Application Performance Management in Virtualized Datacenters

Anne Holler, Velocloud Networks
Xiaoyun Zhu, Futurewei Technologies
Rean Griffith, Illumio

Introduction

- In a **physical datacenter** app execution environment, we see:
  - **Dedicated hardware**, over-provisioned for peak app perf scaling
  - **Manual resource mgmt** for availability & capacity planning
- In a **virtualized datacenter** app execution environment, we see:
  - **Multiplexed scheduling** of shared hardware resources
  - **Automatic resource mgmt** for efficient app perf scaling
  - **More complicated application performance management**
    - Datacenter resource schedulers, automatic application scaling, workload telemetry data are important elements

Tutorial Outline

- **Survey of datacenter resource schedulers**
  - **Their role in application performance mgmt**
- Achieving Service Level Objectives (SLOs) via automatic application scaling
- Analytics pipelines for workload telemetry data

Datacenter Resource Schedulers

- Many in active use
    - E.g.: Mesos, Kubernetes, Borg, VMware DRS, Openstack Nova, Microsoft SCVMM, Hadoop, etc.
- What do they all do?
- Why are there so many?
- How are they used for app performance mgmt?

What do they all do?

- **Environment**: Datacenter resource schedulers **run in frameworks with**
  - **Encapsulation support**: Virtual machines (VMs); Containers
  - **Infrastructure mgmt**: Inventory; Permissions; Deployment; Migration
  - **Monitoring**: Metrics; Alerts; Troubleshooting; Capacity planning

- **Activity**: Datacenter resource schedulers **map workloads to resources**
  - **Placement:** Select target capacity for workloads
  - **Deployment:** Orchestrate launching of workloads
  - **Remediation:** Address workloads’ runtime issues

Why are there so many?

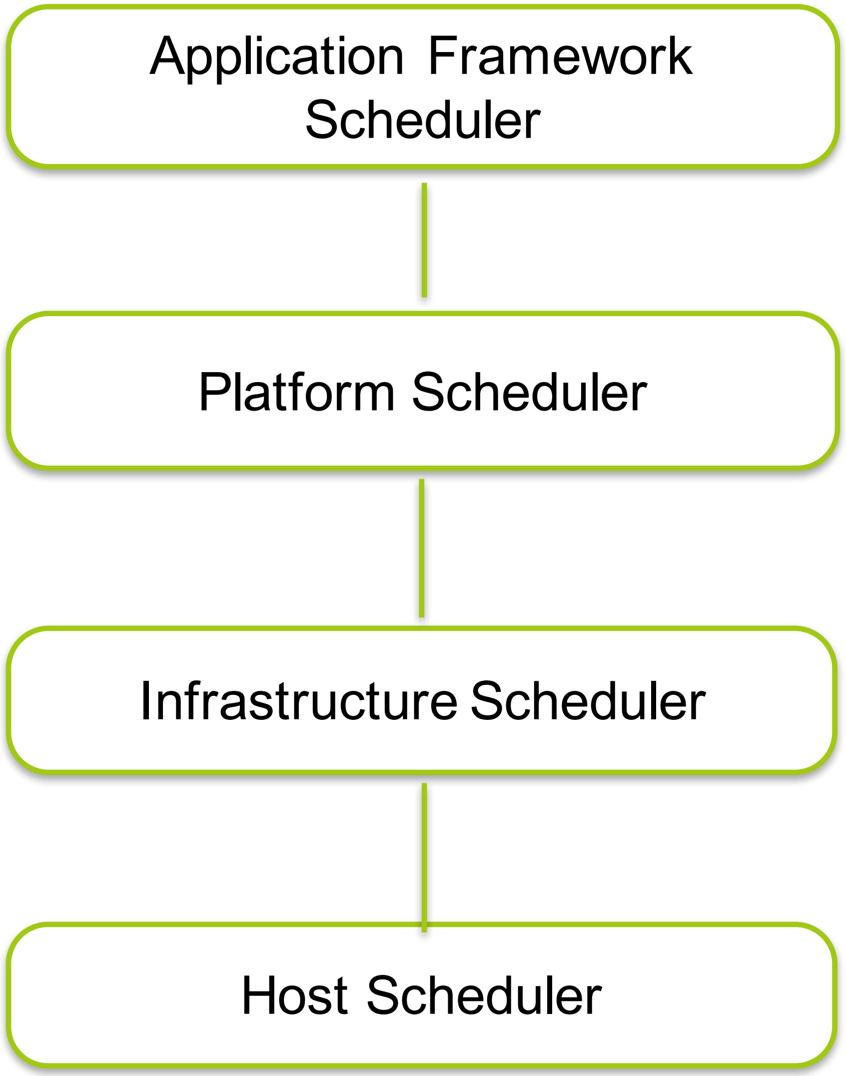
- **Placement criteria vary**: Datacenter resource schedulers may consider
  - **Constraints**
    - Hard: e.g., hardware, availability, licenses, storage access
    - Soft: e.g., network locality, cost
  - **Resources**
    - Available CPU, memory, I/O bandwidth, storage space, power
  - **Policies**
    - Performance guarantees
    - Efficiency: static versus dynamic partitioning
    - Fairness: resolution of contention

Why so many? continued

- **Target apps vary**: Datacenter resource schedulers may target
  - **Scale-up applications**
    - Examples: Exchange, Oracle, SQL server
  - **Scale-out applications**
    - Example: Web service stacks like Apache/JBoss/MySQL
  - **Scale-out applications w/specialized job management**
    - Examples: Hadoop, Spark, Jenkins

- **Operating levels vary**: Datacenter resource schedulers may operate
  - At different time scales, on different entities, at different infrastructure granularity

Datacenter Resource Schedulers Levels



Manage application jobs; map application jobs to platform tasks

Application Framework Scheduler

Broker the infrastructure scheduling of tasks, orchestrate deployment, provide app features such as service discovery & replication

Platform Scheduler

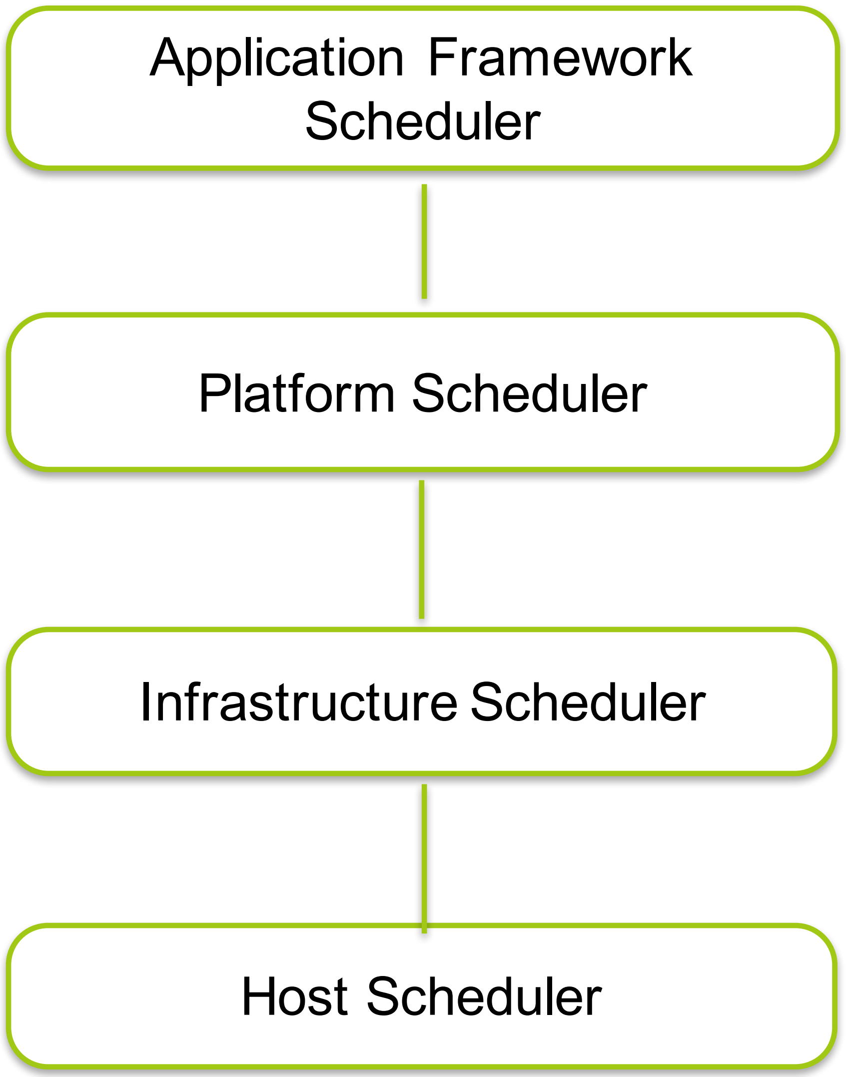
Place tasks on infrastructure & perform on-going cross-host coarse-grained scheduling

Infrastructure Scheduler

Perform on-going host-level fine-grained task scheduling

Host Scheduler

Datacenter Resource Schedulers Examples



Hadoop, Spark, Jenkins	Application Framework Scheduler
Mesos, Kubernetes	Platform Scheduler
DRS, SCVMM, Nova	Infrastructure Scheduler
ESX, Xen, KVM, Hyper-V, Linux+Docker	Host Scheduler

How are scheduler levels used for app perf mgmt?

- Hierarchy allows separation of concerns suited to the application & operating environment
  - Virtualized datacenter may not include all levels
- Higher levels depend on the capabilities of lower levels to efficiently achieve app perf SLOs
  - We examine typical level capabilities from lowest up
  - We consider examples at each level & pros/cons of including that level in app perf mgmt stack

Virtualized Datacenter Host Resource Schedulers

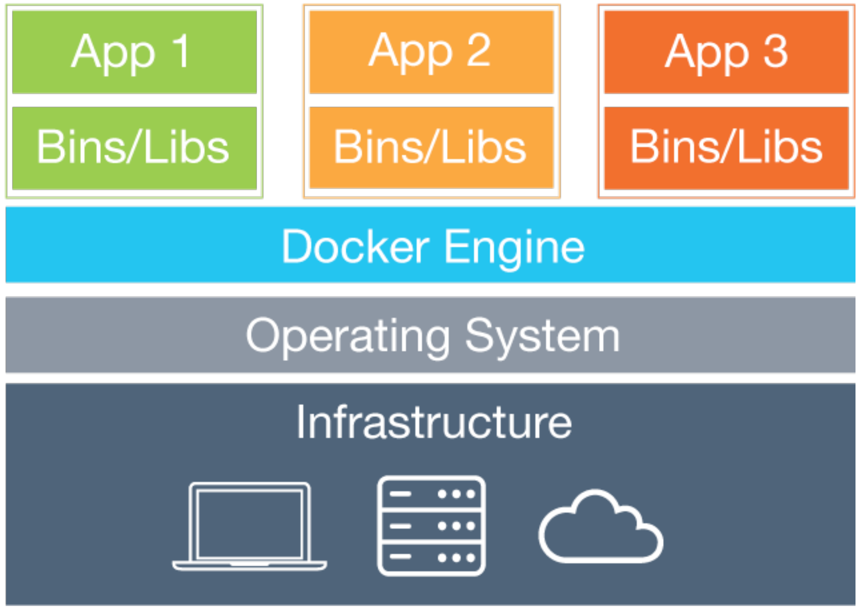
- Key attributes
  - Resources managed
    - Controls provided
  - Encapsulation supported
    - Isolation characteristics
    - Runtime overhead
    - Deployment model at scale

Host Resource Scheduler Example: VMware ESX hypervisor

- Resources managed, controls provided
  - CPU & Memory: reservations, limits, shares
    - CPU work conserving, Memory partially WC
  - Power: performance/power related policies
  - Network & Storage bandwidth: reservations, limits
- Encapsulation supported: VMs
  - Isolation: strong; reservation, limit controls; no guest OS sharing
  - Runtime overhead: fair; sufficient for production app usage
  - Deployment model at scale: clone from VM template
- Other hypervisors: Xen (AWS), KVM (GCE), Hyper-V (Azure)

Host Resource Scheduler Example: Linux+Docker

- Resources managed, controls provided
    - CPU: limits, shares [work-conserving]
    - Memory: limits [not work-conserving]
    - Network & Storage bandwidth: limits



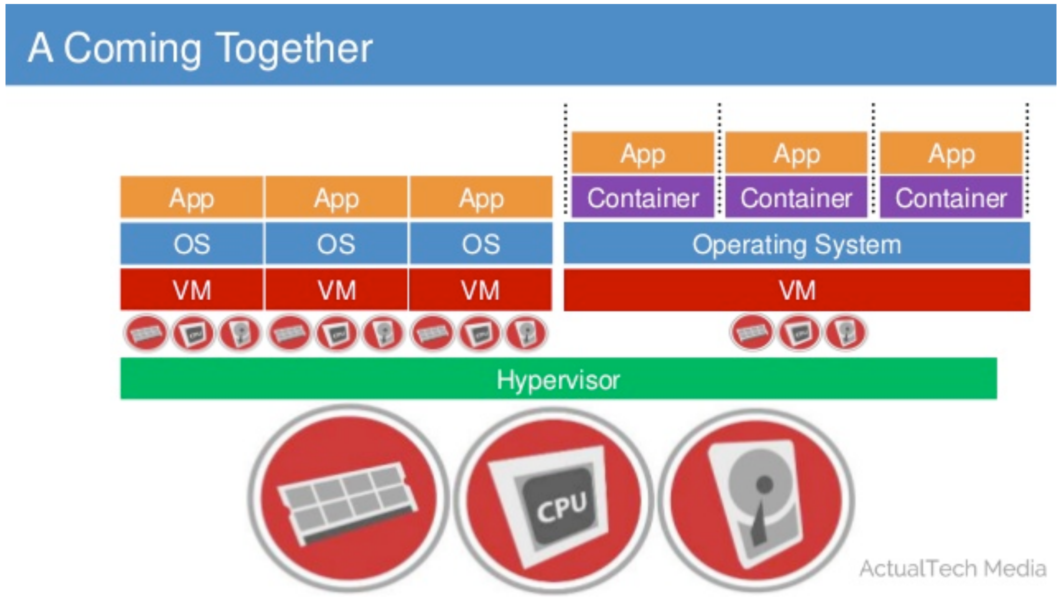
- Encapsulation supported: containers
    - Isolation: fair; partitioning via cgroups, namespaces, UFS
    - Runtime overhead: low; workloads share guest OS kernel
    - Deployment model at scale: run dockerfile to make image

Host Resource Scheduler
App Perf Mgmt Trade-offs

- More resource controls for VMs than containers, higher consolidation
- VMs provide better isolation than containers
- Legacy applications run as-is

- Containers avoid overhead of separate OS instance per encapsulation
- Containers have a simpler and faster deployment model than VMs
- Applications need to be developed or adapted for containerization

- Hybrid (containers in VMs) can be used to get benefits of both



Virtualized Datacenter Infrastructure Schedulers

- Key capabilities
  - Select host & datastore from cluster to run encapsulated tasks
    - Consider resource availability and a wide variety of constraints
  - Perform ongoing coarse-grained infrastructure scheduling
    - May support resource controls for perf, fairness, efficiency
    - May migrate encapsulated tasks to satisfy resource needs

Infrastructure Scheduler Example: VMware DRS & Storage DRS

- Resources managed, controls provided: same as ESX Hypervisor
    - CPU, Memory, Power, Network, Storage
- Hard & Soft Constraints respected
    - VM/host compatibility (cpu features, storage access, etc), availability, VM/VM,VM/Host,VMDK/VMDK affinity/anti-affinity
- Goal
    - Satisfy constraints and balance normalized entitlement for headroom benefit subject to migration cost

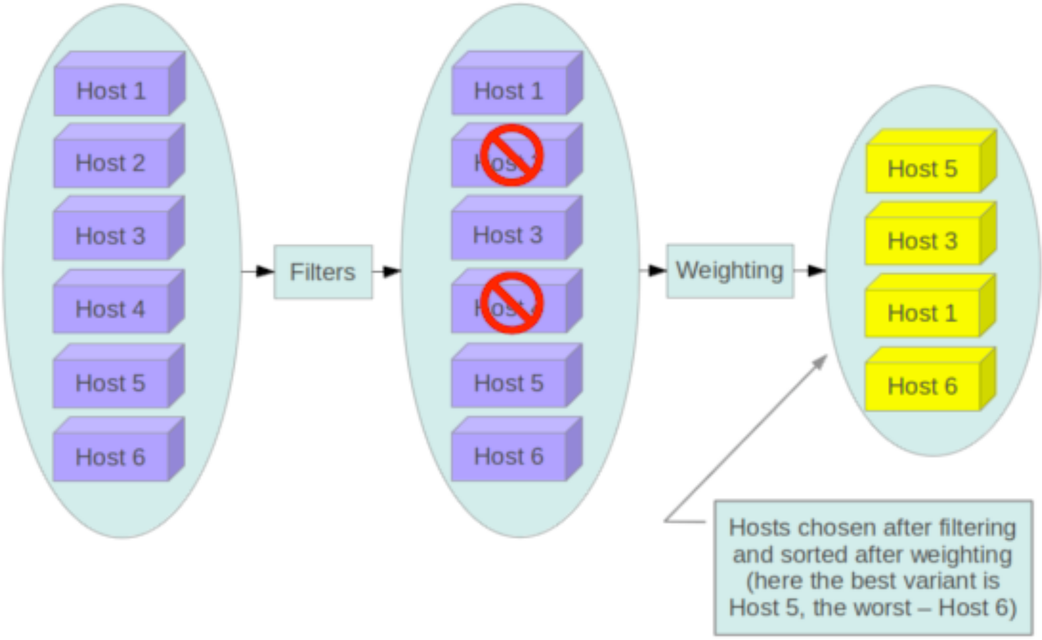
VMware DRS automatically and intelligently balances resources among virtual machines.

