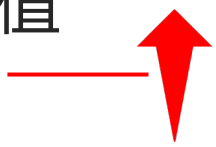
- 调整操作之后一段时间**不会再次触发相同类型的操作**
- 有助于**防止过度反应和资源波动**，确保已进行的调整操作有足够的时间生效

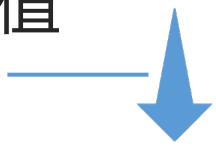
常见的资源扩缩容方法

- 1. 基于阈值
- 2. 负载预测
- 3. 强化学习

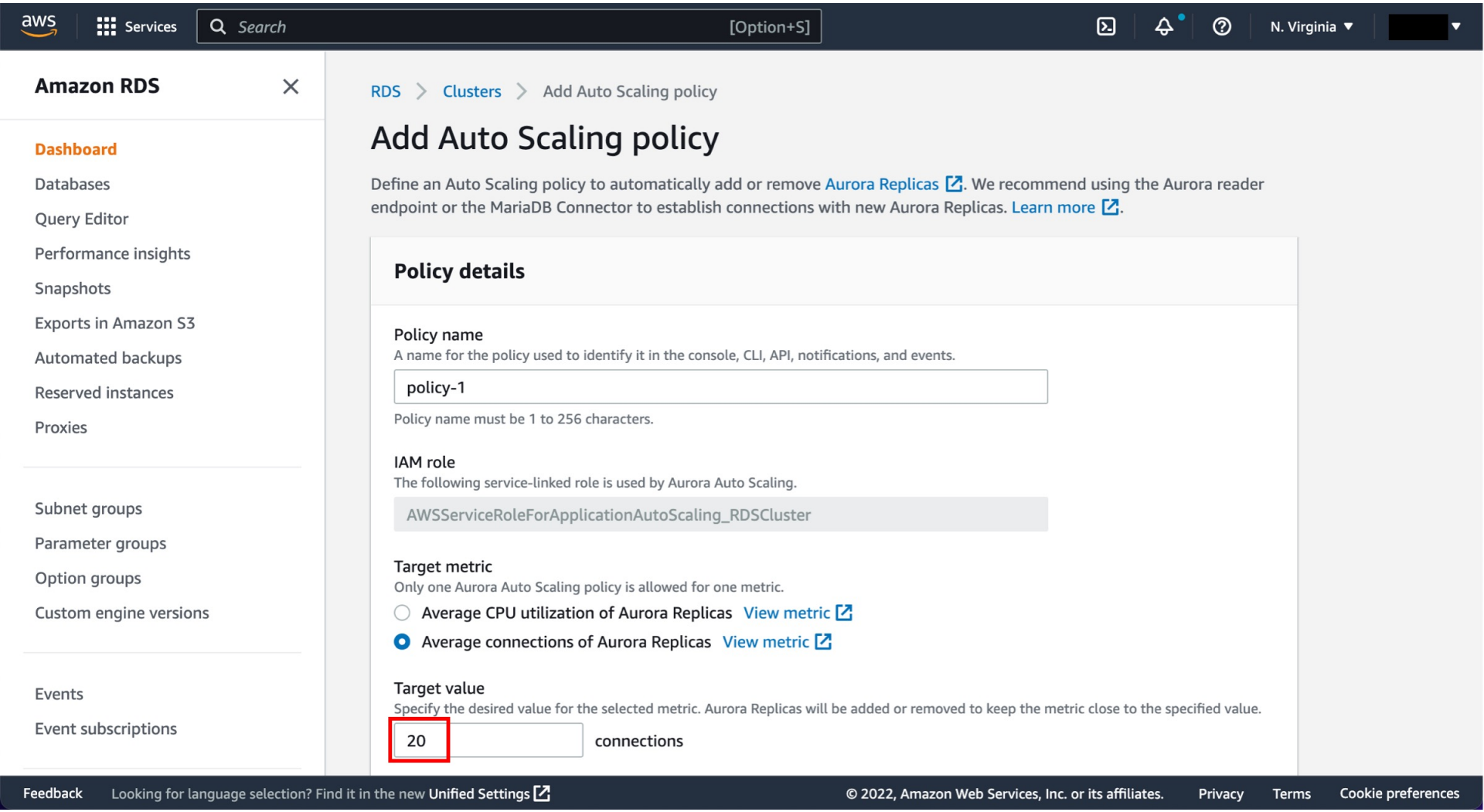
基于阈值的弹性伸缩

根据特定资源利用率指标（如 CPU 使用率、内存使用率）及其相应阈值自动调整资源

高阈值  超过高阈值，增加资源

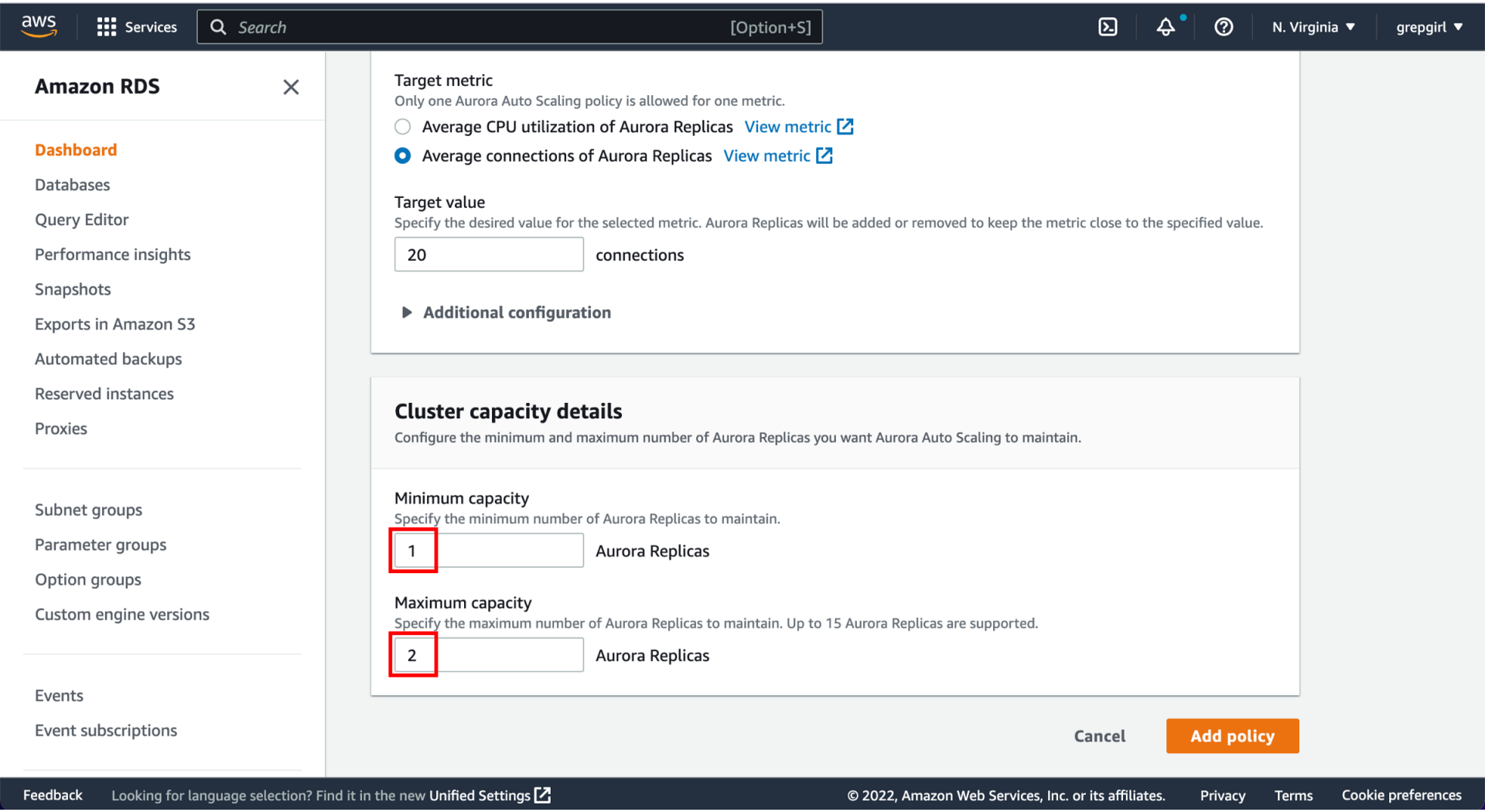
低阈值  低于低阈值，减少资源

AWS Auto Scaling 配置



https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Integrating.AutoScaling.Add.html

AWS Auto Scaling 配置



阈值的类型

□静态阈值

- ✓根据经验设定阈值并维持不变
- ✓必要时需进行人工动态调整

□动态阈值

- ✓在一定程度上克服了静态阈值的局限性
- ✓根据系统的实时负载和历史数据自动调整阈值
  - 比如一段时间内的指标的平均值

基于阈值的资源弹性伸缩有什么缺点?

基于预测的弹性伸缩

预测性扩缩容算法利用历史数据和机器学习技术预测未来资源需求，实现更快的资源调整

实现更快的资源调整，提供更高效的资源分配和成本节省

基于负载预测的资源扩缩容

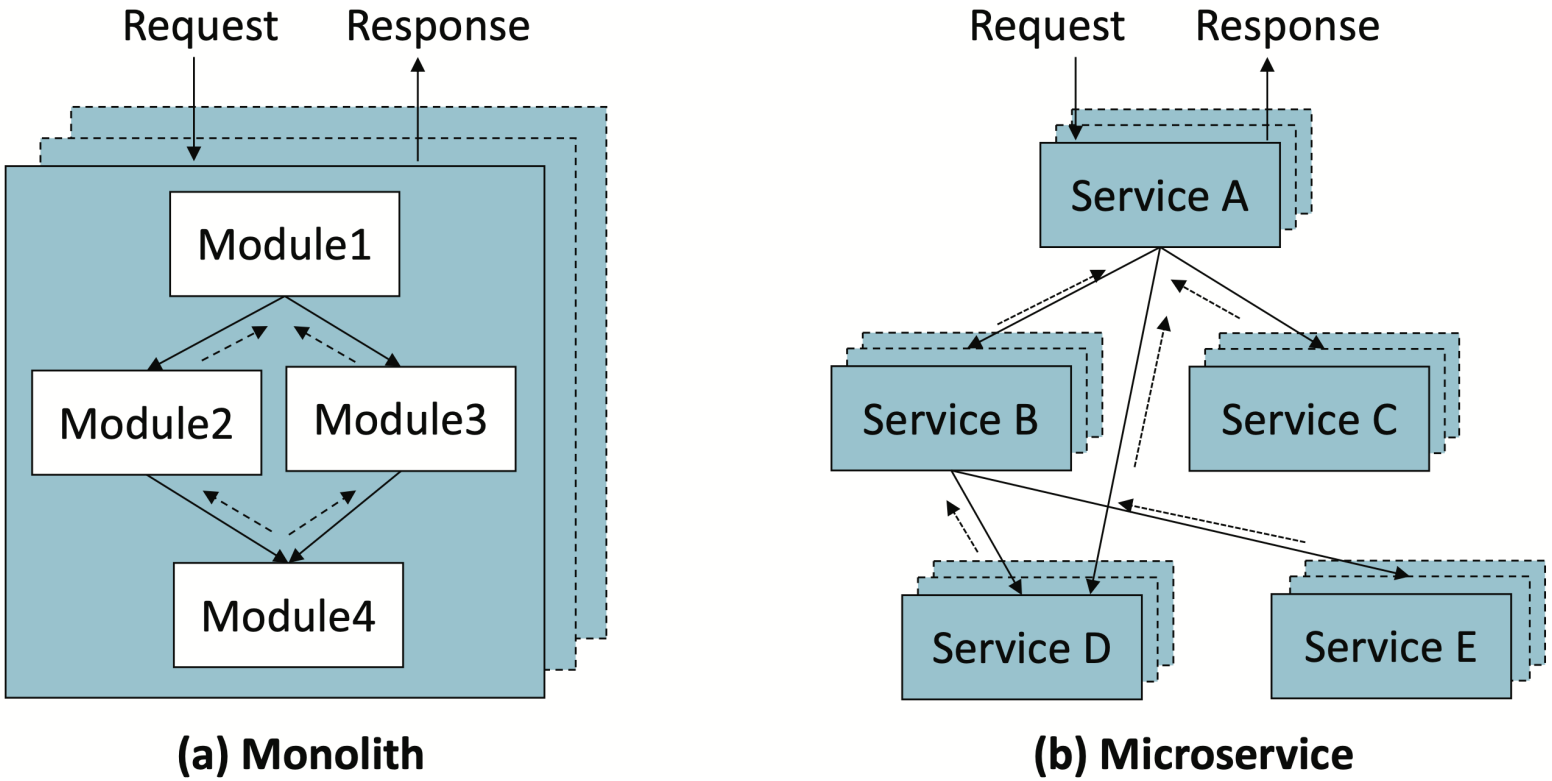
DeepScaler: Holistic Autoscaling for Microservices Based on Spatiotemporal GNN with Adaptive Graph Learning

Chunyang Meng[†], Shijie Song[†], Haogang Tong[†], Maolin Pan[‡] and Yang Yu[‡,*]
[†]*School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China*
[‡]*School of Software Engineering, Sun Yat-sen University, Guangzhou, China*
[*]*Author to whom correspondence should be addressed*
Email: {mengchy3, songshj6, tonghg}@mail2.sysu.edu.cn, {panml, yuy}@mail.sysu.edu.cn

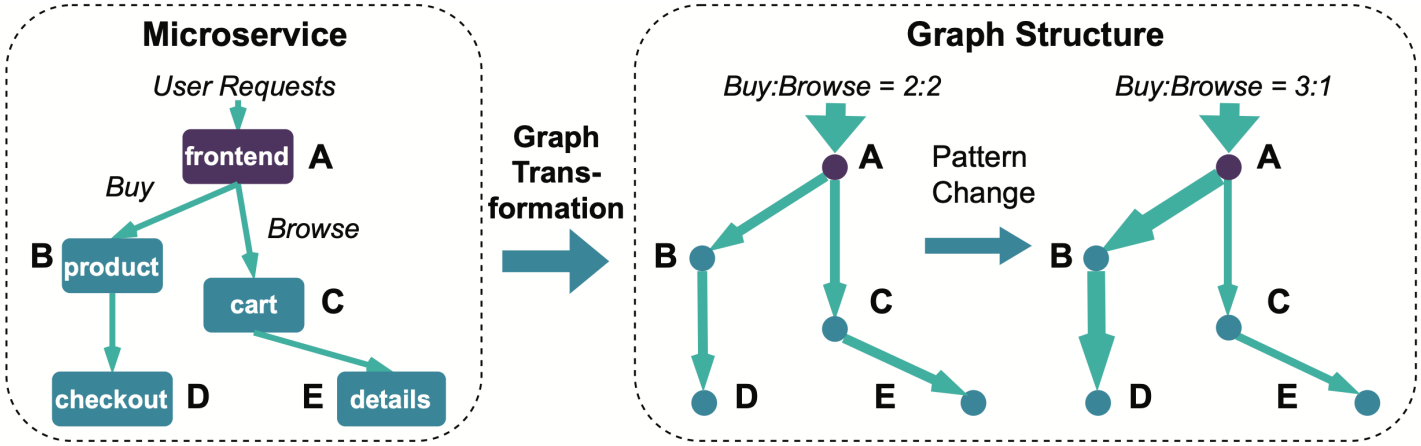
2023 ACM SIGSOFT 杰出论文奖

Ment et al. *DeepScaler*: Holistic Autoscaling for Microservices Based on Spatiotemporal GNN with Adaptive Graph Learning. ASE '23