Infrastructure Scheduler Example: Openstack Nova Filter Scheduler

- Resources managed
  - Weights consider available RAM, free disk space, IOPS, running VMs count, and optionally utilization

- Hard & Soft Constraints respected
  - VM/host compatibility (cpu features, storage access, etc), availability, affinity/anti-affinity

- Goal: Satisfy constraints and choose host w/best weight score
  - No migrate for remediation; admin can request re-placement



Hosts chosen after filtering and sorted after weighting (here the best variant is Host 5, the worst – Host 6)

Infrastructure Scheduler App Perf Mgmt Pros/Cons

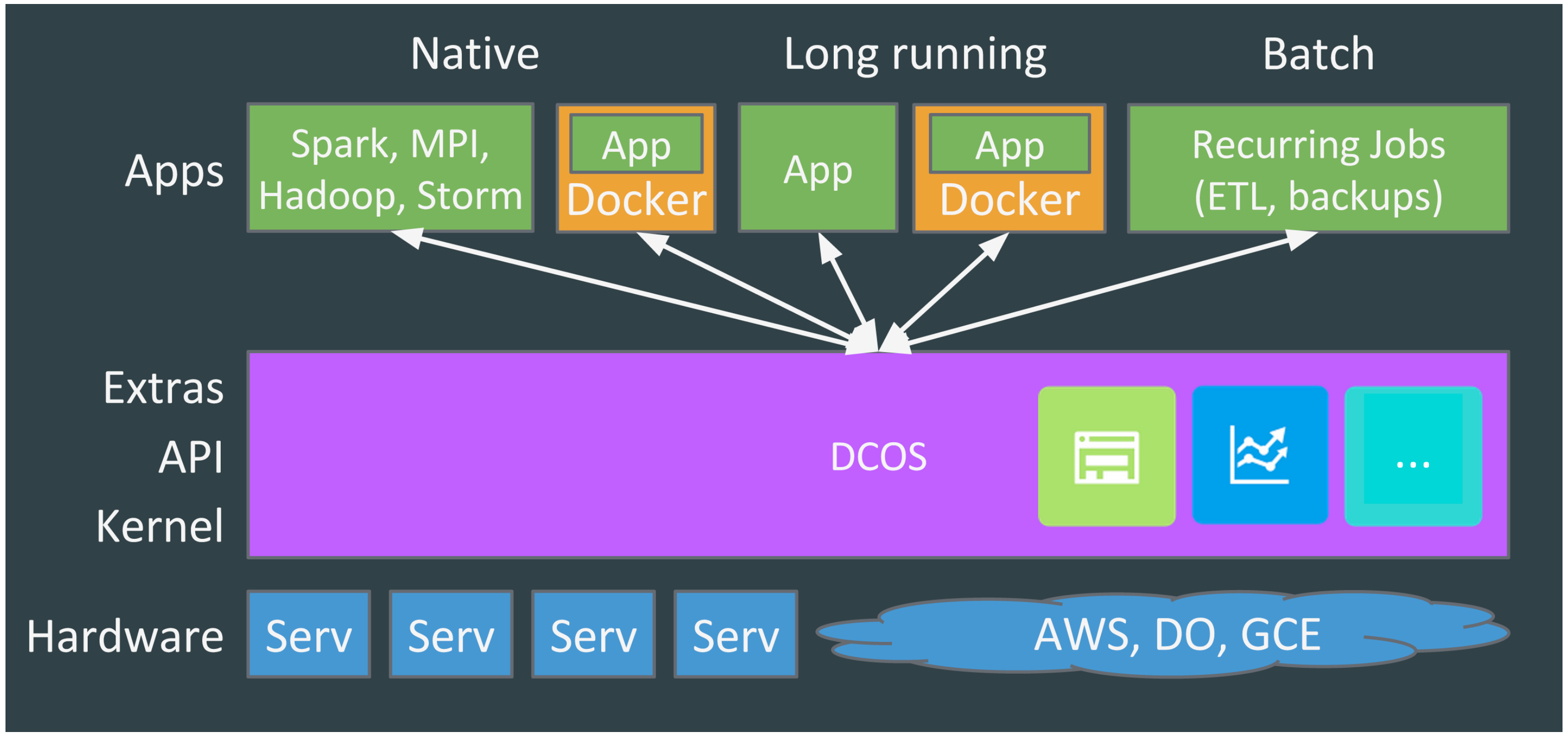
- Pros
  - Efficient sharing of heterogeneous infrastructure by heterogeneous apps
  - Can use live migrate for perf remediation or hw maintenance; avoids app perf impact, good for scale-up apps
- Cons
  - Supporting heterogeneity & high host efficiency limits cluster scalability
- Hybrid
  - Use higher level scheduler to choose cluster, use infrastructure scheduler to do placement & ongoing mgmt w/in cluster

Virtualized Datacenter Platform Schedulers

- Key capabilities
  - Match available resources to application frameworks’ tasks
  - Handle task deployment orchestration & ongoing mgmt
    - Encapsulate tasks & assign to available resources; Set up communication channel btw encapsulated tasks
    - Maintain desired number of healthy task instances; Provide service discovery
  - Track/arbitrate resource allocation across app frameworks

Platform Scheduler Example: Mesos DCOS

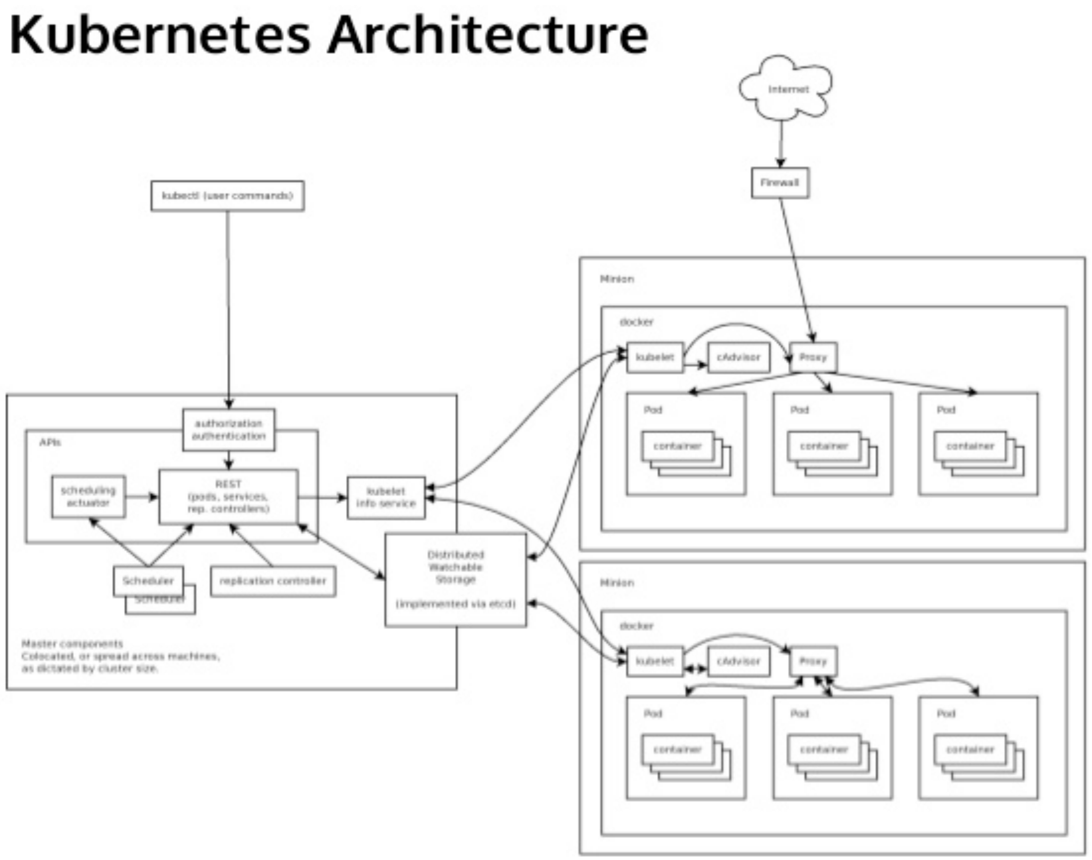
- Resources managed & controls: same as Linux+Docker containers
  - CPU, Memory, Network, Storage
- Handles common application orchestration
- Interoperates with & arbitrates btw many app frameworks



Platform Scheduler Example: Google Kubernetes

- Resources managed & controls: same as Linux+Docker containers
  - CPU, Memory, Network, Storage

- Schedules pods [colocated containers], provides replication count controller, supports labeling and service discovery



Platform Scheduler
App Perf Mgmt Pros/Cons

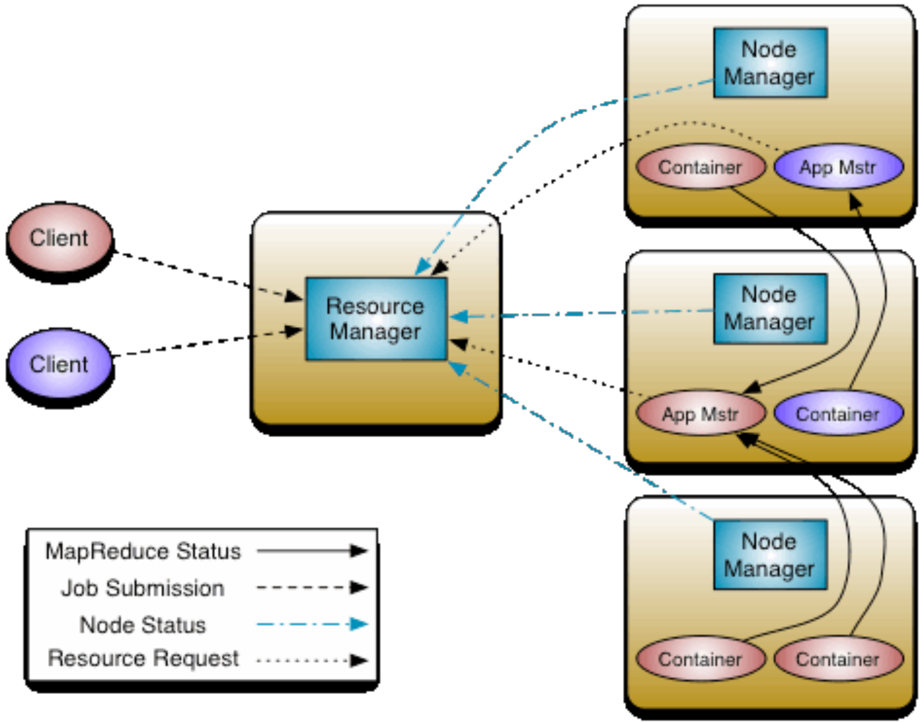
- Pros
  - High scale managing homogeneous tasks & infrastructure
  - At Twitter, Mesos manages 10s of thousands of hosts
- Cons
  - Scale can sacrifice heterogeneity support & high consolidation
- .Hybrid
  - Add co-scheduler [e.g. Netflix Fenzo] to handle heterogeneity & drive higher consolidation as per framework policy
  - Layer platform over infrastructure scheduler to offload detailed handling of heterogeneity & infrastructure runtime remediation

Virtualized Datacenter Application Framework Schedulers

- Key capabilities
  - Provide job queuing & prioritization
  - Break jobs into tasks based on app attributes, e.g., parallel/serial operations & data locality
  - Launch task w/appropriate sequencing; handle task failure
  - Report job status & results to user

App Framework Scheduler Example: Hadoop YARN

- Resource management characteristics
  - Admits jobs based on platform resource availability, queueing jobs pending admission
  - Supports policies that can give differentiated treatment btw users or btw batch & interactive jobs



Application Framework Scheduler App Perf Mgmt Pros/Cons

- Pros
  - Handles app-specific aspects of scheduling
- Cons
  - Operational model of infrastructure ownership can result in low utilization w/o sharing & unexpected behavior w/sharing
- Hybrid
  - Layer over platform scheduler for sharing across apps, expose impact of sharing to application framework [e.g., vHadoop]

Tutorial Outline

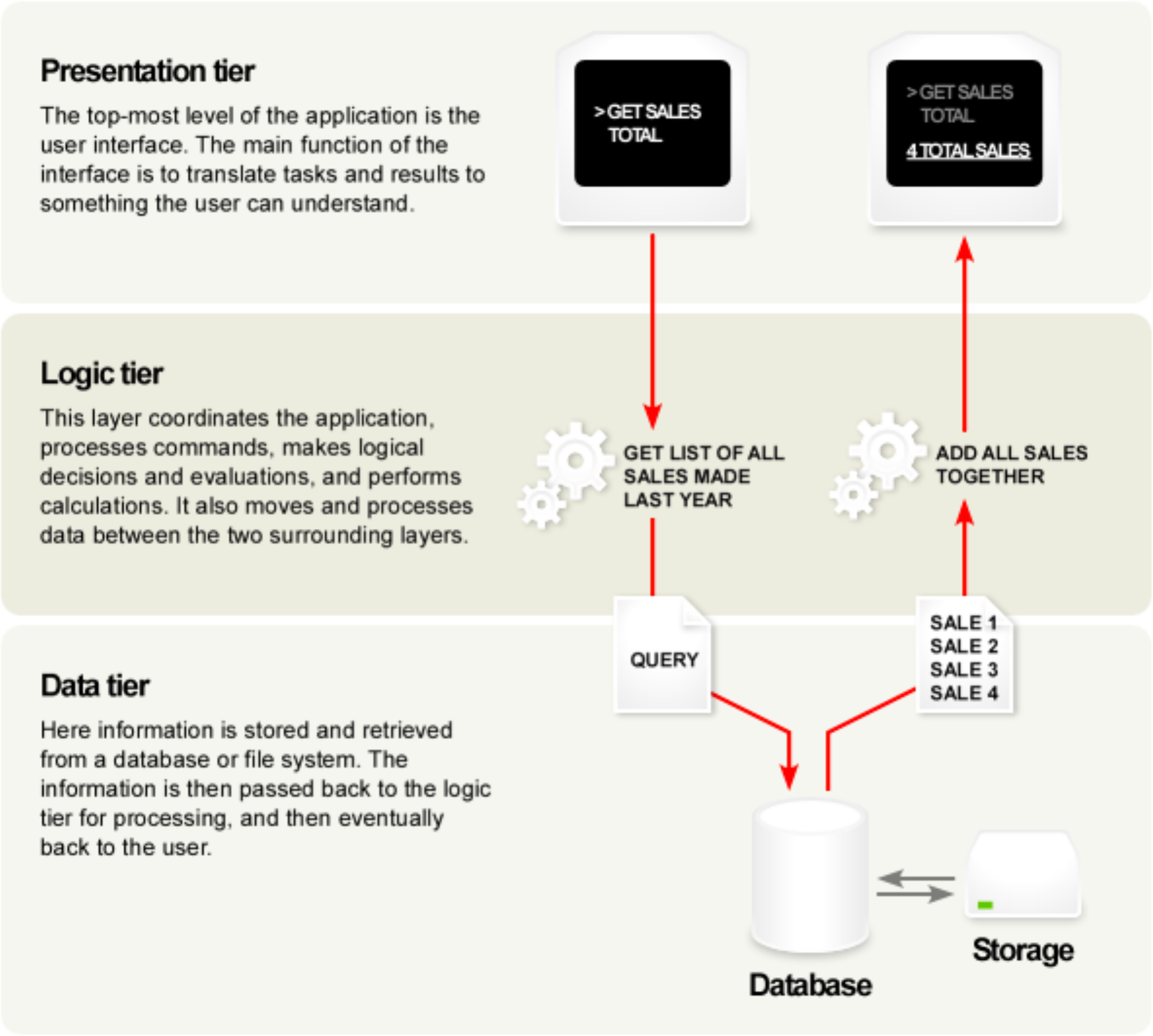
- Survey of datacenter resource schedulers
- **Achieving Service Level Objectives (SLOs) via automatic application scaling**
- Analytics pipelines for workload telemetry data

Achieving service level objectives (SLOs)

Why is it so hard – distributed application architecture

Wikipedia:

In software engineering, **multitier architecture** (often referred to as **n-tier architecture**) is a *client–server architecture* in which **presentation**, **application processing**, and **data management** functions are physically separated.