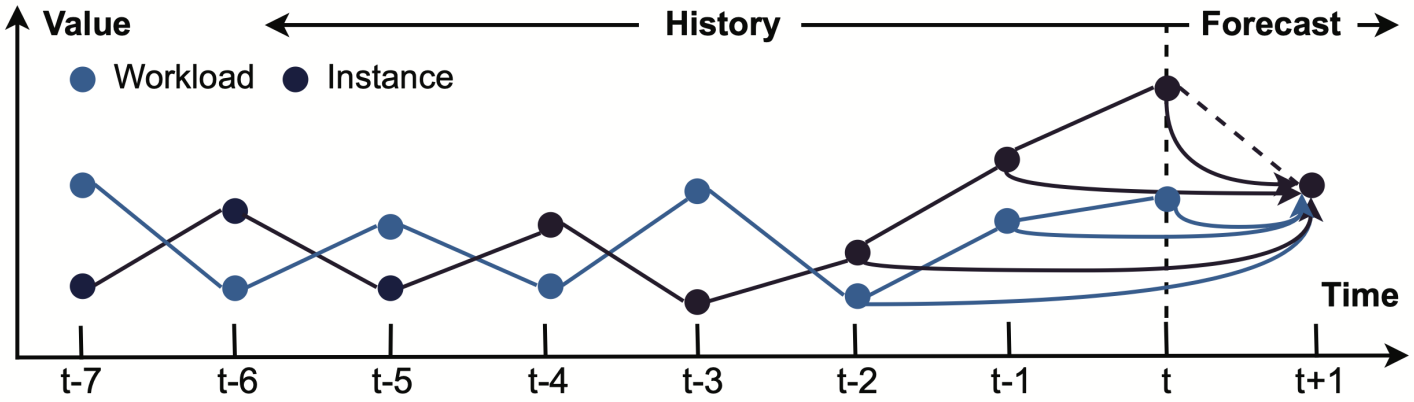
单体软件架构 vs. 微服务架构



微服务系统中的动态时空关系

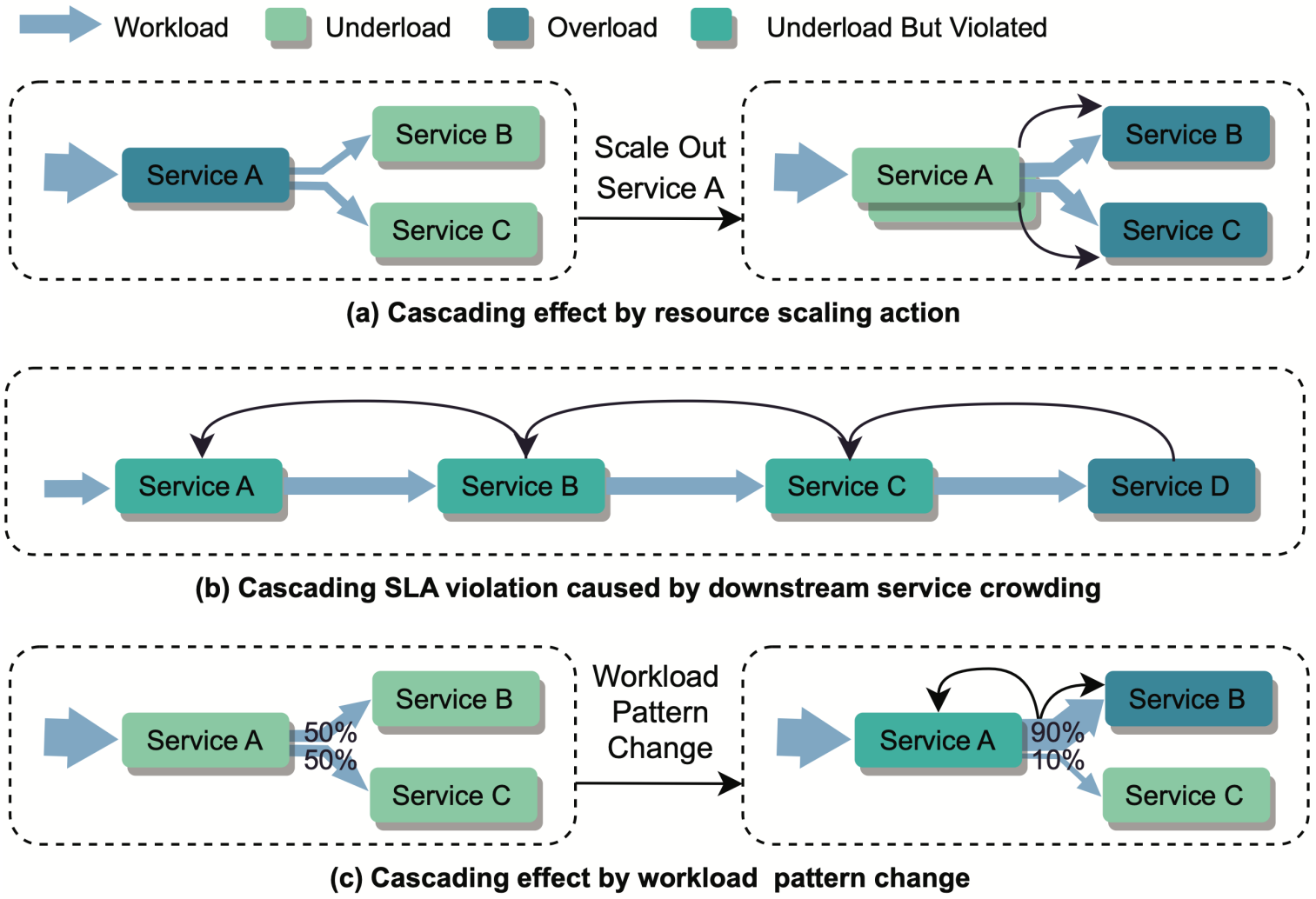


(a) Dynamic spatial features in microservice



(b) Dynamic temporal features over time

微服务系统中的级联效应

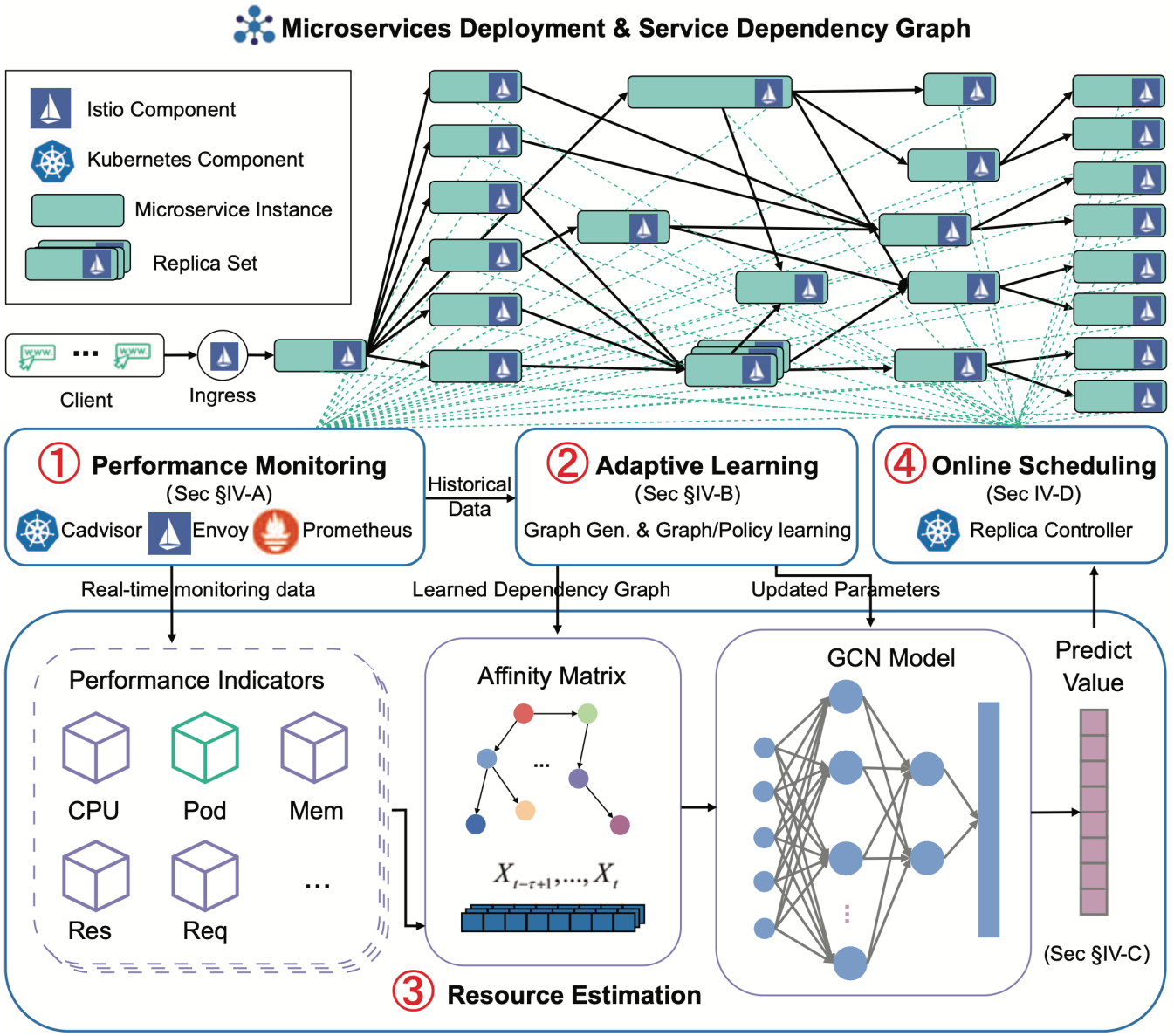


(a) Cascading effect by resource scaling action

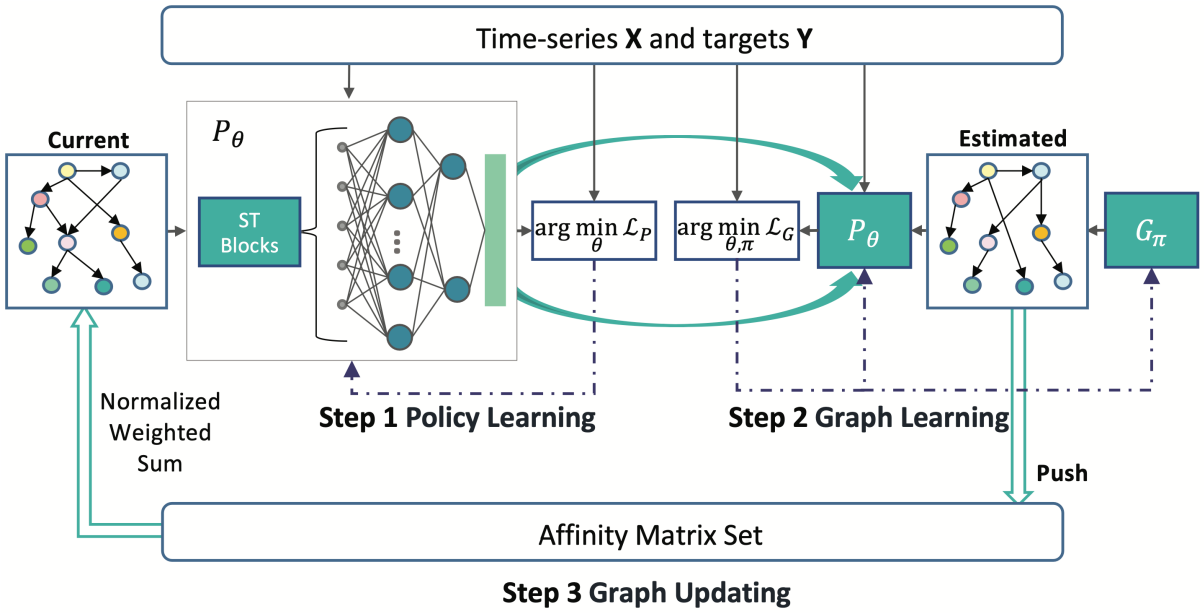
(b) Cascading SLA violation caused by downstream service crowding

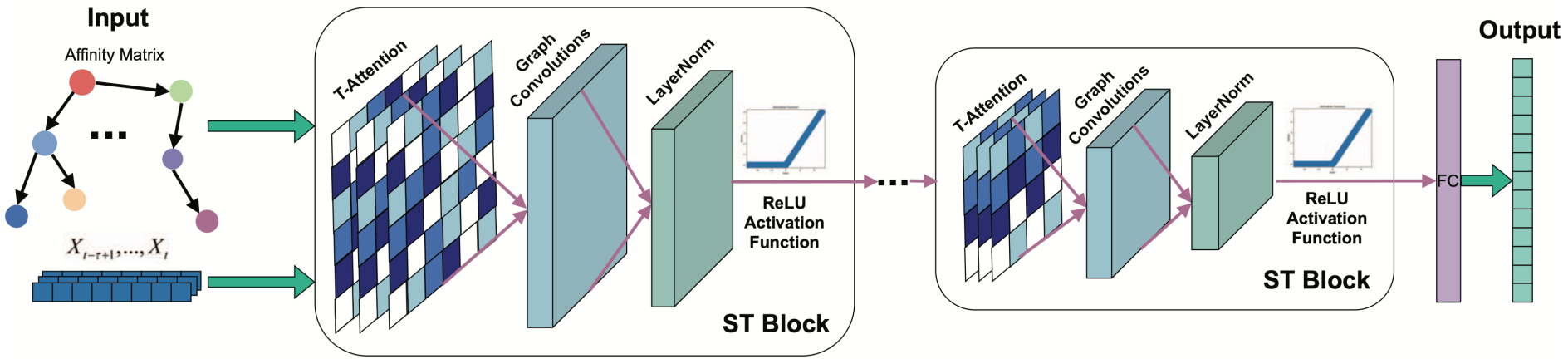
(c) Cascading effect by workload pattern change

微服务负载预测系统概览

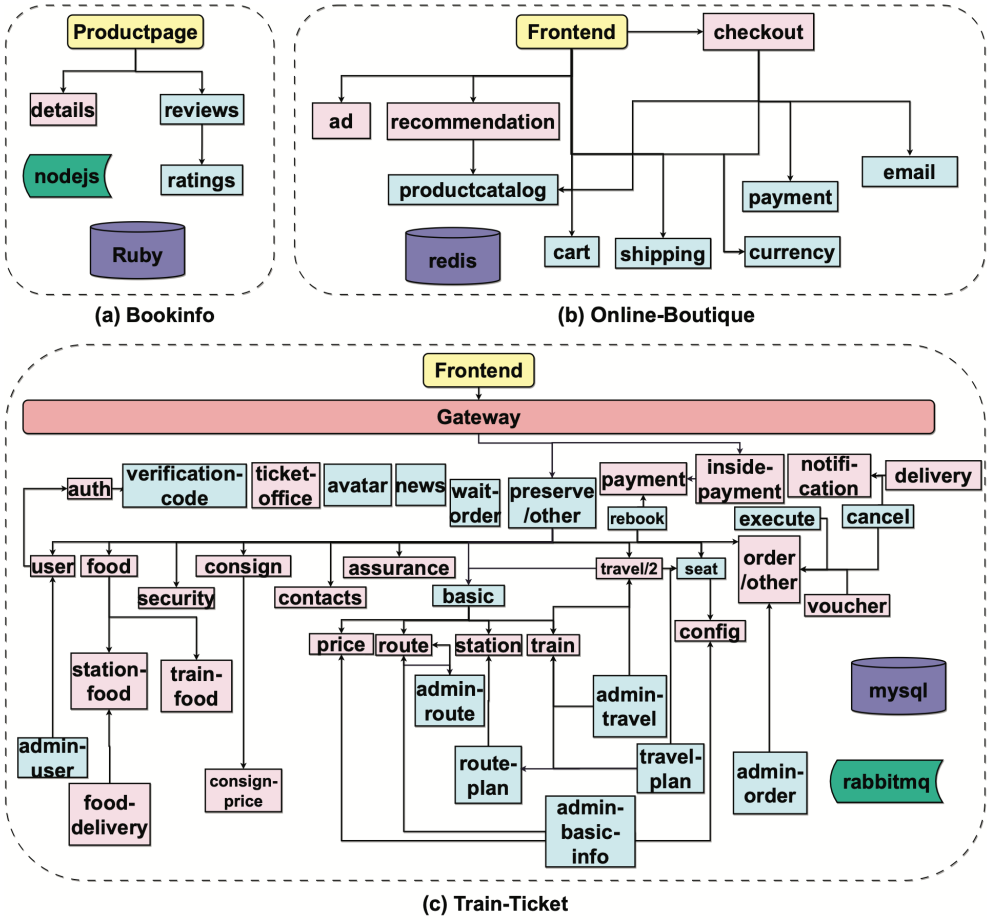


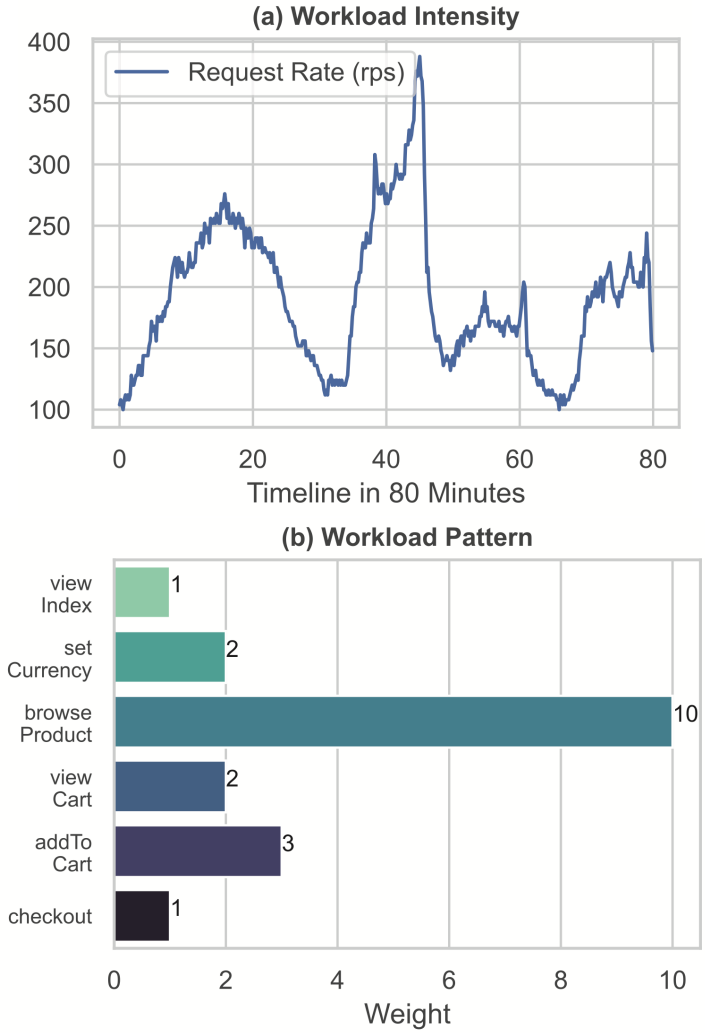
GNN 模型


Time-series X and targets Y
Current
P_θ
ST Blocks
$\arg\min_\theta \mathcal{L}_P$
$\arg\min_{\theta,\pi} \mathcal{L}_G$
P_θ
Estimated
G_π
Normalized Weighted Sum
Step 1 Policy Learning
Step 2 Graph Learning
Push
Affinity Matrix Set
Step 3 Graph Updating


Input
Affinity Matrix
$X_{t-\tau+1},\ldots,X_t$
T-Attention
Graph Convolutions
LayerNorm
ReLU Activation Function
ST Block
T-Attention
Graph Convolutions
LayerNorm
ReLU Activation Function
ST Block
FC
Output

实验设置

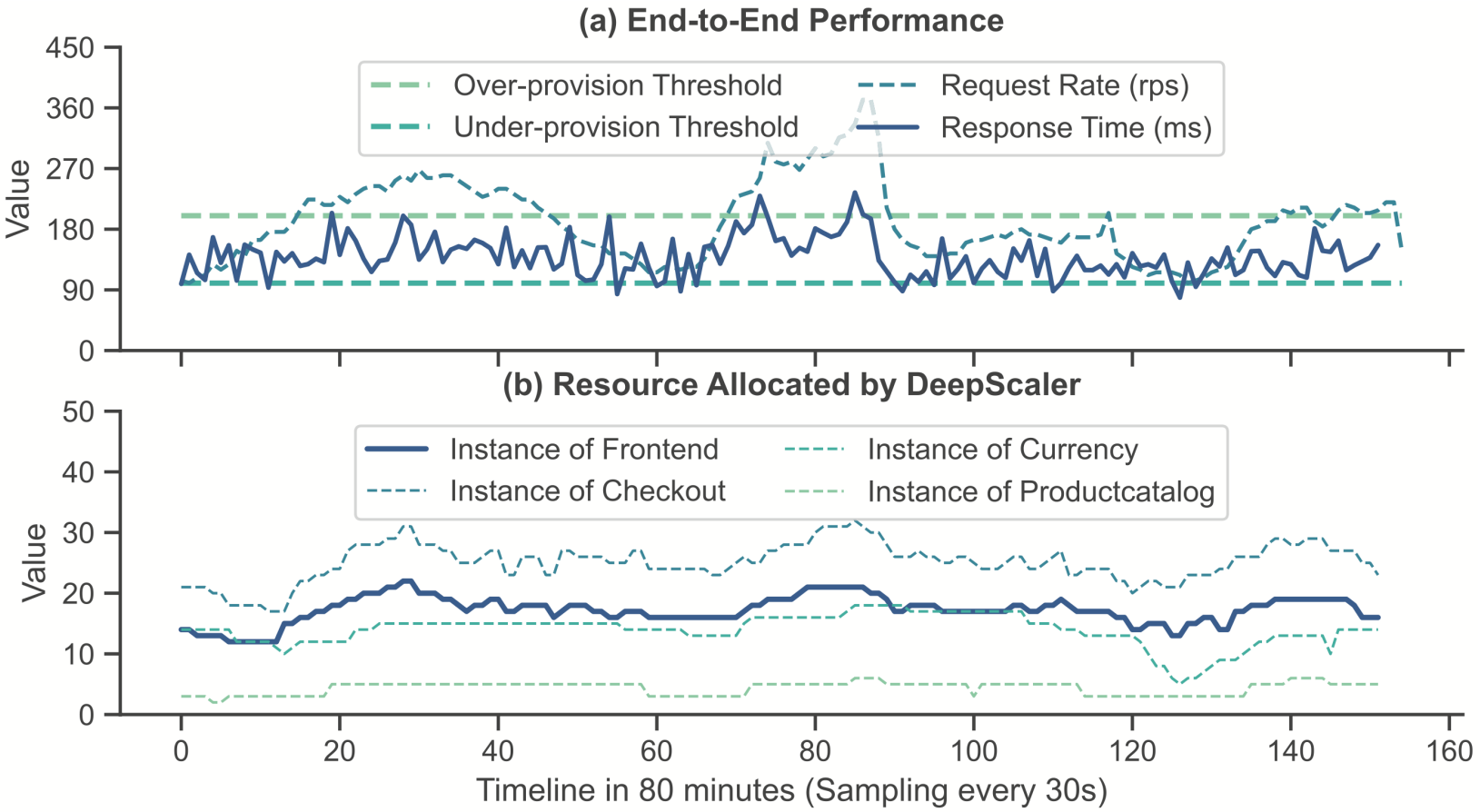




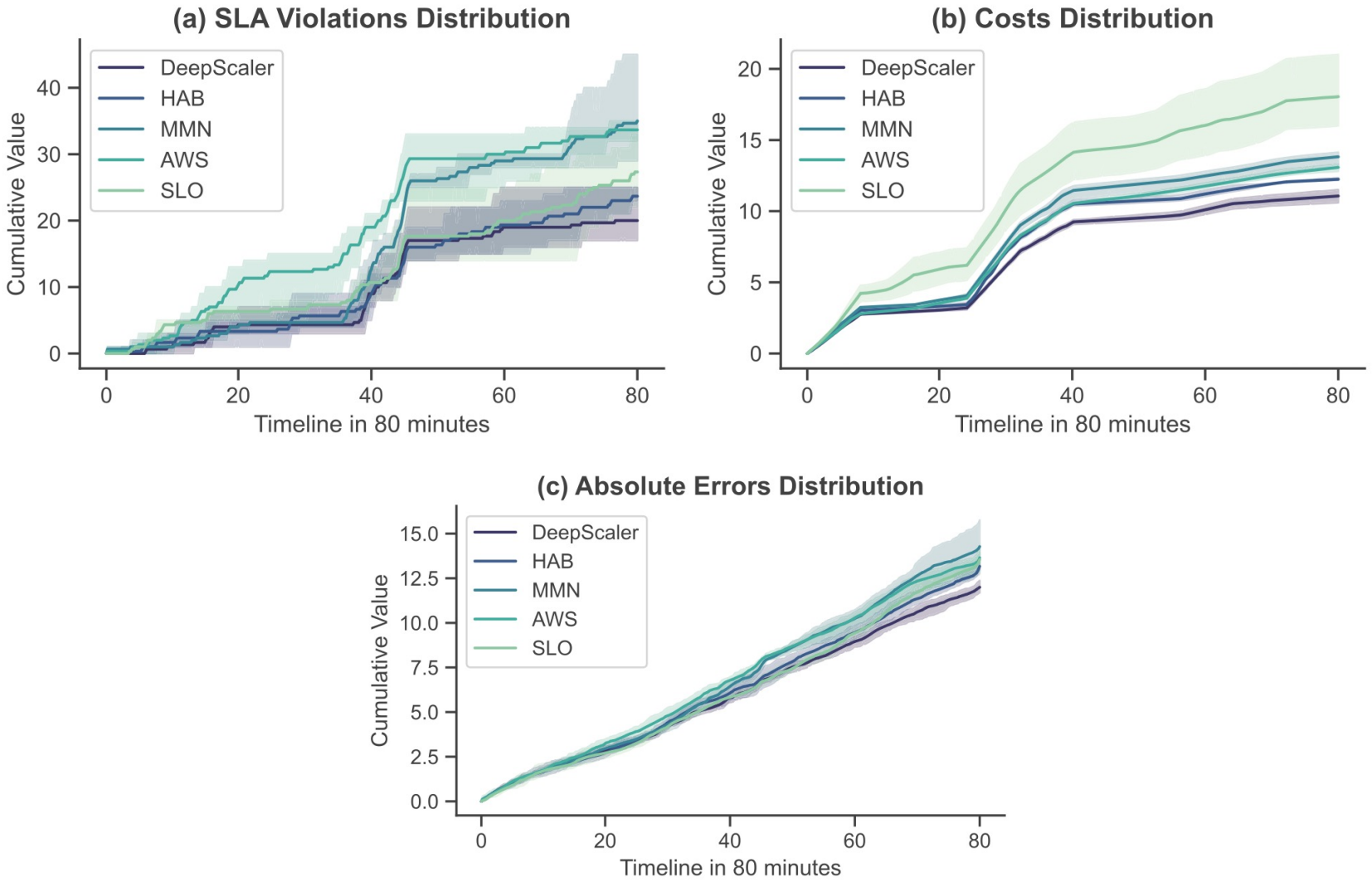
微服务基准系统

工作负载

实验例子


(a) End-to-End Performance
Over-provision Threshold
Under-provision Threshold
Request Rate (rps)
Response Time (ms)
Value
450
360
270
180
90
0
(b) Resource Allocated by DeepScaler
Instance of Frontend
Instance of Checkout
Instance of Currency
Instance of Productcatalog
Value
50
40
30
20
10
0
0
20
40
60
80
100
120
140
160
Timeline in 80 minutes (Sampling every 30s)

实验结果


(a) SLA Violations Distribution
DeepScaler
HAB
MMN
AWS
SLO
Cumulative Value
Timeline in 80 minutes
(b) Costs Distribution
DeepScaler
HAB
MMN
AWS
SLO
Cumulative Value
Timeline in 80 minutes
(c) Absolute Errors Distribution
DeepScaler
HAB
MMN
AWS
SLO
Cumulative Value
Timeline in 80 minutes

基于强化学习的资源扩缩容

Horizontal and Vertical Scaling of Container-based Applications using Reinforcement Learning

Fabiana Rossi, Matteo Nardelli, Valeria Cardellini
Dept. of Computer Science and Civil Engineering, University of Rome Tor Vergata, Italy
{f.rossi,nardelli,cardellini}@ing.uniroma2.it

Rossi, et al. Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning. CLOUD 2019.

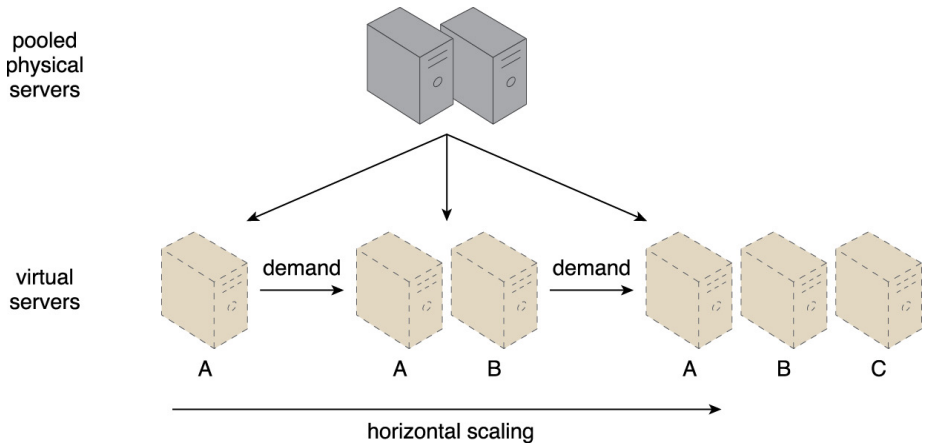
背景

- 云服务供全球用户访问，用户数量庞大且访问模式多样

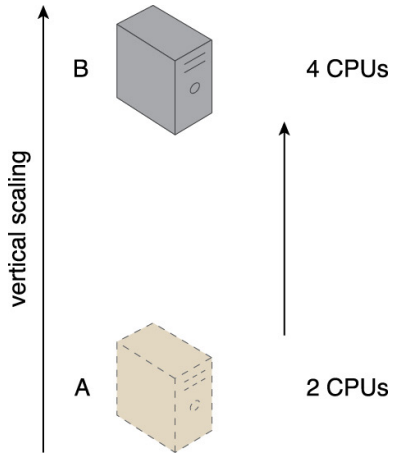
- 容器在云计算环境中表现出卓越的弹性和伸缩性，能有效应对不断变化的访问负载模式

- 现有的容器扩缩容策略将**垂直扩展**和**水平扩展**视为两种独立的手段，**无法达到极致表现**

资源扩缩容措施



水平扩展：增加虚拟机/服务器的数量



垂直扩展：迁移到更强大的服务器或增加虚拟机占用CPU的时间

更加常用

目标

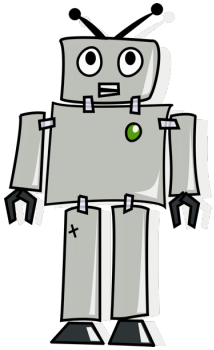
- 利用强化学习技术，结合水平和垂直扩展在运行时调整应用程序的容器资源

- 与基于阈值的方法不同，本文的目标是设计一种**灵活的方法**，可以定制自适应策略，**无需手动调整各种资源扩缩容手段**

强化学习流程

Find a policy f maximizing the total reward:

$$Action = f(Observation)$$



Actor

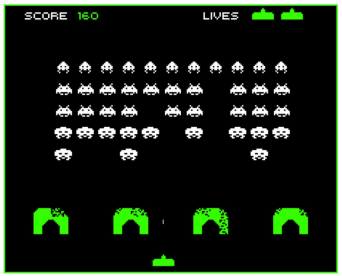
Observation

Function input

Action

Function output

Reward



Environment (Space invaders)

Space invaders 中的关键要素

☐Observation (state)