Presentation tier

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

Source: https://en.wikipedia.org/wiki/Multitier_architecture

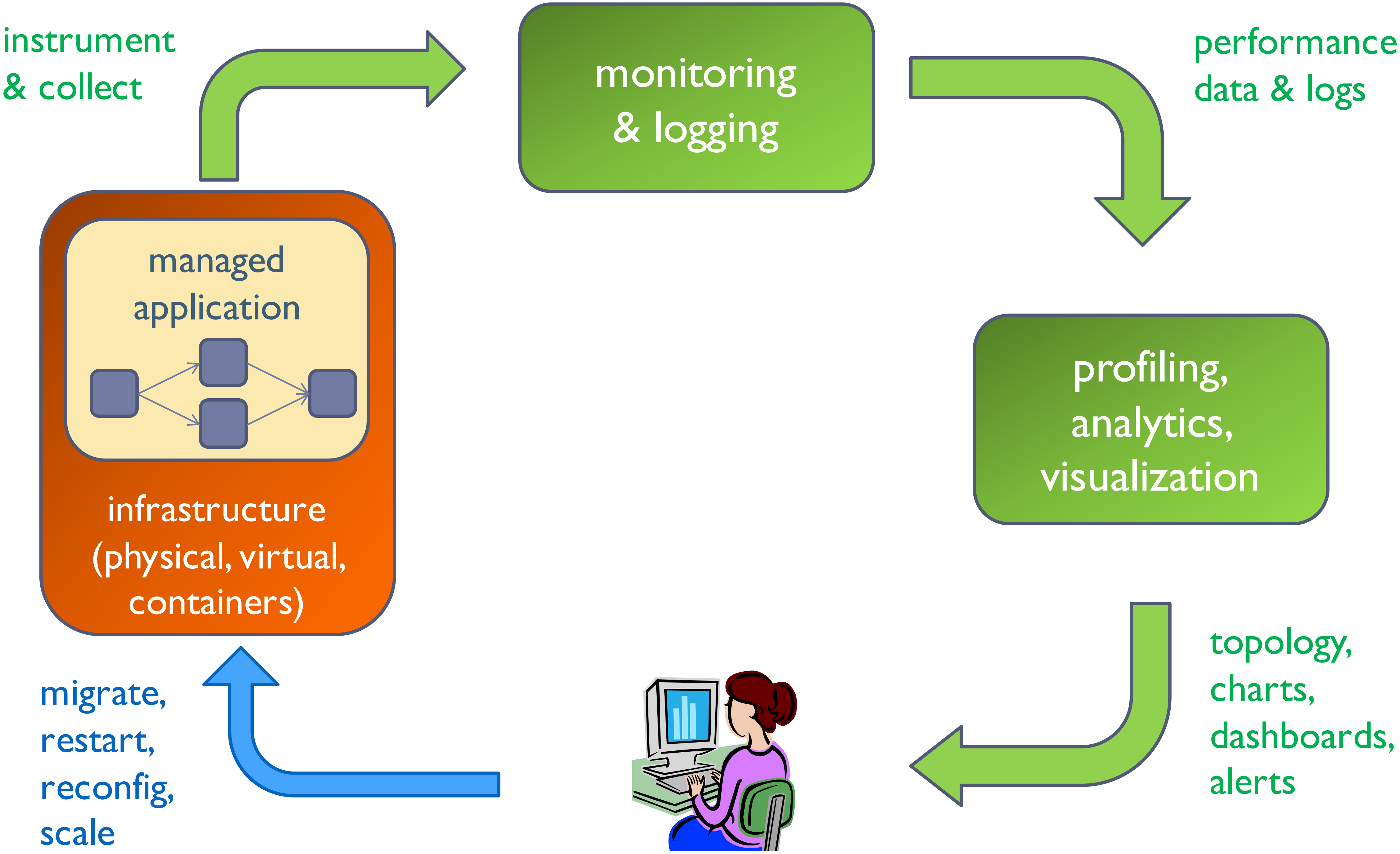
Achieving service level objectives (SLOs)

Why is it so hard – complexity of operating conditions

- End-to-end application performance depends on access to many heterogeneous resources
  - **HW**: CPU, memory, cache, network, storage, flash
  - **SW**: threads, connection pool, locks
- Dynamic and non-uniform hosting conditions
  - On-premises, cloud, hybrid
  - Physical, virtual, containers
- Time-varying application behavior
  - Frequent software updates
  - Seasonal or bursty workload demands
- Performance interference due to resource sharing
  - **Visible**: CPU/memory overcommitment
  - **Invisible**: processor cache, memory bandwidth

How is service level assurance done today?

An open loop system (human closes the loop)



instrument
& collect

monitoring
& logging

performance
data & logs

profiling,
analytics,
visualization

topology,
charts,
dashboards,
alerts

migrate,
restart,
reconfig,
scale

managed
application

infrastructure
(physical, virtual,
containers)

What kind of performance data are collected?

Infrastructure-level metrics

- System-level stats collected by the host/guest OS or hypervisor
    - CPU, memory, cache, disk, network, interrupt
- ~100s-1000s metrics per host; ~10s-100s metrics per VM/container
- Widely available from most OS/hypervisors/platforms
- Available at a time scale of milliseconds to seconds

Application-level metrics

- End-user experience (e.g., request response times)
- Workload characteristics (e.g., request mix/rate, throughput)
- Transaction tracing through application components
- Need agent deployment, or special instrumentation
- Often available at a time scale of seconds to minutes

What happens in a service level violation?

Log analysis

- Requires expert knowledge of the target application
- For modern, distributed, complex applications
  - Log files are distributed and need to be aggregated for analysis
  - Hard to know which log files to look at - finding a needle in a haystack
- Logs may not contain the necessary information
  - Performance concerns cause lower log levels (e.g., info) to be used in production
  - Often requires re-production of the problem with higher log levels
  - No information on infrastructure or third-party dependencies

What happens in a service level violation?

Performance charts and cook book

- Requires domain expertise and deep understanding of application behavior
- Best practice cook books cannot be used for problems not seen before
- Human-driven and reactive in nature
  - Rely on end user to report a problem first
  - Time consuming and error-prone
- Not scalable to large infrastructure with many (evolving) applications

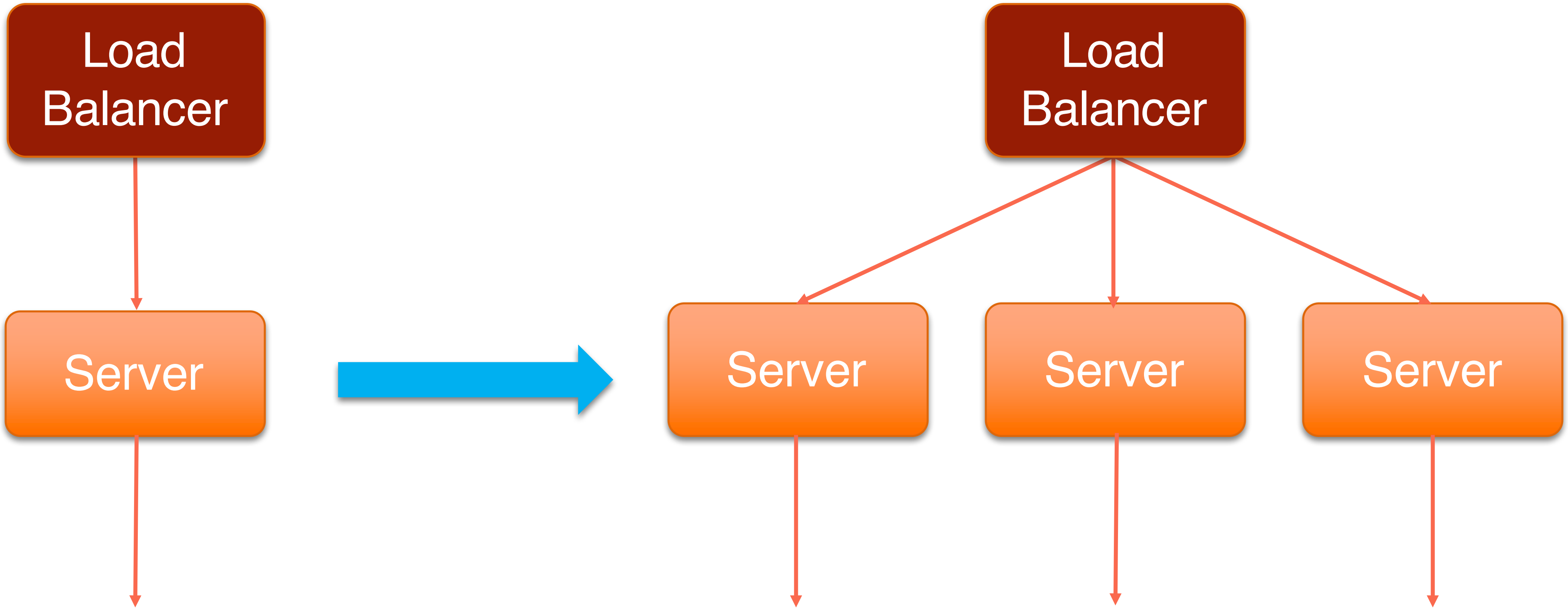
Application service level assurance

Necessary features

- **Proactive**: identify a problem before it impacts end users and business outcomes
- **Data-driven**: reduce dependency on human expertise and domain knowledge
- **Automation**: reduce time-to-resolution and increases scalability
  - Suitable for resolving service level violations due to resource bottlenecks or configuration errors that can be fixed programmatically (using APIs)
  - **In this talk: focus on automatic application scaling**
    - **Horizontal scaling**
    - **Vertical scaling**

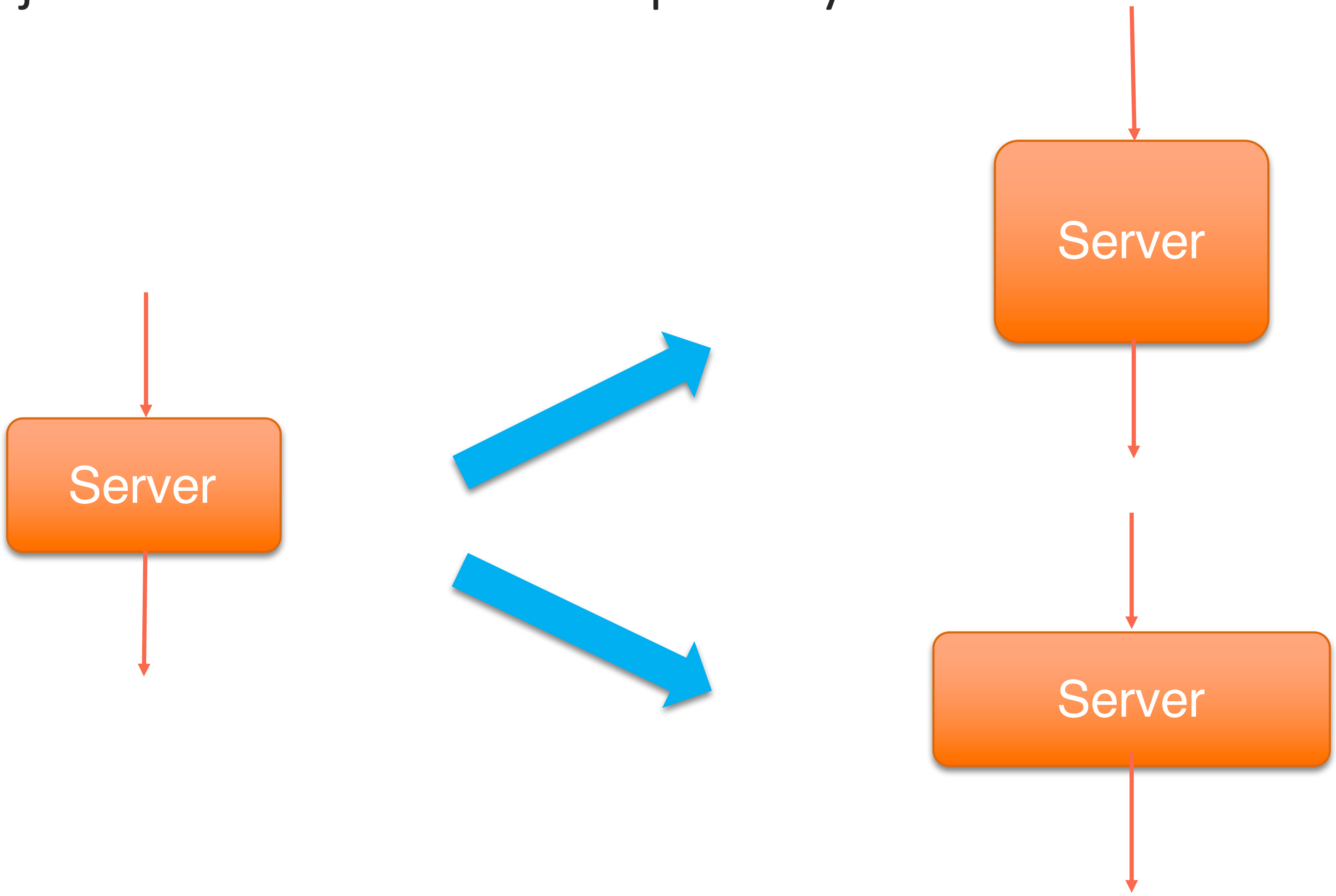
What’s Horizontal scaling?

- **Scale out/in:** Adding/removing instances in a specific tier
- Adjusting **concurrency level** in the application
- Usually requires a load balancer



What's vertical scaling?

- **Scale up/down:** Adding/removing capacity in a single instance
- Adjusting the **productivity level** of individual resources
- Can adjust different resources separately

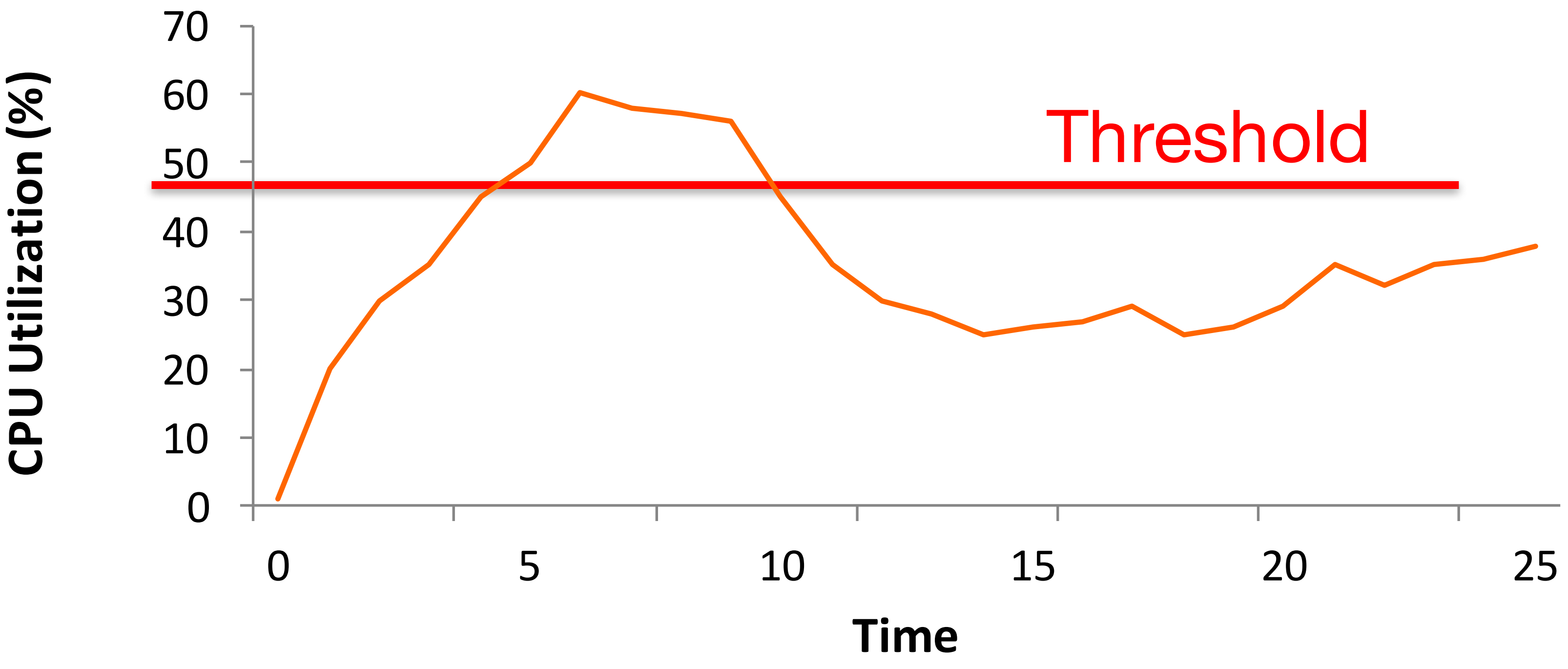


Horizontal vs. vertical scaling

Horizontal Scaling	Vertical Scaling
Requires scalable application architecture	No special architecture requirement
Fixed resource capacity profile	Flexible resource capacity profile
Slower execution	Faster execution
More suitable for stateless services	Suits both stateless and stateful services
No restart of application services	May require a restart of the application
No need for special platform support	Requires support from the platform
Not limited by physical host size	Limited by capacity of physical hosts

Horizontal scaling widely adopted in industry

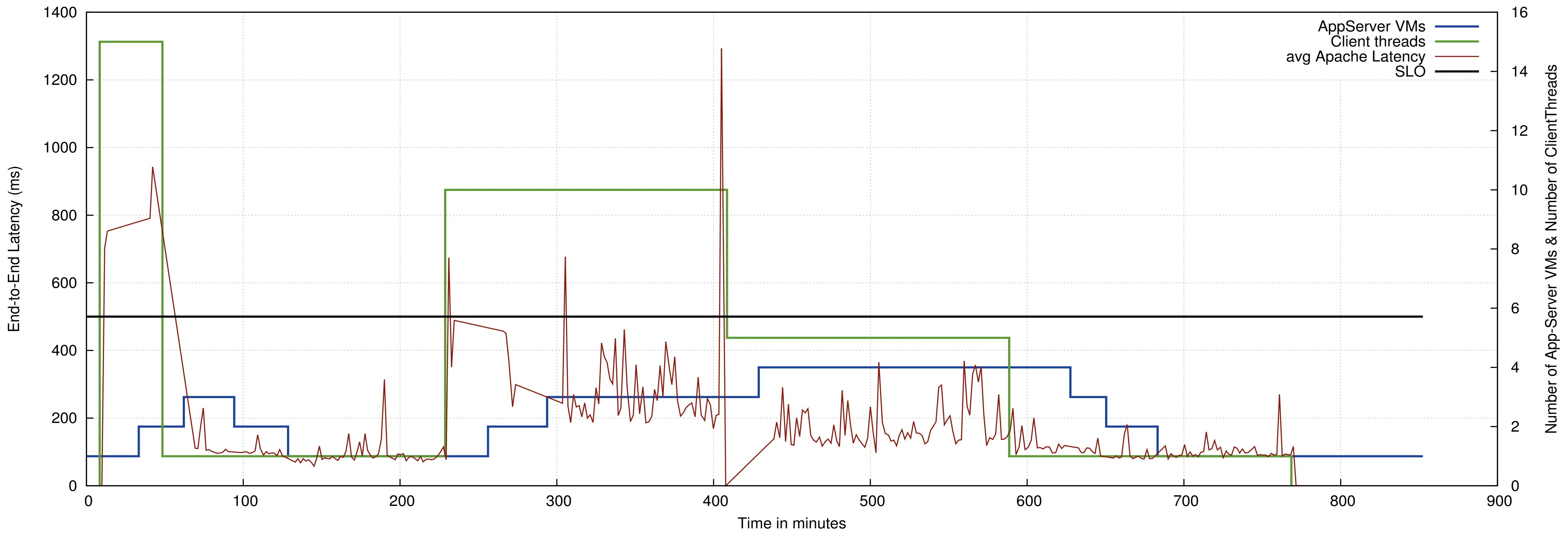
- Available from all major public cloud providers
  - Amazon AWS, Microsoft Azure, Google Cloud Platform, Rackspace
- Schedule-based or trigger-based
  - Spin up new instances when threshold is violated



- **Challenges**
  - How to handle different application services?
  - How to determine the right trigger and right threshold value?

Learning-based auto scaling

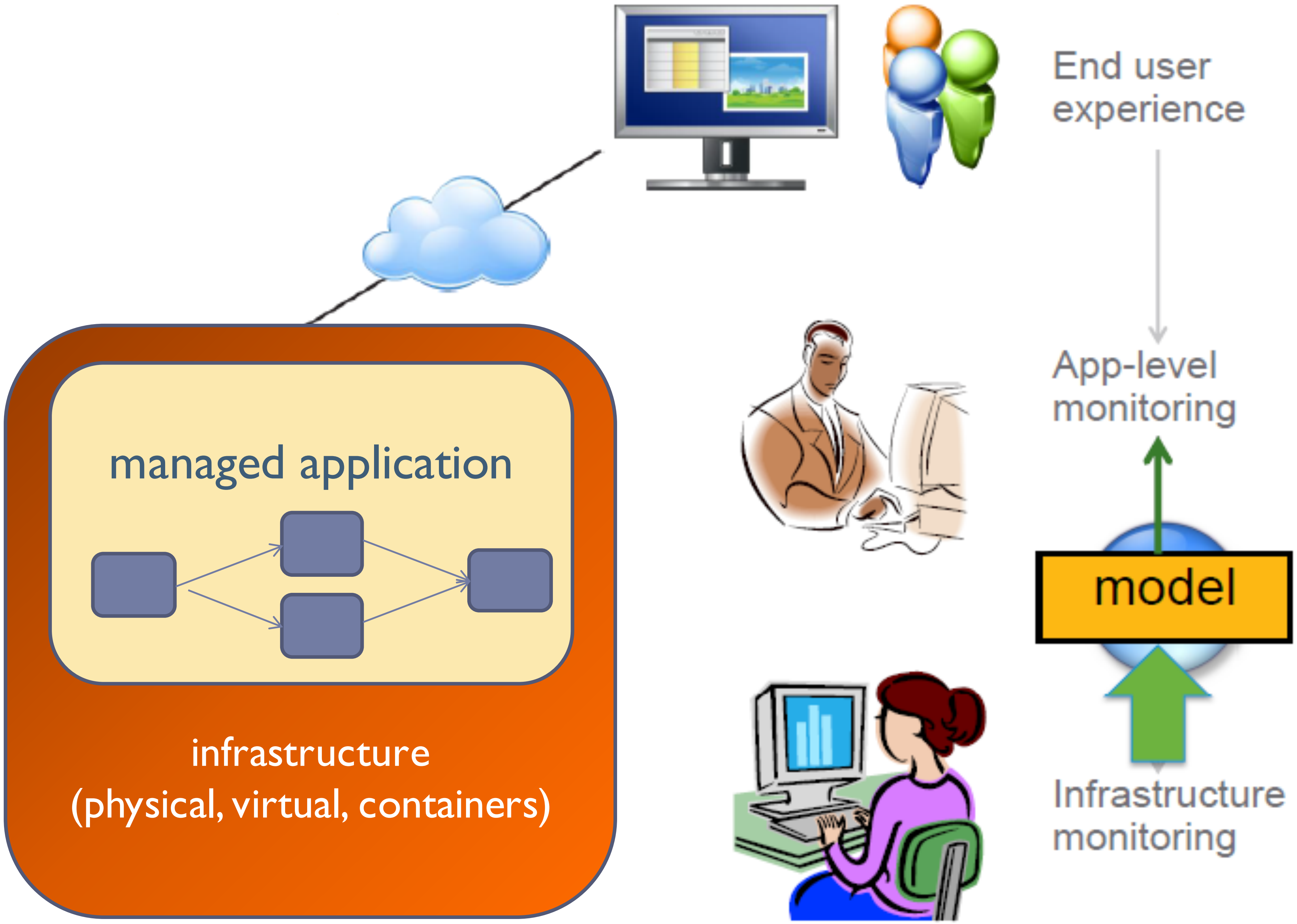
- Use **reinforcement learning** to capture application's scaling behavior and inform future actions
- Use **heuristics** to seed the learning process
- User only needs to provide **end-to-end latency** goal
- Handles **multiple** tiers automatically



** P. Padala et al. "Scaling of cloud applications using machine learning." VMware Technical Journal, Summer 2014.*

Challenges in vertical scaling

The semantic gap



managed application

infrastructure
(physical, virtual, containers)

End user
experience

App-level
monitoring

model

Infrastructure
monitoring

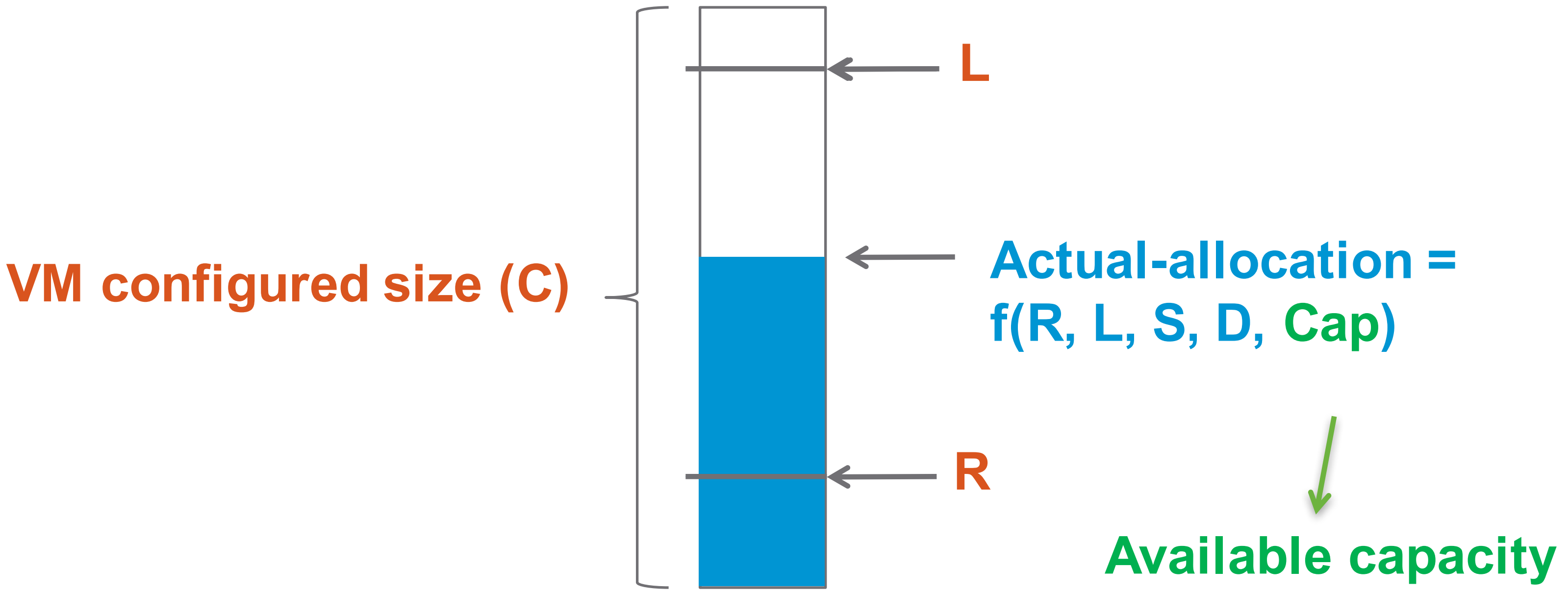
How to translate app-level performance goals to resource-level requirements?

Use models to capture app-resource mapping

Which metrics go into the model?

Depends on the **resource-level control knobs** available from the platform

- On VMware ESX, for shared CPU, memory, disk I/O*, network I/O*, :
  - ***Reservation (R)**** – minimum guaranteed amount of resources
  - ***Limit (L)*** – upper bound on resource consumption (non-work-conserving)
  - ***Shares (S)*** – relative priority during resource contention
- For CPU/memory: ***Configured size (C)*** – controllable by the user
- For CPU/memory: ***Demand (D)*** – estimated by the hypervisor (not directly controllable)



** A. Gulati et al. "VMware distributed resource management: Design, implementation, and lessons learned." VMware Technical Journal, April 2012..*

Use models to capture app-resource mapping

What kind of model should we use?