- ✓当前看到的画面

☐Action

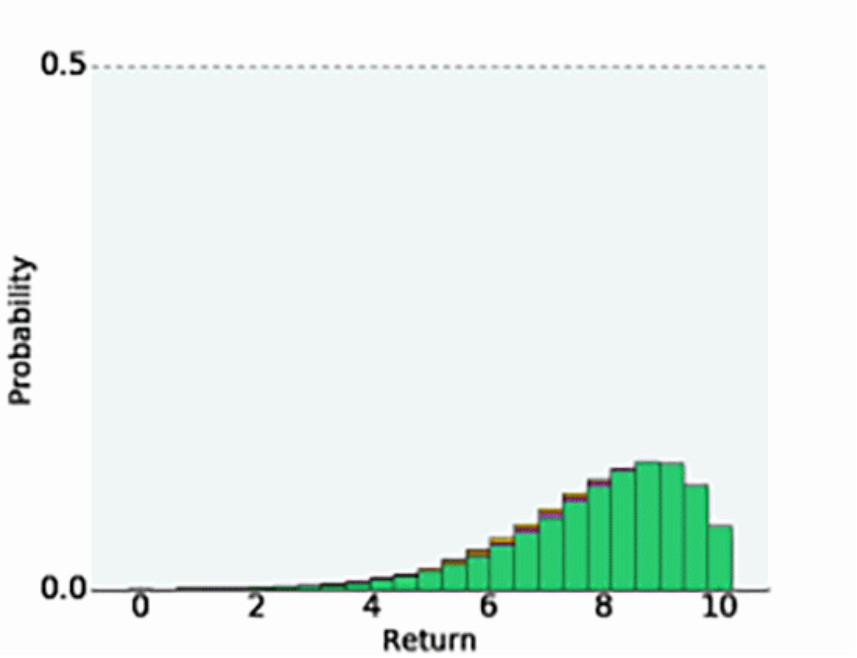
- ✓向左移
- ✓向右移
- ✓开火

☐Reward

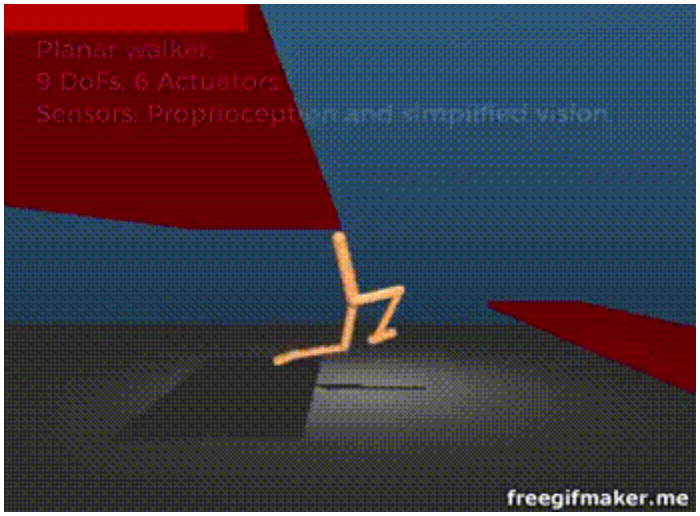
- ✓得到的分数
- ✓剩余生命数
- ✓时间
- ✓护盾损伤程度

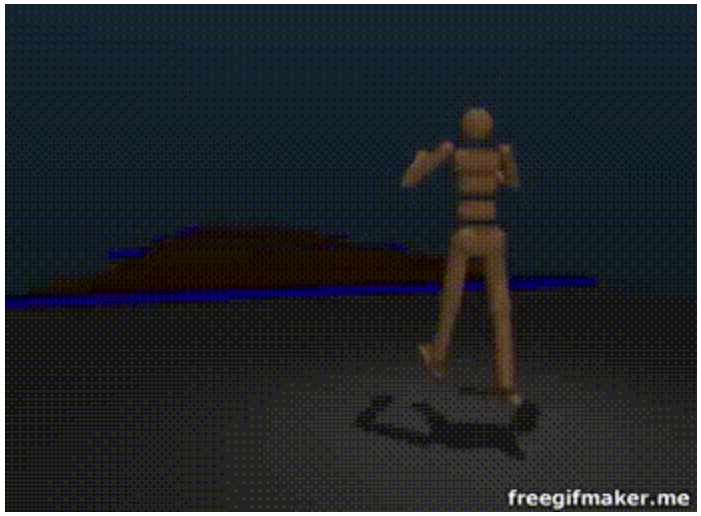
https://spaceinvaders.viperfish.com.au/

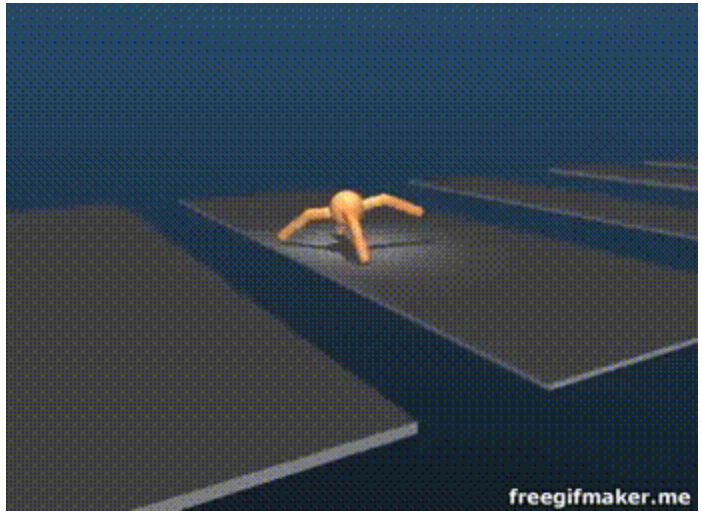
Space invaders运行过程



Robotics and Locomotion







资源弹性伸缩系统模型

作者考虑通用的应用程序模型，把应用程序当成是一个执行特定任务的黑盒实体（比如计算服务、数据库访问）

为了处理不断增加的工作负载，应用程序并行创建和执行多个容器。每个容器都是自主工作的，处理部分请求

强化学习

强化学习旨在通过与系统的直接交互来学习最佳资源调整策略

应用程序有一个 RL 代理，负责在运行时调整应用的资源，目的是最大限度地降低长期成本

强化学习

- RL 代理以离散的时间步长与应用程序进行交互
- 在每个时间步长，代理都会观察应用程序状态并执行一个操作
- 一个时间步长后，程序转换到一个新状态，产生相应的代价
- RL 代理学习一个 Q 函数，在应用状态为 s 时采取动作 a，以最小化期望代价 $Q(s, a)$

应用程序状态State

口应用程序在 i 时刻的状态定义如下

$$s_i = (k_i, u_i, c_i)$$

- ✓ k_i 为容器的数量
- ✓ u_i 为 CPU 利用率
- ✓ c_i 为每个容器所分配的 CPU 资源

定义 S 为所有可能的状态集合

资源调整动作Action

□在每一个状态 $s \in S$ 下，可进行某些资源调整动作，定义为

$$\mathcal{A}(s) \subseteq \mathcal{A}$$

$\mathcal{A}$ 为所有动作的集合： $\mathcal{A} = \{-1,0,+1\} \times \{-r,0,r\}$

- ✓ ± 1 为水平调节，即增加或减少一个容器
- ✓ $\pm r$ 为垂直调节，即增加或减少 r 个单位的 CPU 资源
- ✓ 0 表示不执行任何动作
- ✓ 集合乘积表示水平和垂直调节可同时进行

资源离散化

- CPU 利用率 u_i 和 CPU 资源 c_i 都是连续的，对其进行离散化

$$u_i \in \{0, \bar{u}, 2\bar{u}, \dots, L\bar{u}\}$$
$$c_i \in \{0, \bar{c}, 2\bar{c}, \dots, M\bar{c}\}$$

$\bar{u}$ 和 $\bar{c}$ 为合适的离散量

- 容器的数量不能无限增加，因此设置一个阈值 K_{max}

$$k_i \in \{1, 2, \dots, K_{max}\}$$

在特定状态下，有些动作是不能执行的，比如当 s 包含 $k = K_{max}$ 和 $c = M\bar{c}$ 时，可执行的动作集合为 $\mathcal{A}(s) = \{-r, -1, 0\}$

代价函数设计

- 当一个动作 a 被执行时，系统的状态从 s 变为 s'，此过程产生的代价记为函数 $c(s, a, s')$

- RL 代理从下列三个方面优化代价函数
  - ✓减少资源调整的次数
  - ✓保证应用程序的性能满足需求，即响应时间小于 R_{max}
  - ✓减少资源浪费

- 为不同目标设置代价
  - ✓调整资源的代价为 c_{adp}，仅需要为垂直调整设置，水平调整不需要（为什么？）
  - ✓当响应时间超过 R_{max}，产生代价 c_{perf}
  - ✓资源使用的代价 c_{res} 与容器的数量和 CPU 资源成正比

加权代价函数

口通过为不同类型的代价设置权重，表明其重要性

$$
\begin{aligned}
c(s,a,s') &= w_{\text{adp}} \frac{\mathbb{1}_{\{vertical-scaling\}} c_{\text{adp}}}{c_{\text{adp}}} + \\
&\quad + w_{\text{perf}} \frac{\mathbb{1}_{\{R(k+a_1,u',c+a_2)>R_{\max}\}} c_{\text{perf}}}{c_{\text{perf}}} + \\
&\quad + w_{\text{res}} \frac{(k+a_1)(c+a_2)c_{\text{res}}}{K_{\max} \cdot c_{\text{res}}} \\
&= w_{\text{adp}} \mathbb{1}_{\{vertical-scaling\}} + \\
&\quad + w_{\text{perf}} \mathbb{1}_{\{R(k+a_1,u',c+a_2)>R_{\max}\}} + \\
&\quad + w_{\text{res}} \frac{(k+a_1)(c+a_2)}{K_{\max}}
\end{aligned}
$$

其中 $\mathbb{1}_{\{.\}}$ 为指示性函数（indicator function）

$w_{adp}, w_{perf}, w_{res}$ 为权重， $w_{adp} + w_{perf} + w_{res} = 1$

优化算法

- Q-learning
- Dyna-Q
- Model-based reinforcement learning

系统实现：Elastic Docker Swarm

- 拓展 Docker Swarm 架构，实现 MAPE 控制环
  - ✓Monitor：收集有关应用程序和执行环境的数据
  - ✓Analyze：使用收集的数据来确定资源调整是否有益
  - ✓Plan：为应用程序确定一个资源调整计划
  - ✓Execute：执行上述资源调整计划

- 采用 master-worker 架构
  - ✓Master 运行 Monitor 和 Analyze
  - ✓Worker 运行 Plan 和 Execute

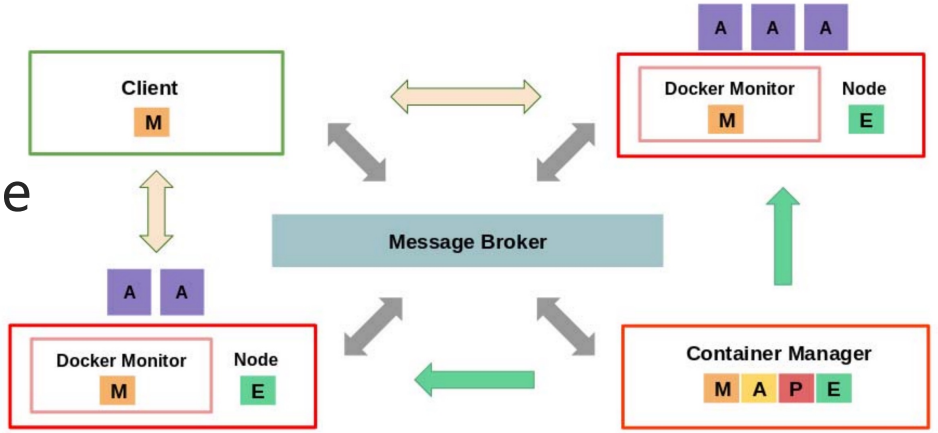


Fig. 1: Architecture of Elastic Docker Swarm

Workload

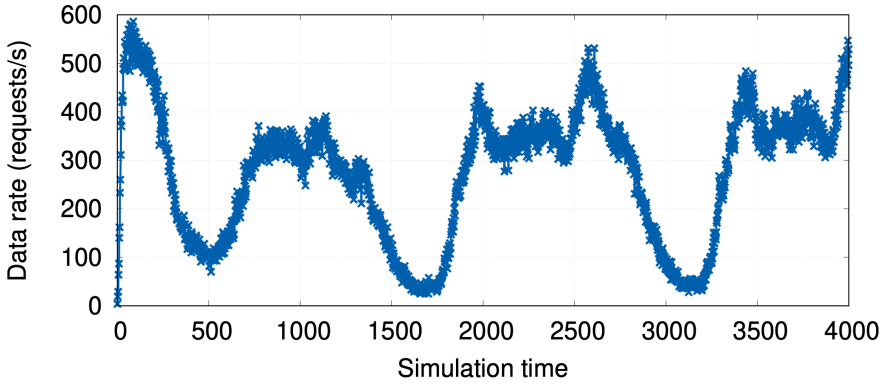


Fig. 2: Application workload used in simulation.