- White-box vs. black-box empirical models
- Linear vs. nonlinear models
- Offline vs. online models

White-box performance models

Pros

- Solid theoretical foundation
- Application-aware, easier to interpret
- Closed-form solution in some special cases

Cons

- Detailed knowledge of system, application, workload, deployment
- More often used for aggregate behavior or offline analysis
- Harder to automate, scale, or adapt

Black-box empirical models

Pros

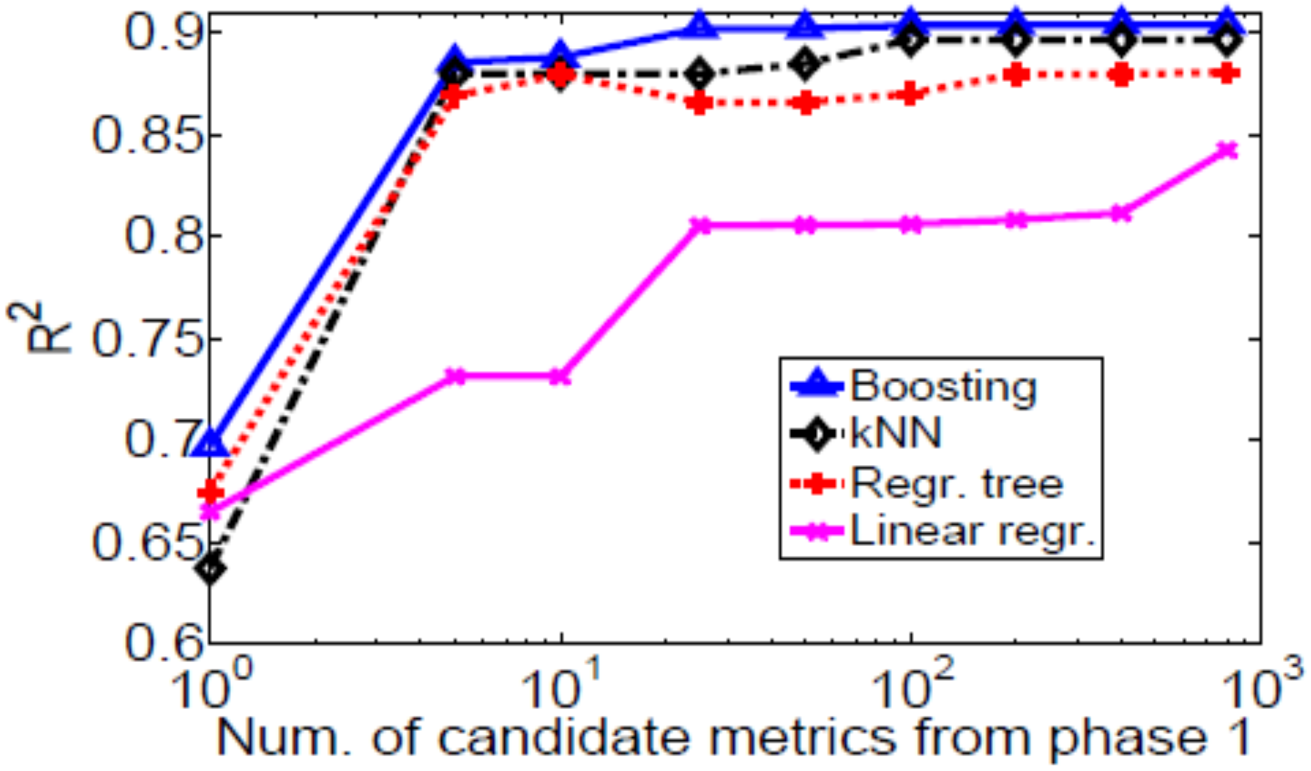
- **Generic**: No *a priori* assumptions
- **Tools**: Many learning algorithms available
- **Automation**: Easier to do partially or fully
- **Scalable**: Easier to codify analysis in algorithms

Challenges

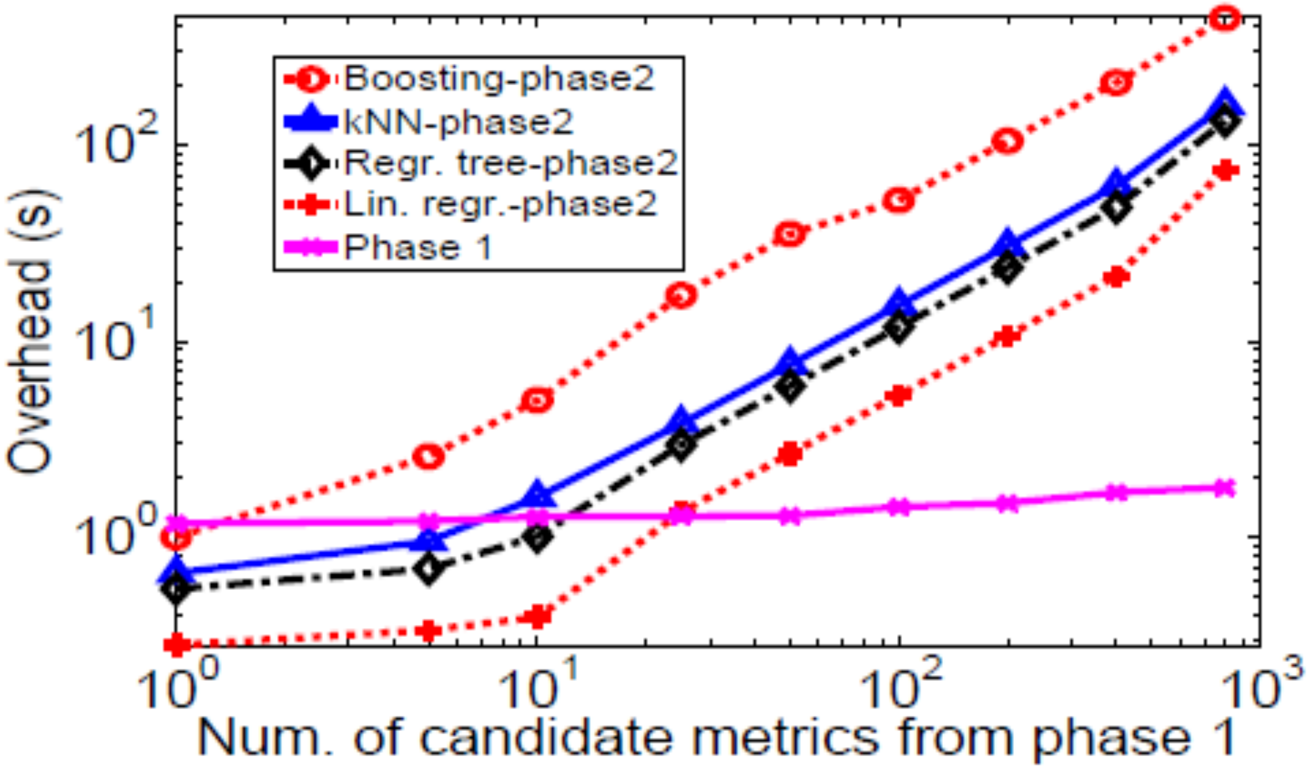
- **Efficiency**: Real-time data processing and analytics
- **Accuracy**: Reduces ***false positives*** and ***false-negatives***
- **Adaptivity**: Handles changing workloads and environments

Linear vs. nonlinear models

- **Nonlinear models** have better accuracy than linear regression model
- **Linear regression** model has the least computation cost
- **Boosting algorithm** has the best accuracy and highest cost



(e) R^2 of different models



(f) Two phase algorithm overhead

** P. Xiong et al. "vPerfGuard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments." ICPE 2013.*

Offline vs. online models

Offline modeling

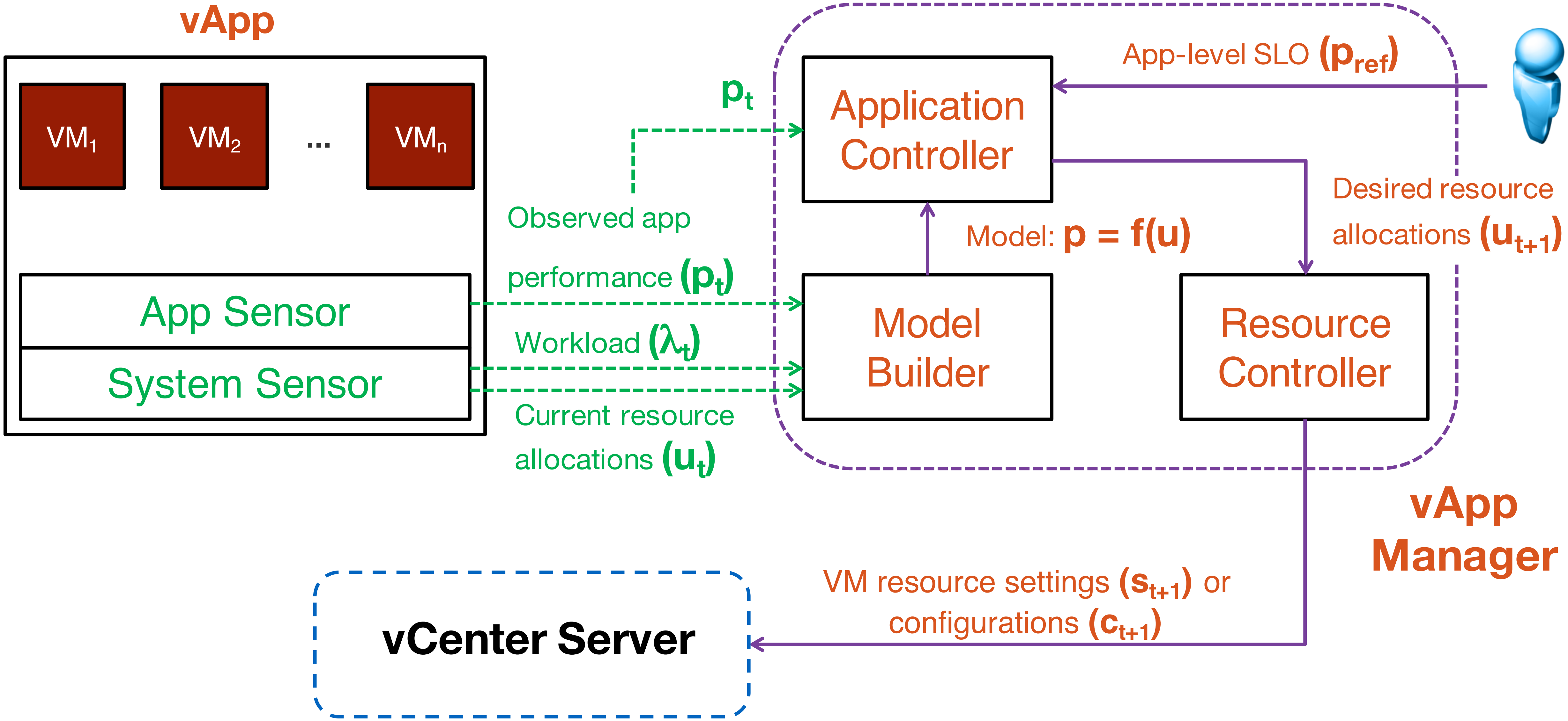
- More appropriate for nonlinear models
- More suitable for capacity planning and initial sizing
- Cannot adapt to runtime changes in app, workload, or system

Online modeling

- Should be cheap to compute and update
- Linear models more appropriate
- Can adapt to changes in application, workload, and system
- Suitable for runtime adaptation and reconfiguration

Automatic vertical scaling with control & optimization

For individual applications



Automatic vertical scaling – case studies

- **Case I:** VM CPU and memory scaling for MongoDB servers
- **Case 2:** CPU scaling for Zimbra Mail Transfer Agent (MTA)
- **Case 3:** Proactive memory scaling for Zimbra Mailbox Server

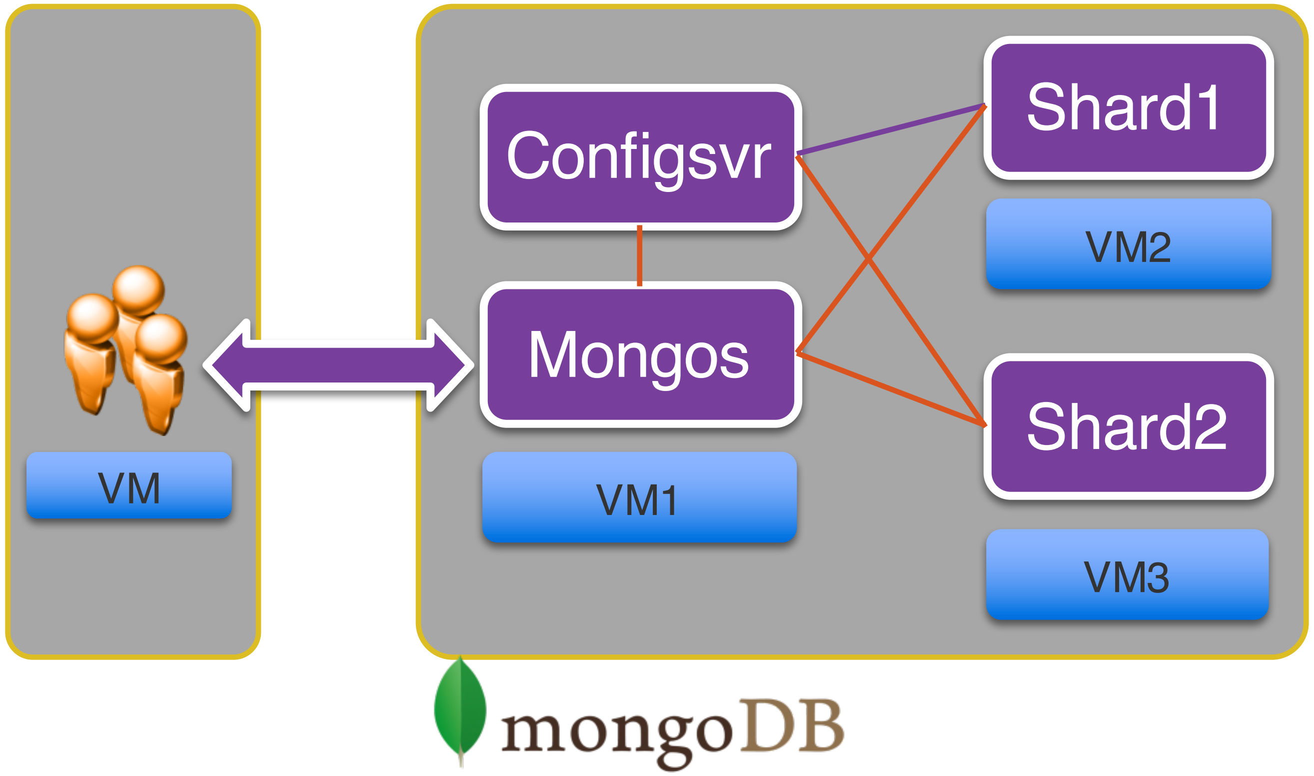
Case study I

CPU & memory scaling for MongoDB

- **Application**
  - **MongoDB** – distributed data processing application with sharding
  - **Rain** – workload generation tool to generate dynamic workload

- **Workload**
  - Number of clients
  - Read/write mix

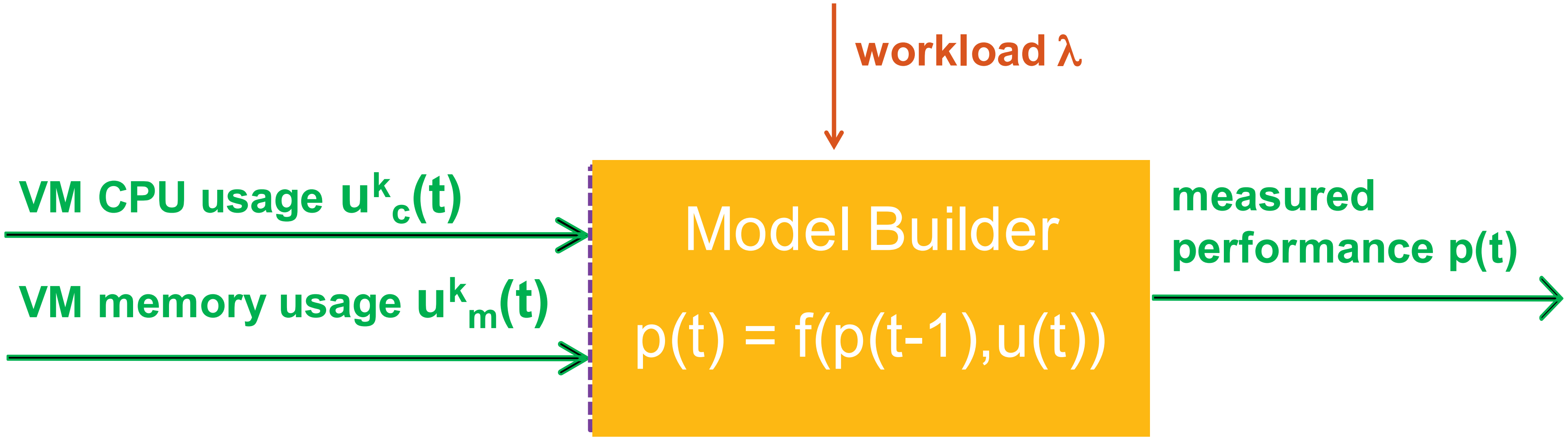
- **Evaluation questions**
  - Can the vApp Manager meet individual application SLO?



Performance model builder for a vApp

Maps VM-level resource allocations to app-level performance

- Captures multiple tiers and multiple resource types
- Choose a linear low-order model (easy to compute)
- Workload indirectly captured in model parameters
- Model parameters updated online in each interval (tracks nonlinearity)



workload λ

VM CPU usage $u^k_c(t)$

VM memory usage $u^k_m(t)$

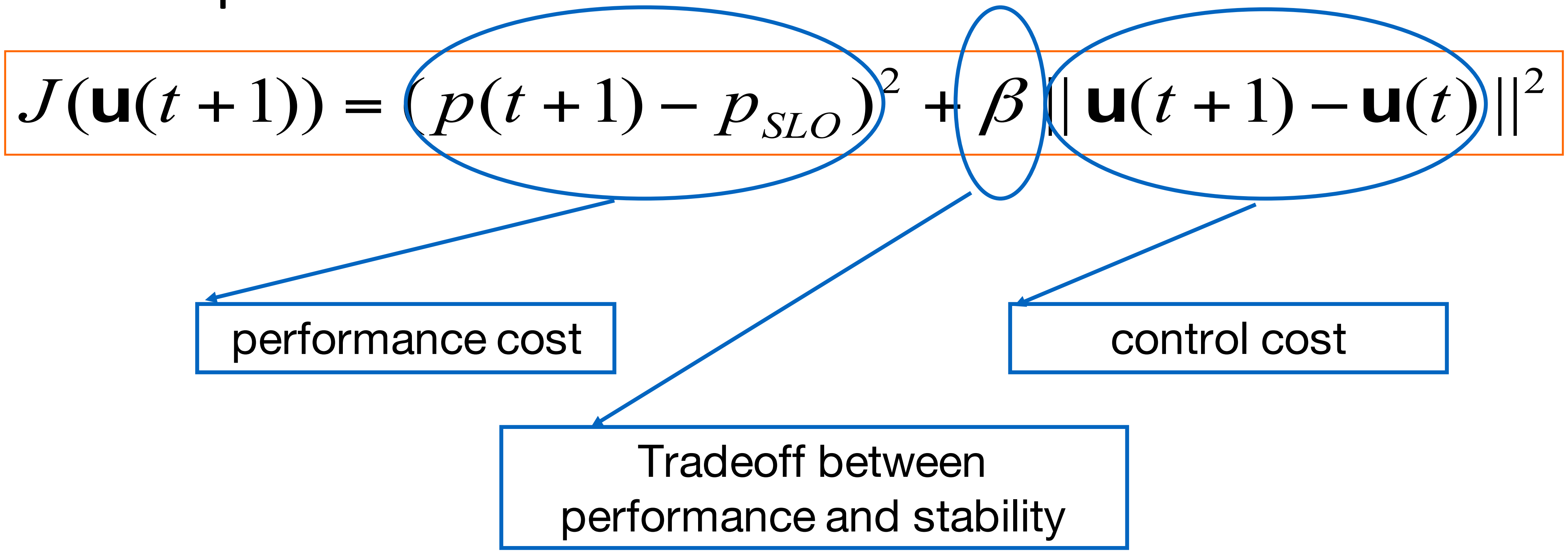
Model Builder

$p(t) = f(p(t-1), u(t))$

measured performance $p(t)$

Use optimization to handle design tradeoff

- An example cost function



$$J(\mathbf{u}(t+1)) = (p(t+1) - p_{SLO})^2 + \beta \,||\, \mathbf{u}(t+1) - \mathbf{u}(t) \,||^2$$

performance cost

control cost

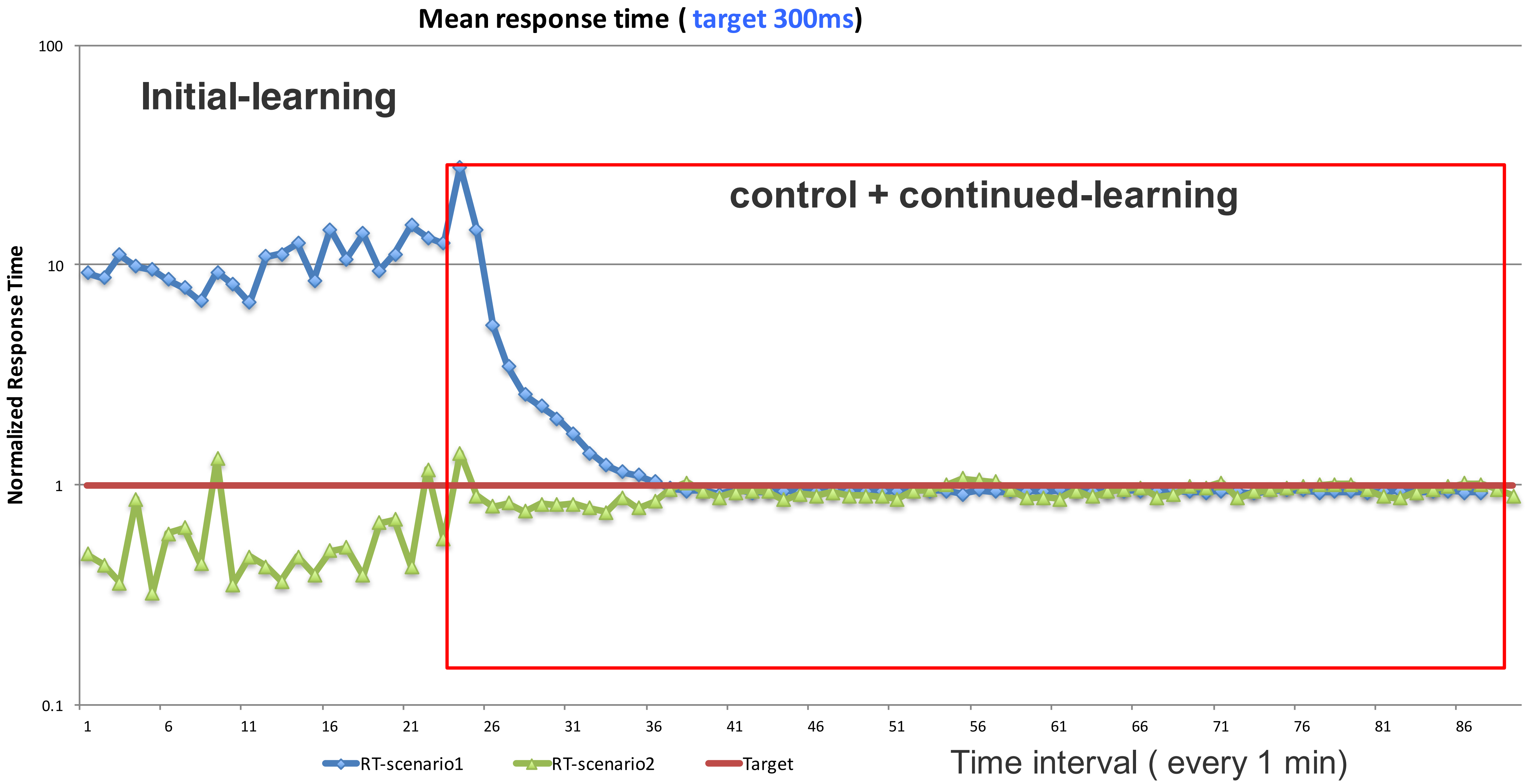
Tradeoff between performance and stability

- Solve for optimal resource allocations

$$u^*(t+1) = g(p(t), p_{SLO}, \mathbf{u}(t), \lambda, \beta)$$

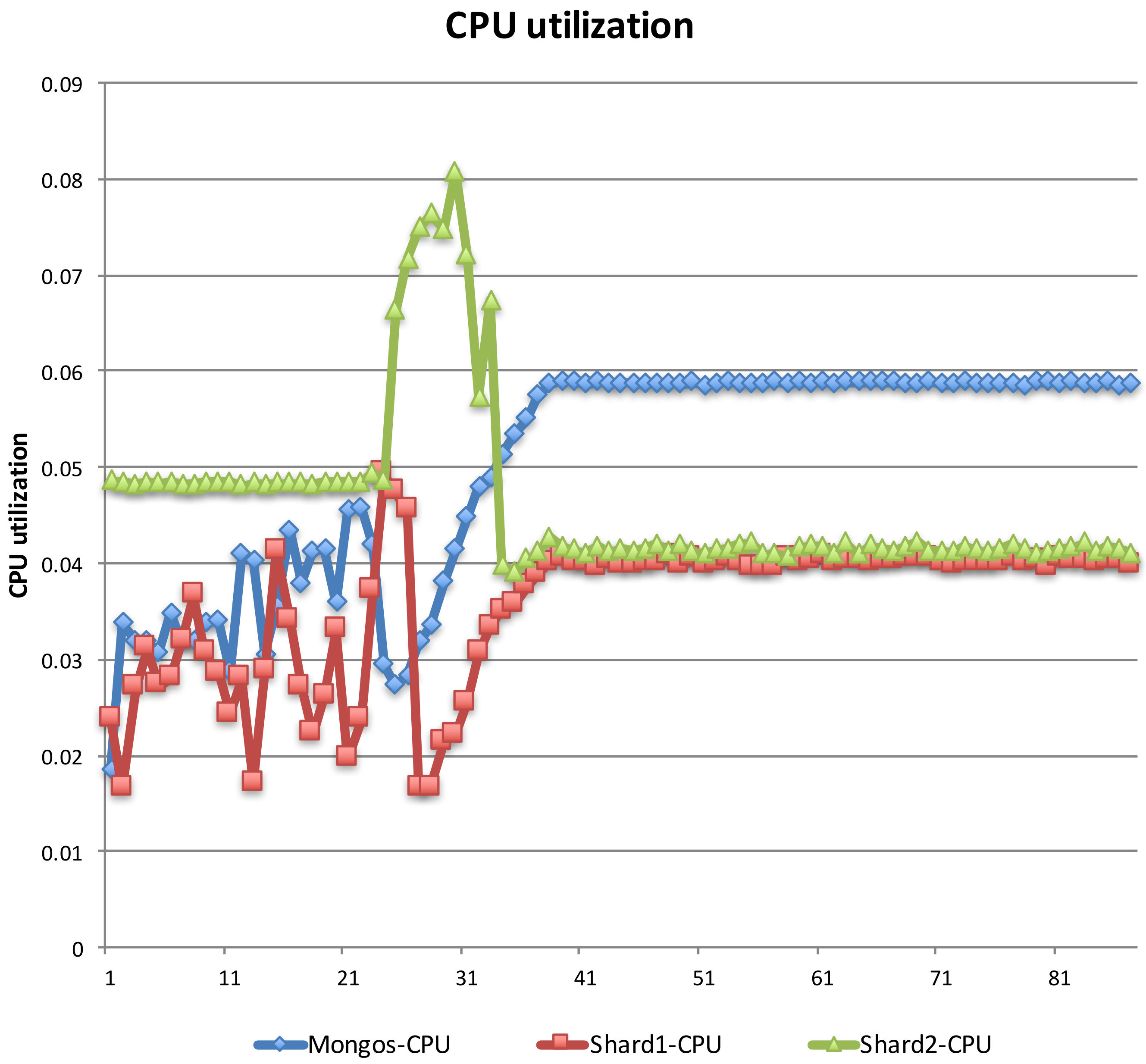
Result: Meeting mean response time target

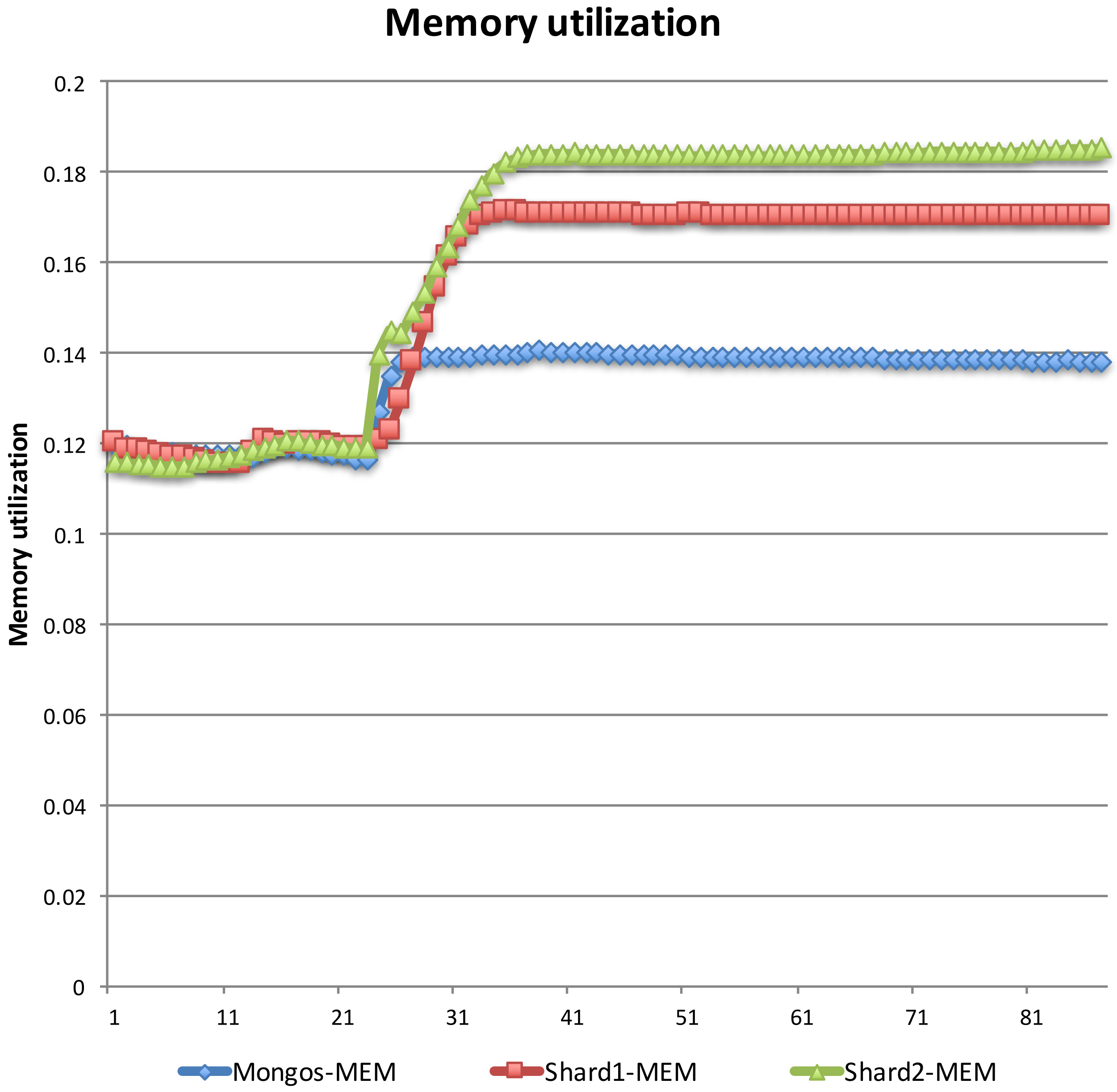
- Under-provisioned initial settings: R = 0, Limit = 512 (MHz, MB)
- Over-provisioned initial settings: R = 0, L = unlimited (cpu, mem)



Resource utilization (under-provisioned case)

- Target response time = 300 ms
- Initial setting R = 0, L = 512 MHz/MB (under-provisioned)