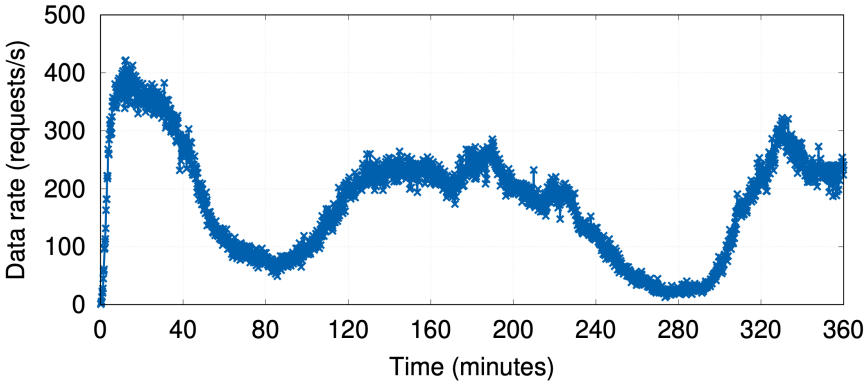


Fig. 3: Workload used in the prototype-based experiments.

实验结果

Weights	*Policy*	$R_{\max}$ *violations* (%)	*Average CPU utilization* (%)	*Average CPU share* (%)	*Average number of containers*	*Median R* (ms)	*Adaptations* (%)
$w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$	**Q-learning**	27.17	62.82	61.32	3.87	15.59	88.98
	Dyna-Q	25.77	63.39	62.60	3.56	15.29	92.53
	Model-based	**2.85**	60.73	87.43	2.57	10.15	37.32
$w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, $w_{\text{adp}} = 0.01$	**Q-learning**	61.18	80.95	46.62	3.08	$+\infty$	89.38
	Dyna-Q	62.58	81.89	46.32	3.70	$+\infty$	94.30
	Model-based	99.80	**99.85**	**11.04**	**1.09**	$+\infty$	2.95
$w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$	**Q-learning**	39.69	69.22	53.62	3.89	25.00	85.70
	Dyna-Q	35.64	65.08	54.61	4.35	20.53	91.10
	Model-based	**19.50**	**70.75**	78.35	2.56	15.16	**40.29**

算法运行过程

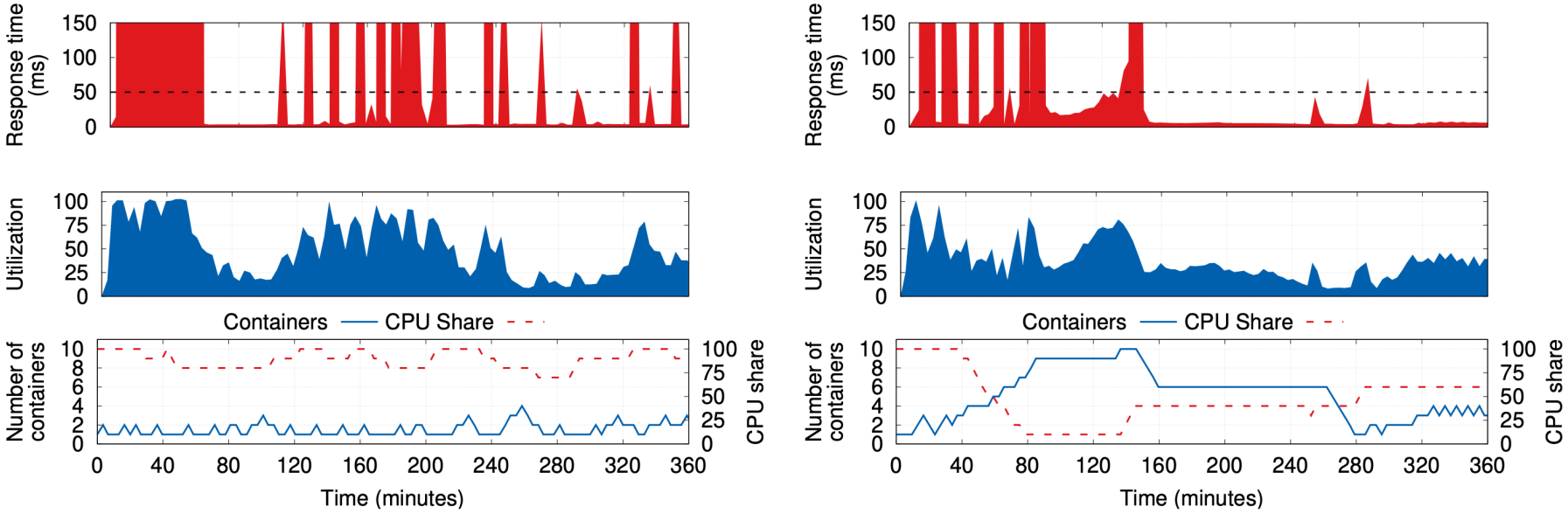
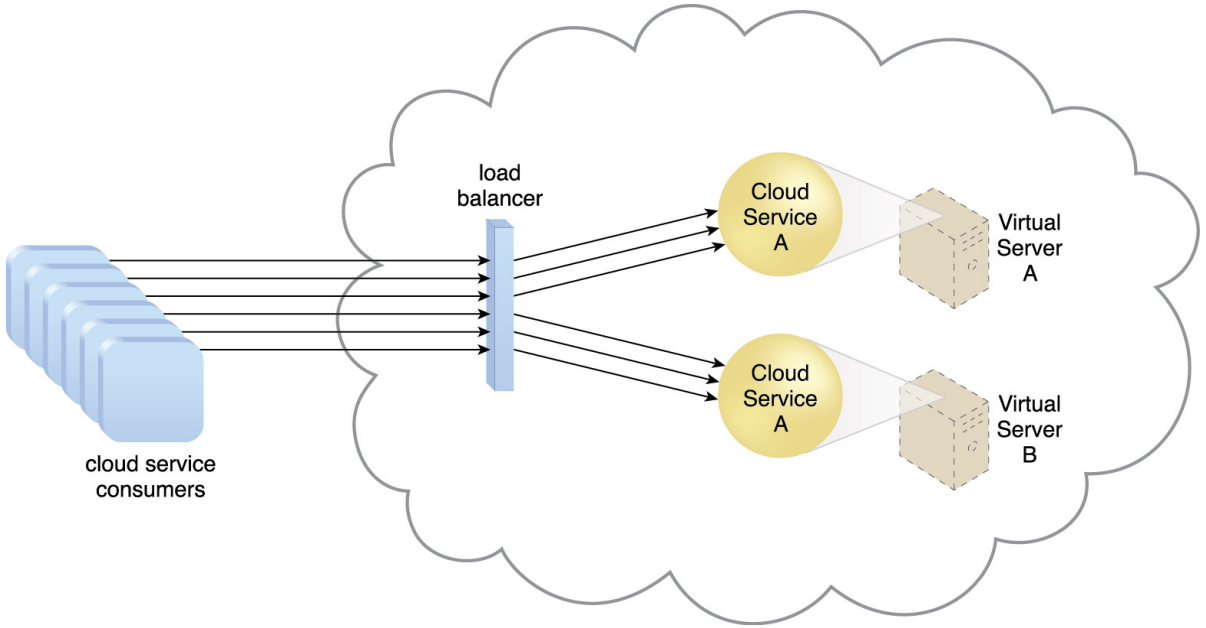


Fig. 4: Application performance using the 5-action adaptation model and weights $w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$.

負載均衡
Load Balancing

什么是负载均衡?

负载均衡（load balancing）**将流量分发到多个服务器或服务实例**来提高应用程序的可用性和可靠性，由云服务提供商或企业内部搭建



能耗与能效

- 是不是只需要考虑将工作负载尽量均衡就好？

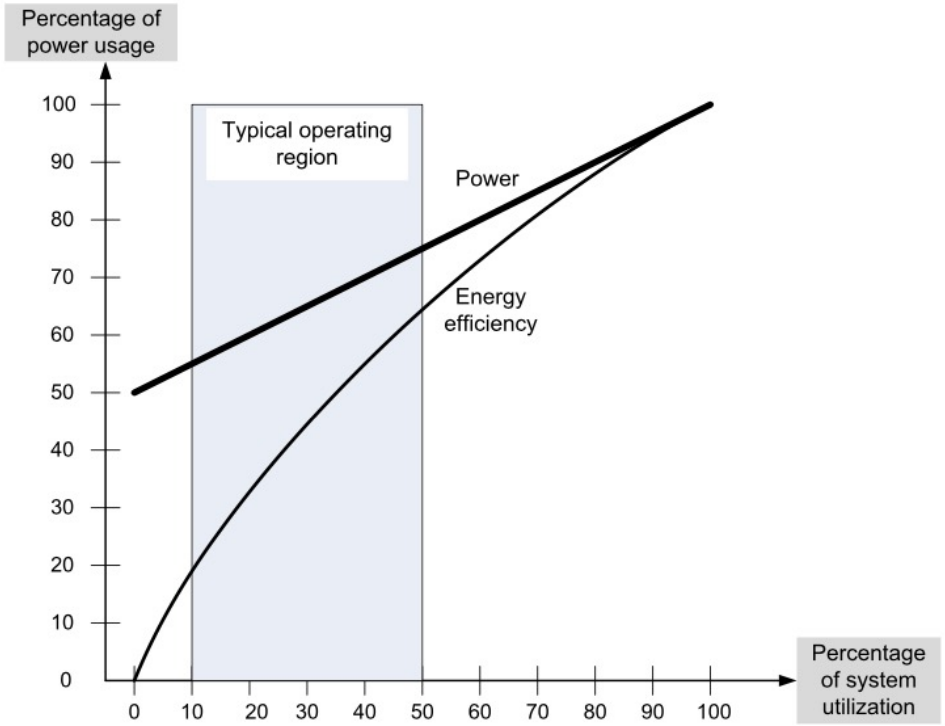


FIGURE 9.1

Even when power requirements scale linearly with the load, the energy efficiency of a computing system is not a linear function of the load. When idle, a system may use 50% of the power corresponding to the full load. Data collected over a long period of time shows that the typical operating region for the servers at a data center is the range 10% to 50% of system utilization [55].

负载均衡算法

静态负载均衡

□预先在负载均衡器中配置，不随着系统负载的变化而自动调整

- 随机算法（Random）：随机选择一个后端服务器来处理请求
- 轮询算法（Round Robin）： 将请求依次分配给后端服务器，每个服务器处理相同数量的请求
- 加权轮询算法（Weighted Round Robin）：与轮询算法类似，但是每个服务器有一个权重值，表示每个服务器处理请求的能力

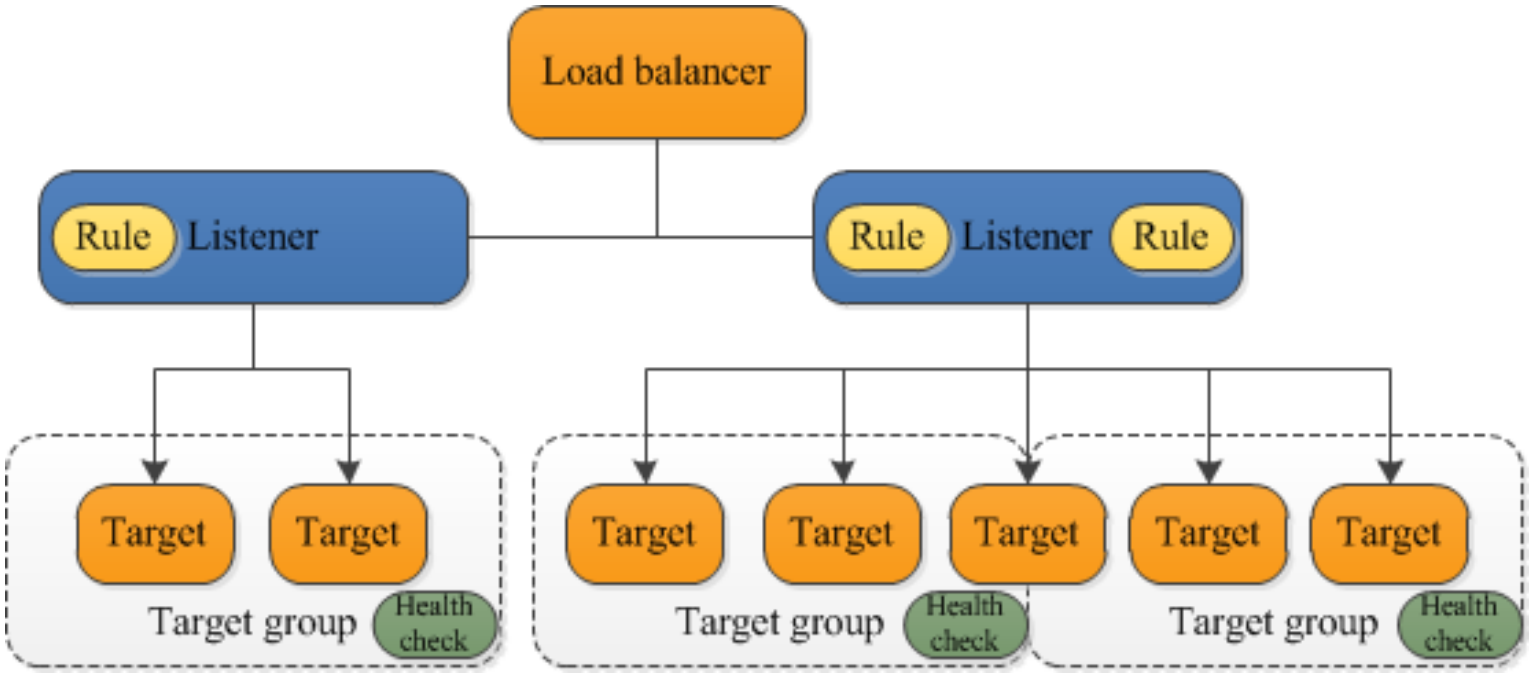
负载均衡算法

动态负载均衡

□实时监测后端服务器的负载情况动态地选择服务器来处理请求

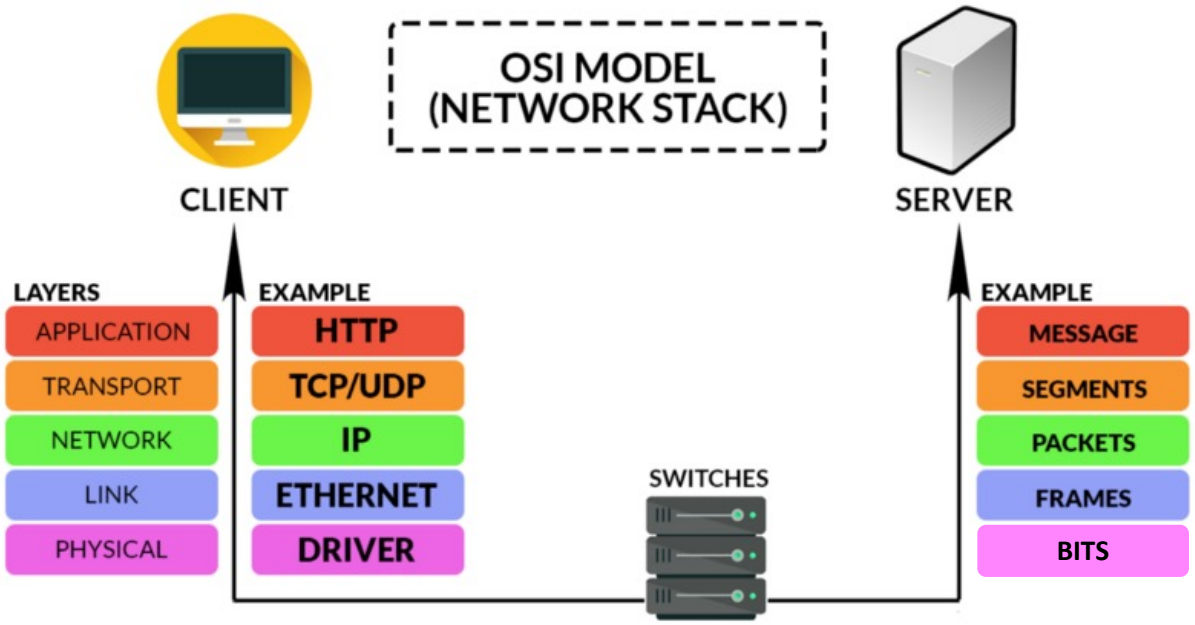
- 最小连接数算法（Least Connections）：将请求分配给当前连接数最少的服务器
- 最短响应时间算法（Shortest Response Time）：将请求分配给响应时间最短的服务器
- 基于哈希的算法（Hash-based）：根据请求的哈希值将请求分配给特定的服务器，确保相同请求总是被分配到同一个服务器上

负载均衡技术



Targets 可以是物理/虚拟服务器、容器和 IP 地址等，横跨单个或多个可用区

负载均衡的种类

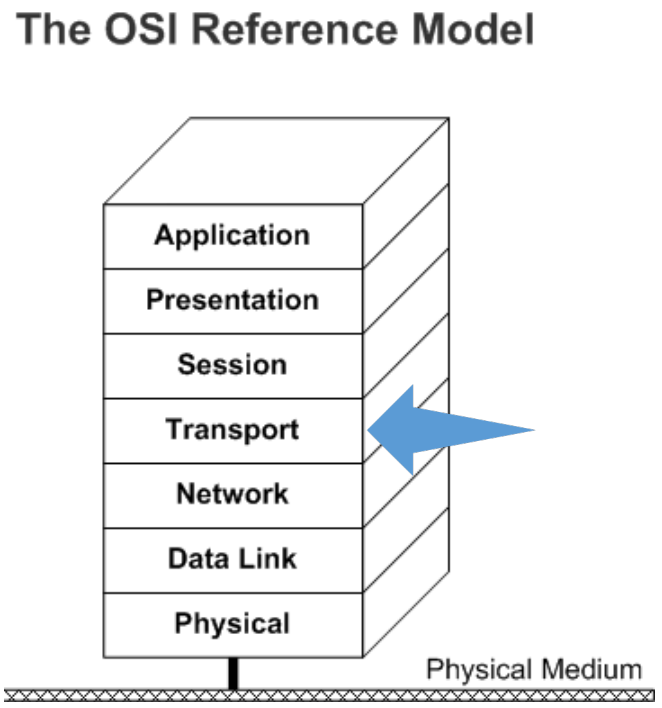


Listener 可以设置在整个请求链路中的不同节点，包括：

- 网络负载均衡（Network load balancing）
- 网关负载均衡（Gateway load balancing）
- 应用负载均衡（Application load balancing）
- ……

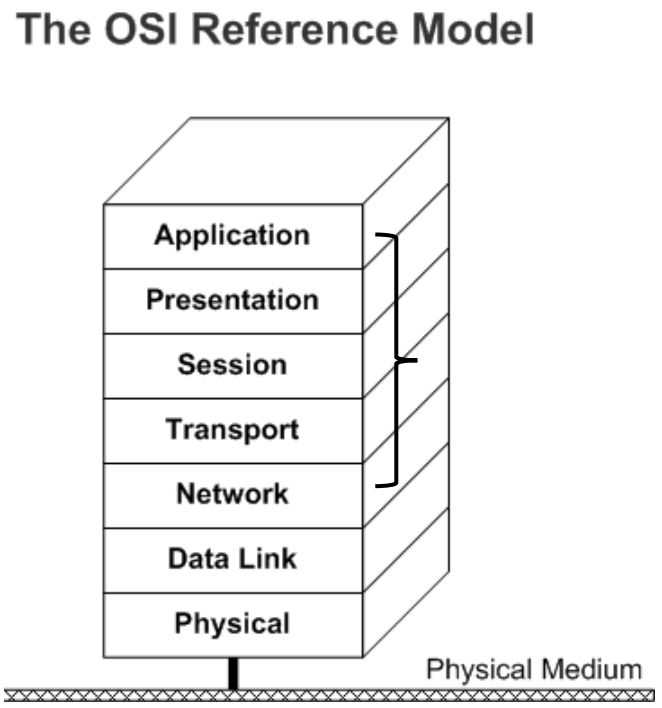
网络负载均衡

- 网络负载均衡主要针对网络流量进行平衡，在传输层（四层）上，如 TCP、UDP 协议
- 通过 IP 地址、端口号、协议、请求类型等进行负载均衡计算
- 延迟低，适合大规模、无状态的 Web 应用程序、数据库、DNS 等服务



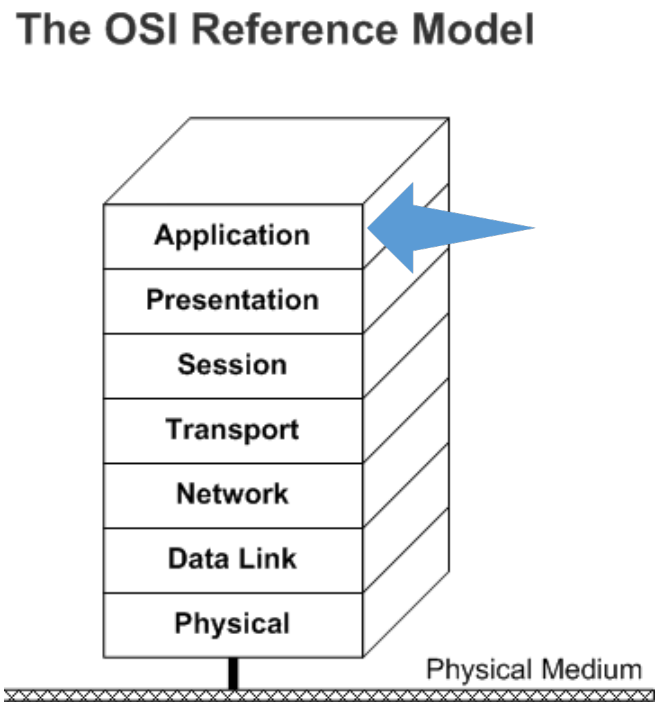
网关负载均衡

- 将负载均衡功能集成到网关设备中，可以处理第三层到第七层的流量
- 可通过硬件设备（如路由器、交换机、防火墙等）或软件（如代理服务器、网关服务器等）实现
- 根据源和目的 IP 地址进行负载均衡，有些网关也使用端口号



应用负载均衡

- 应用层负载均衡是一种将负载均衡应用到应用层（七层）的负载均衡方法
- 更精细地控制请求的负载均衡，可根据请求的内容、用户的地理位置、请求的优先级等
- 应用层负载均衡器通常在请求的首部或者主体中查找关键字，常见的应用层负载均衡协议包括 HTTP、HTTPS、SMTP 等