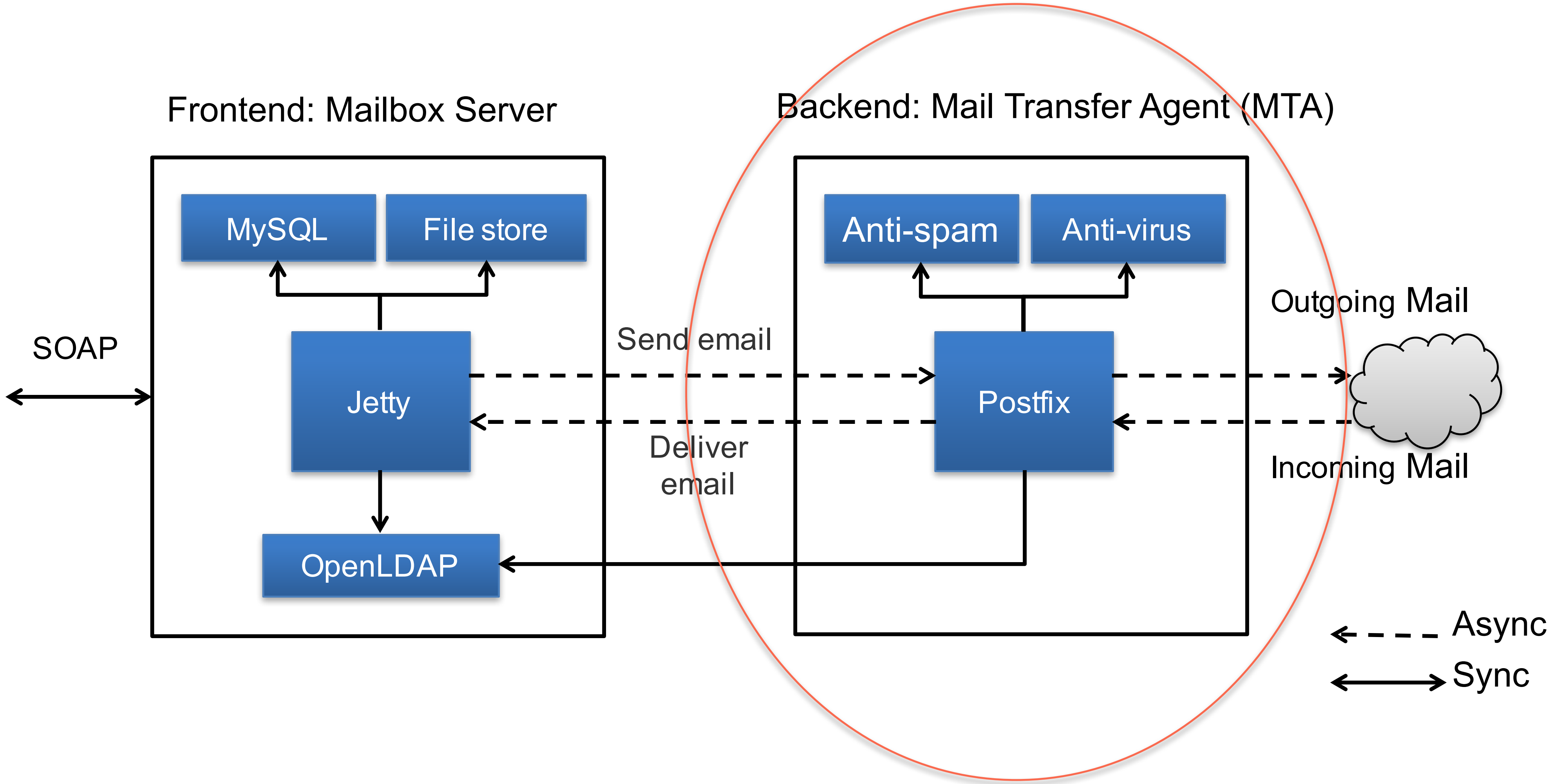


** L. Lu, et al., "Application-Driven dynamic vertical scaling of virtual machines in resource pools." NOMS 2014..*

Case study 2

CPU scaling for Zimbra Mail Transfer Agent (MTA)

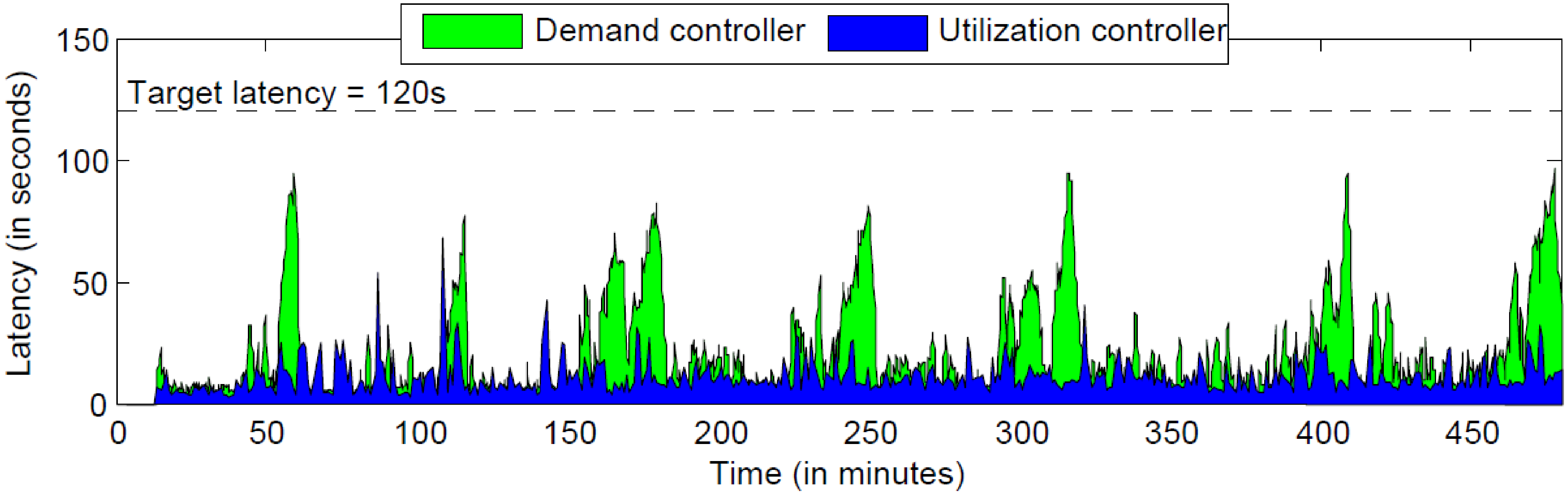
- **Application:** Open-source Zimbra collaboration software
- **Workload: ZimbraPerf** email workload generator
- **Scaling parameter:** CPU configuration (#vCPUs) for MTA VM



Scaling policies evaluated

- **Model-based controller**
  - Control interval 20 seconds
  - Estimation interval 5 minutes
- **Trigger-based controller**
  - Thresholds
    - Scale-up if utilization > 90%
    - Scale-down if utilization < 40%
  - Control interval: 1 or 5 minutes
- **Workload trace**
  - One week from FIFA 98 Worldcup access logs
  - Scaled to 8 hours experiment duration

Performance evaluation result



Scaling Policy	Mean latency [s]	Number of Reconfigurations	Mean vCPUs	Max vCPUs
Model-based	20.48	13	1.4	2
Trigger-based (1 min)	10.82	273	1.83	3
Trigger-based (5 min)	25.97	72	1.46	3
Static allocation	1385	0	1	1

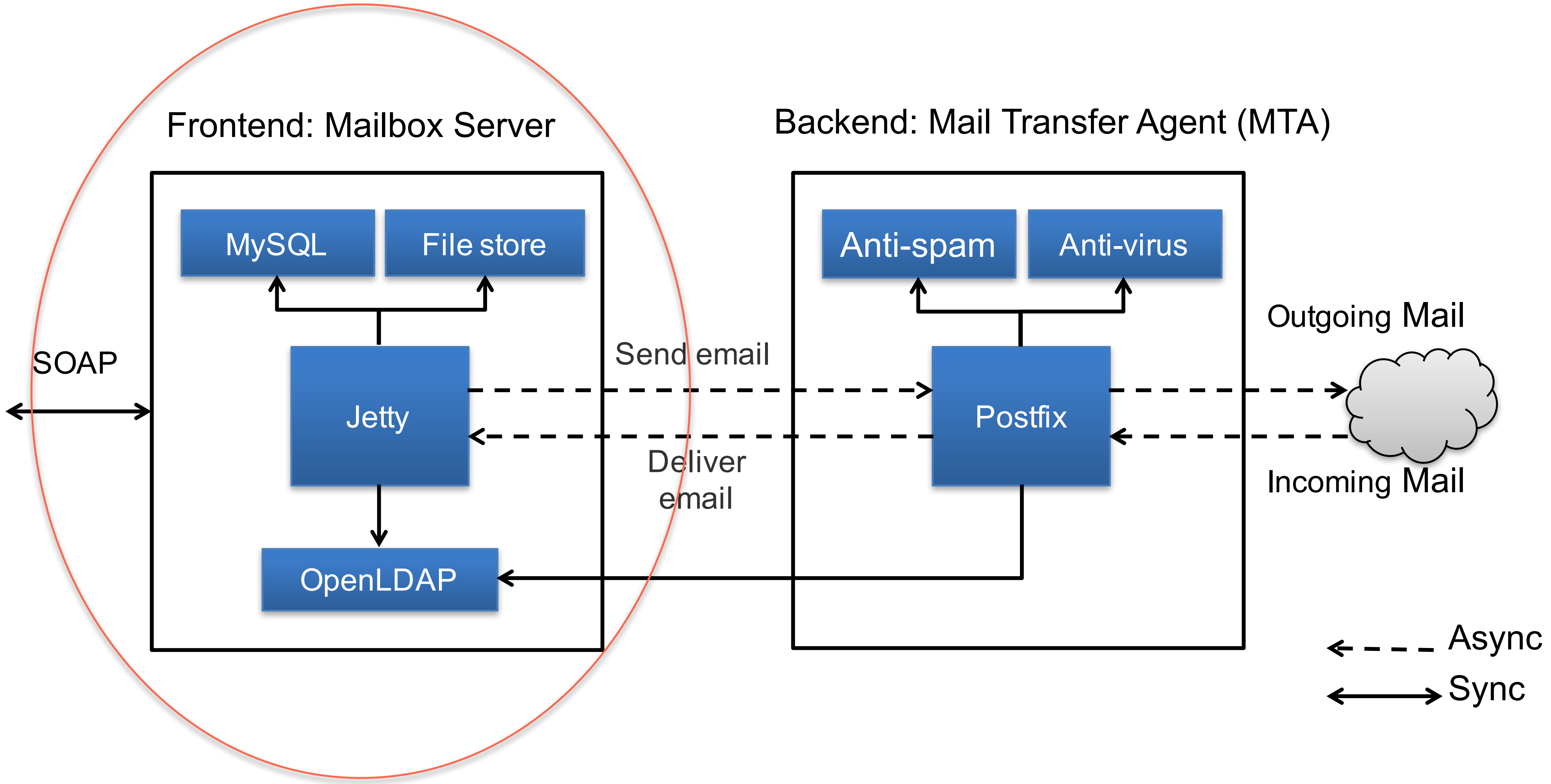
- Both scaling policies successfully avoid SLA violations
- **Model-based** policy is **more efficient** and requires **less reconfigurations**

** S. Spinner et al. "Runtime vertical scaling of virtualized applications via online model estimation." SASO 2014.*

Case study 3

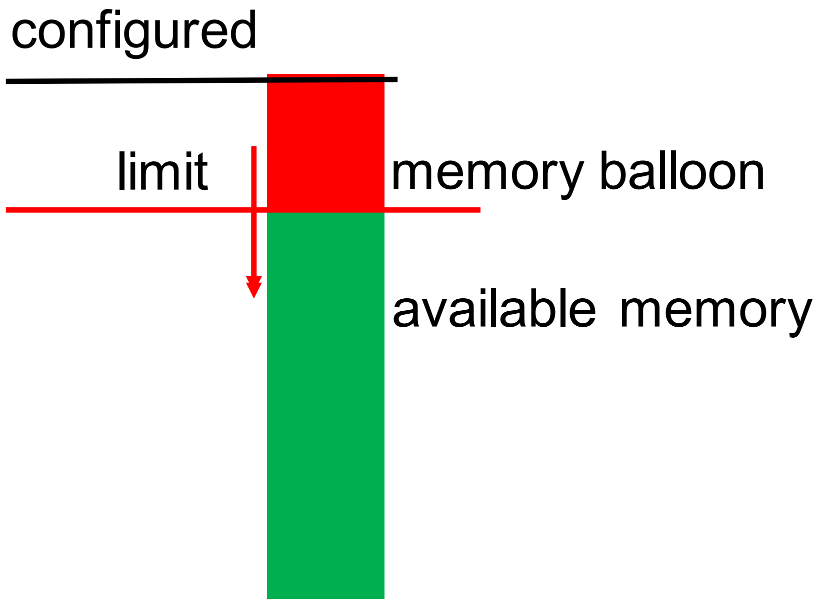
Proactive memory scaling for Zimbra Mailbox Server

- **Application:** Open-source Zimbra collaboration software
- **Workload: ZimbraPerf** email workload generator



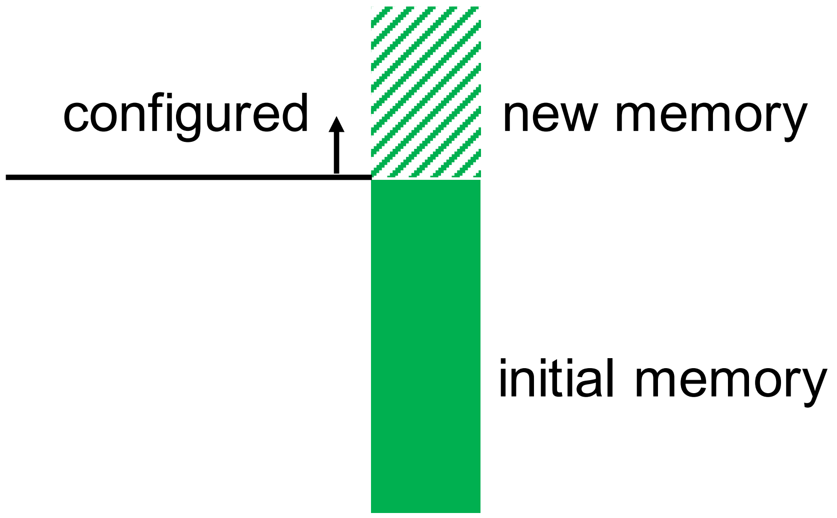
Memory scaling parameters

Ballooning



- Reclaim memory by reducing limit
- Limit ≤ configured

Hot-add



- Add memory by increasing configured
- No restart of VM required
- May require restart of application

Case study 3

Proactive memory scaling for Zimbra Mailbox Server

Challenges

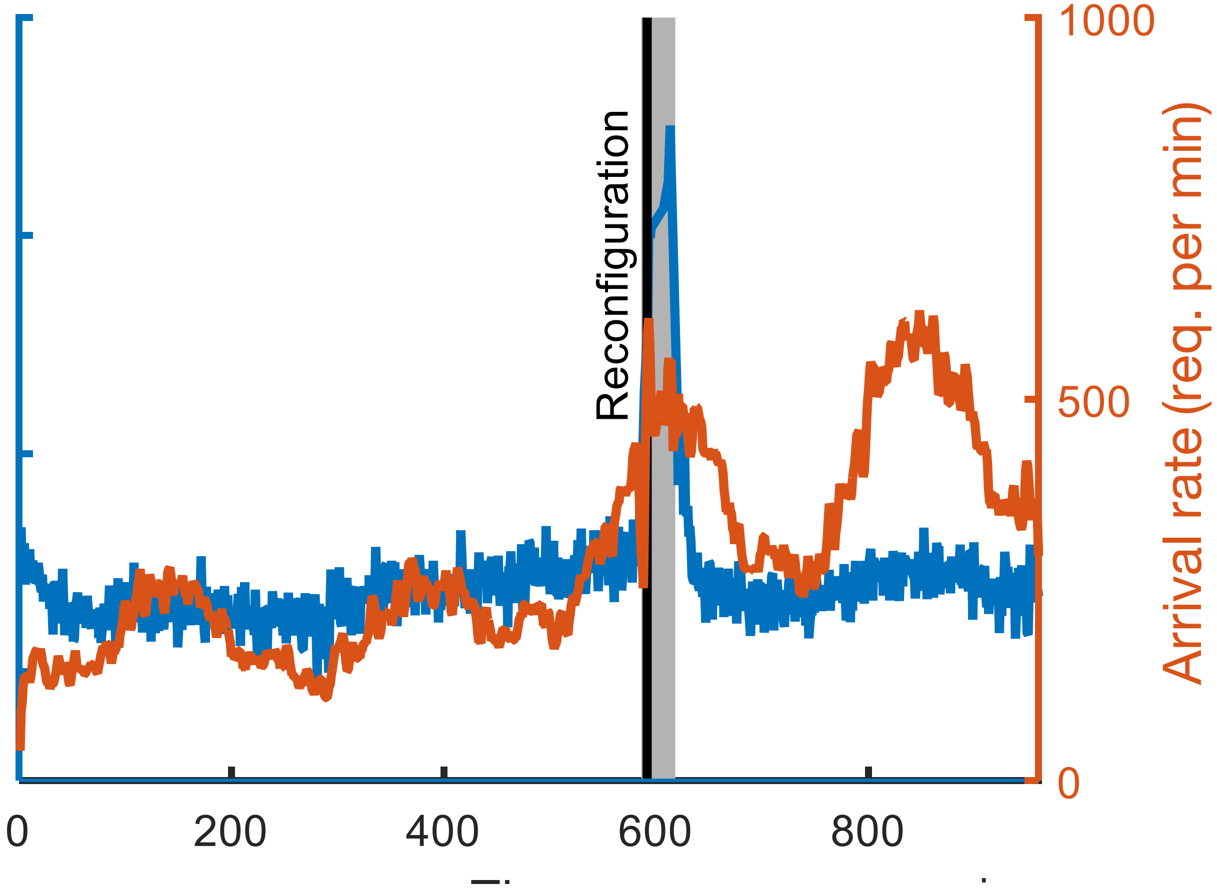
- Application memory management
  - Optimal configuration depends on VM memory size (e.g., JVM, MySQL)
- Application elasticity
  - Restart of application may be required when app memory settings change
- Impact of reconfiguration
  - May cause additional overheads
  - Unreliable under high memory pressure

Approach

- **Proactive scaling** of VM memory size
  - Use load forecasting to determine maximum required memory for the next day
  - Schedule memory reconfig. via hot-add during phases of low load
  - Minimize impact of reconfiguration on application

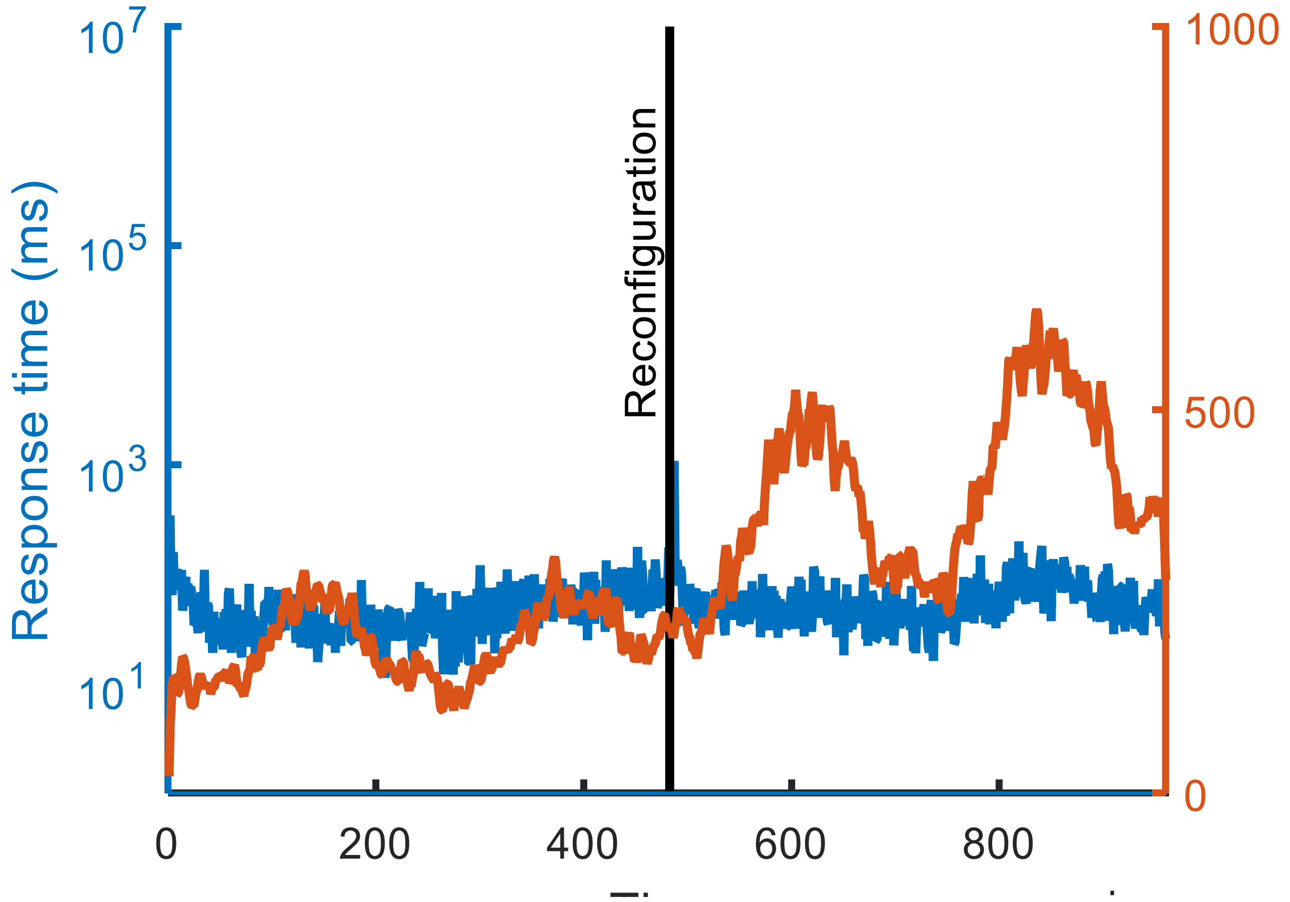
Performance evaluation result

Reactive controller



→ 33 min unavailable

Proactive controller



→ 4 min unavailable

— Response time ▨ Reduced availability

- **Proactive** memory scaling controller reduces application unavailability time **by over 80%** compared to the **reactive** controller

* *S. Spinner et al. "Proactive Memory Scaling of Virtualized Applications." IEEE CLOUD 2015.*

References

- X. Zhu, *et al.* “What does control theory bring to systems research?” *ACM SIGOPS Operating Systems Review, 43(1), January 2009.*
- P. Padala *et al.* “Automated control of multiple virtualized resources.” *Eurosys 2009.*
- A. Gulati *et al.* “VMware distributed resource management: Design, implementation, and lessons learned.” *VMware Technical Journal*, Vol. 1(1), April 2012.
- P. Xiong *et al*. “vPerfGuard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments.” *ICPE 2013*.
- L. Lu, *et al.*, “Application-Driven dynamic vertical scaling of virtual machines in resource pools.” *NOMS 2014*
- P. Padala *et al.* “Scaling of cloud applications using machine learning.” *VMware Technical Journal, Summer 2014.*
- S. Spinner *et al. “*Runtime vertical scaling of virtualized applications via online model estimation*.” SASO 2014.*
- *S. Spinner et al. “*Proactive Memory Scaling of Virtualized Applications.” *IEEE CLOUD 2015.*

Tutorial Outline

- Survey of datacenter resource schedulers
- Achieving Service Level Objectives (SLOs) via automatic application scaling
- **Analytics pipelines for workload telemetry data**

Analytics Pipelines for VM/Workload Telemetry

Presented by Rean Griffith

Based on joint work with Dragos Ionescu (intern/MIT)
and members of the Distributed Resource
Management (DRM) team at VMware

Outline

- Goals: sense-making at scale
- An interesting anomaly detection problem
- Conceptual Steps (4 Example Pipelines)
- Related Work
- Data Sources
- Pipeline Details and Results
- Conclusion

Goal: Do useful things with (lots of) data

- Make use/sense of datacenter telemetry
  - Understand VM-resource relationships
  - Understand VM-VM relationships
  - Understand VM-performance relationships
- Analyze non-trivial amounts of raw data (10’s – 100’s of GBs per day) “quickly” (not real-time) in a scalable way

Motivating Problem

- Anomaly detection in virtualized environments using limited history

Typical Anomaly Detection Steps

- Step 1: Observe the normal behavior of something long enough to construct a baseline
- Step 2: Compare future behavior to baseline and highlight deviations
- Step 3 (optional): Explain deviations (root cause)
- Problems:
  - Mis-labeled training data: How do you know you are observing normal behavior?
  - How long is long enough?
  - What can you say while you are waiting?
  - What does an explanation look like?

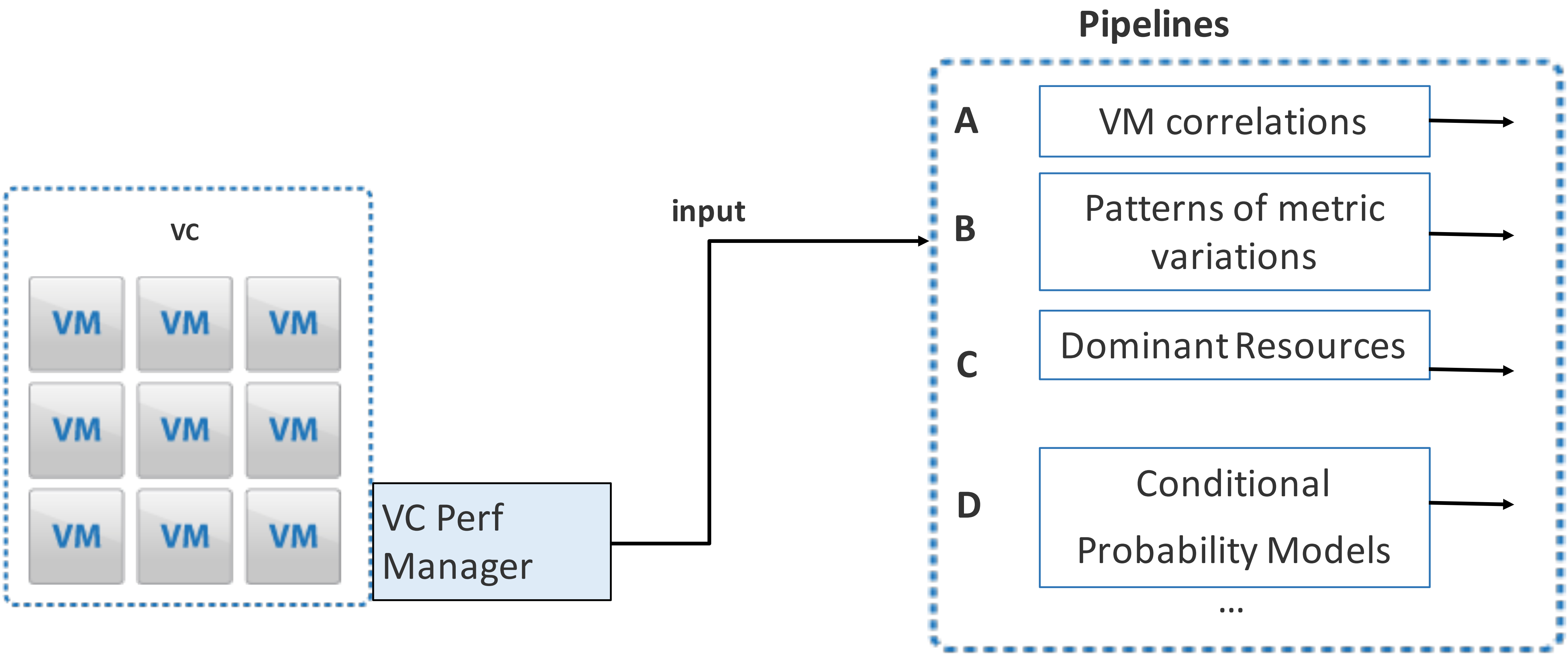
Insight: VM-Similarity Relationships Can be Useful and Robust

- Intuition: virtual machines running the same operating system or performing the same role are more like each other
- For example: a VM running apache on Linux is more like another VM running apache on Linux than an VM running mysql on Linux
- We can do similarity comparisons with smaller amounts of data (history)!

Outline

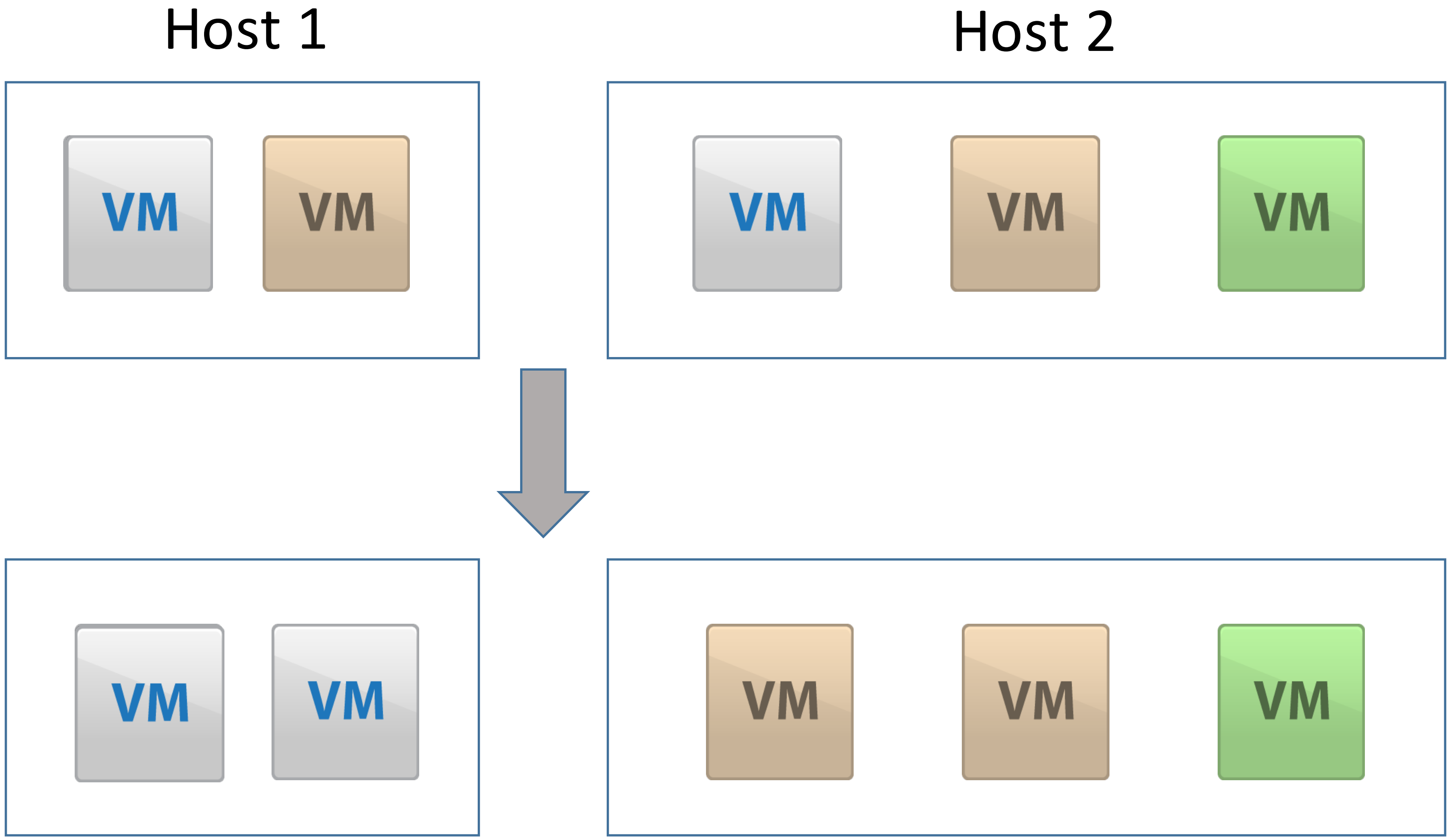
- Goals: sense-making at scale
- An interesting anomaly detection problem
- Conceptual Steps (4 Example Pipelines)
- Related Work
- Data Sources
- Pipeline Details and Results
- Conclusion

Conceptual Pipeline