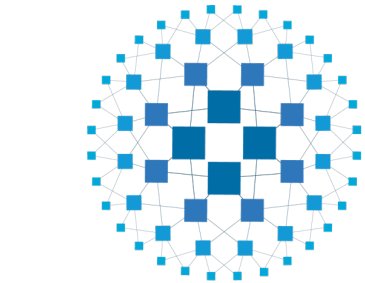
负载均衡的实现方式

硬件负载均衡器

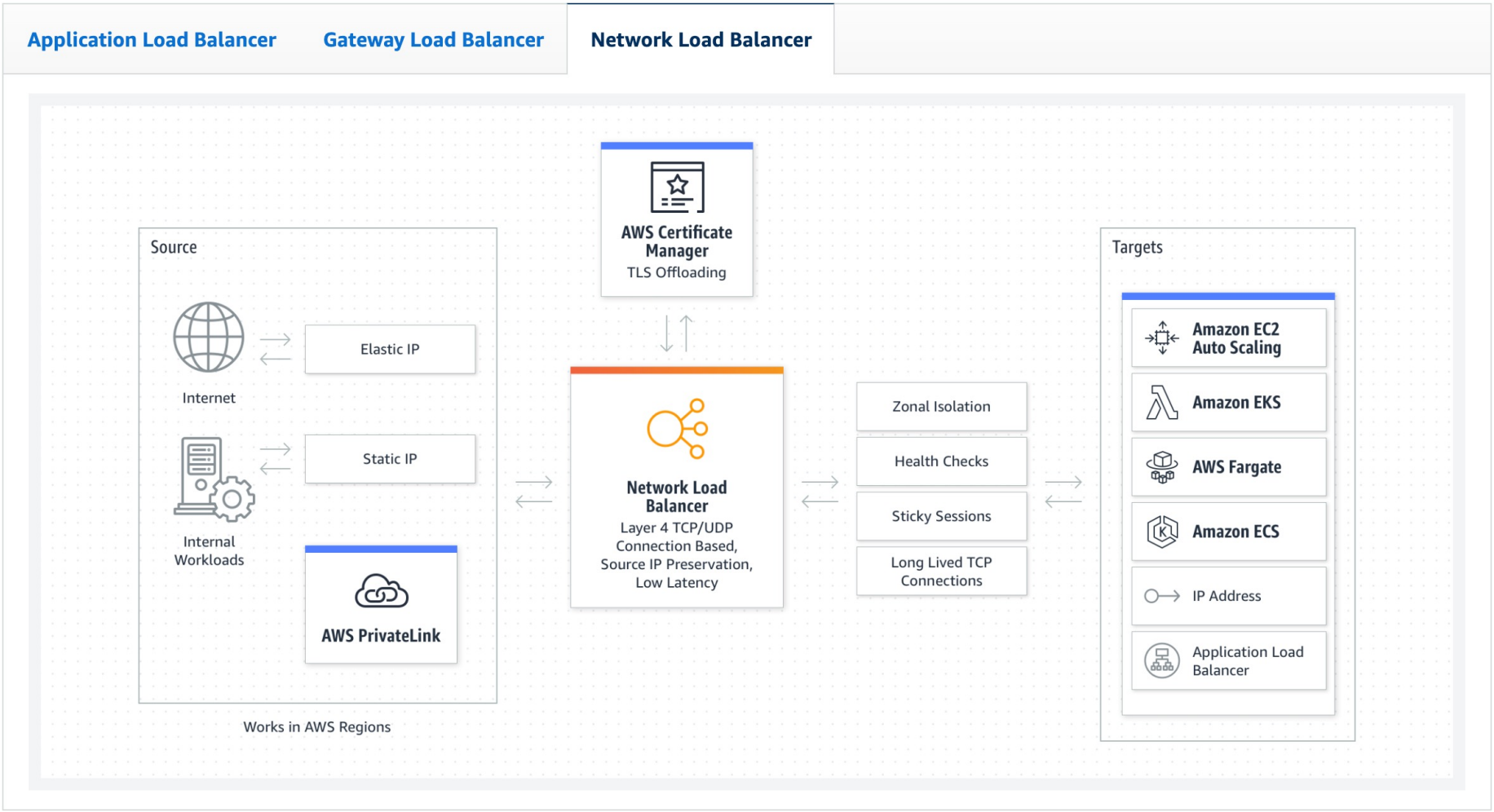
- 硬件负载均衡器是专用设备，提供高性能和高可靠性的负载均衡服务，但成本较高，灵活性较差。常见的厂商有 F5、Citrix 等

软件负载均衡器

- 软件负载均衡器运行在通用服务器上，灵活性高，易于扩展和管理。常见的开源软件有 HAProxy、Nginx、Apache Traffic Server 等

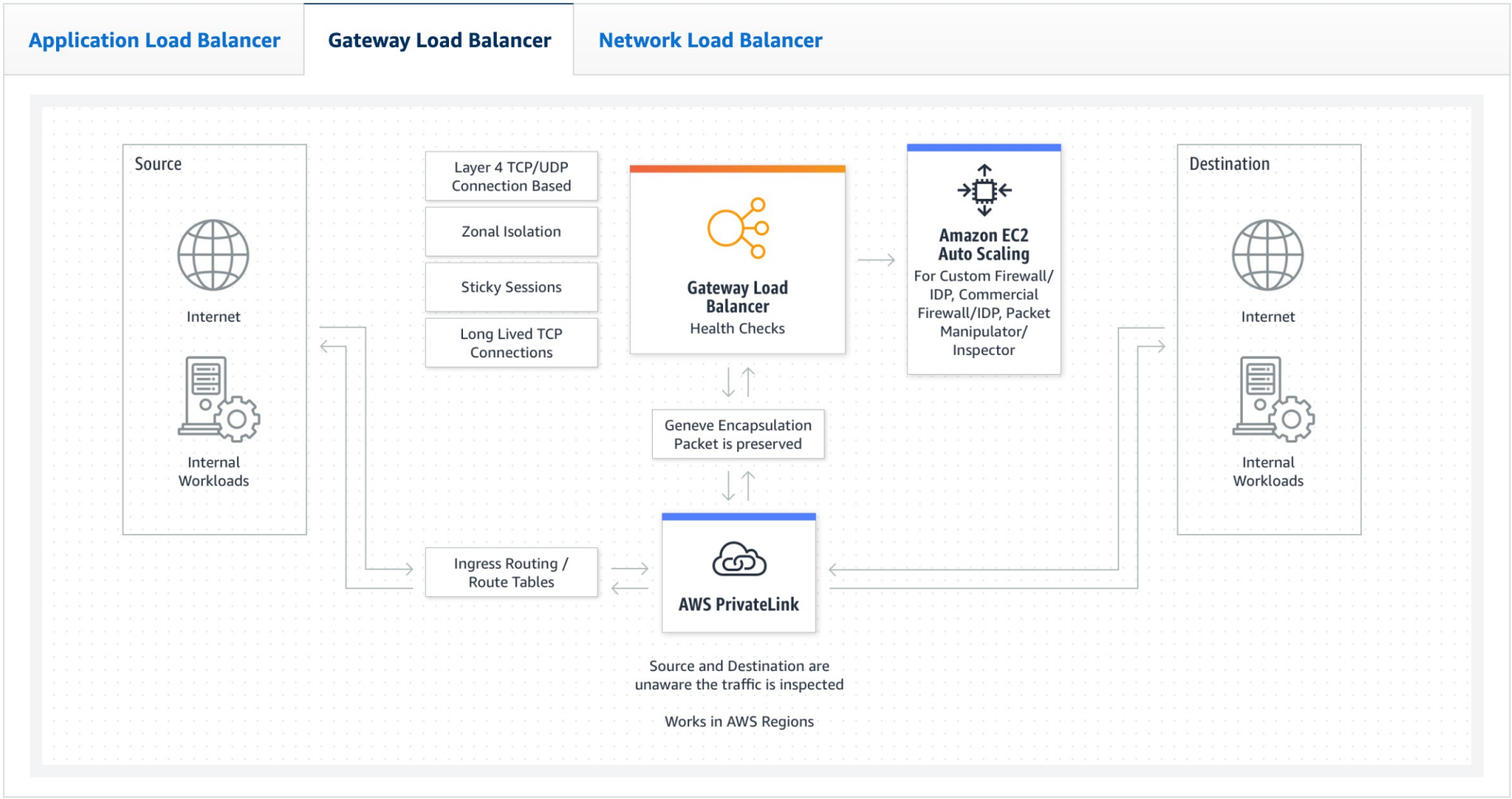


AWS 弹性负载均衡



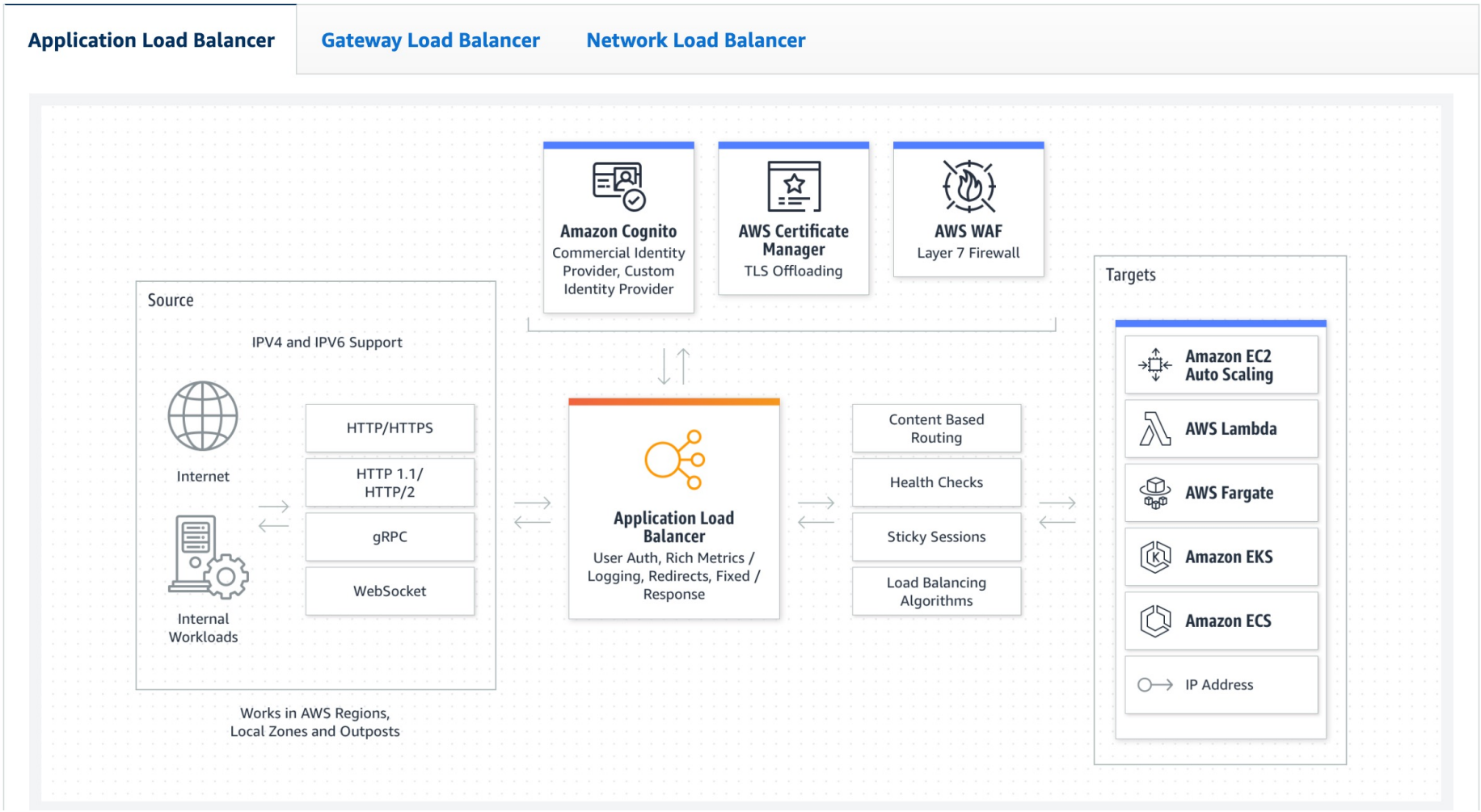
https://aws.amazon.com/elasticloadbalancing/

AWS 弹性负载均衡



https://aws.amazon.com/elasticloadbalancing/

AWS 弹性负载均衡



https://aws.amazon.com/elasticloadbalancing/

中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn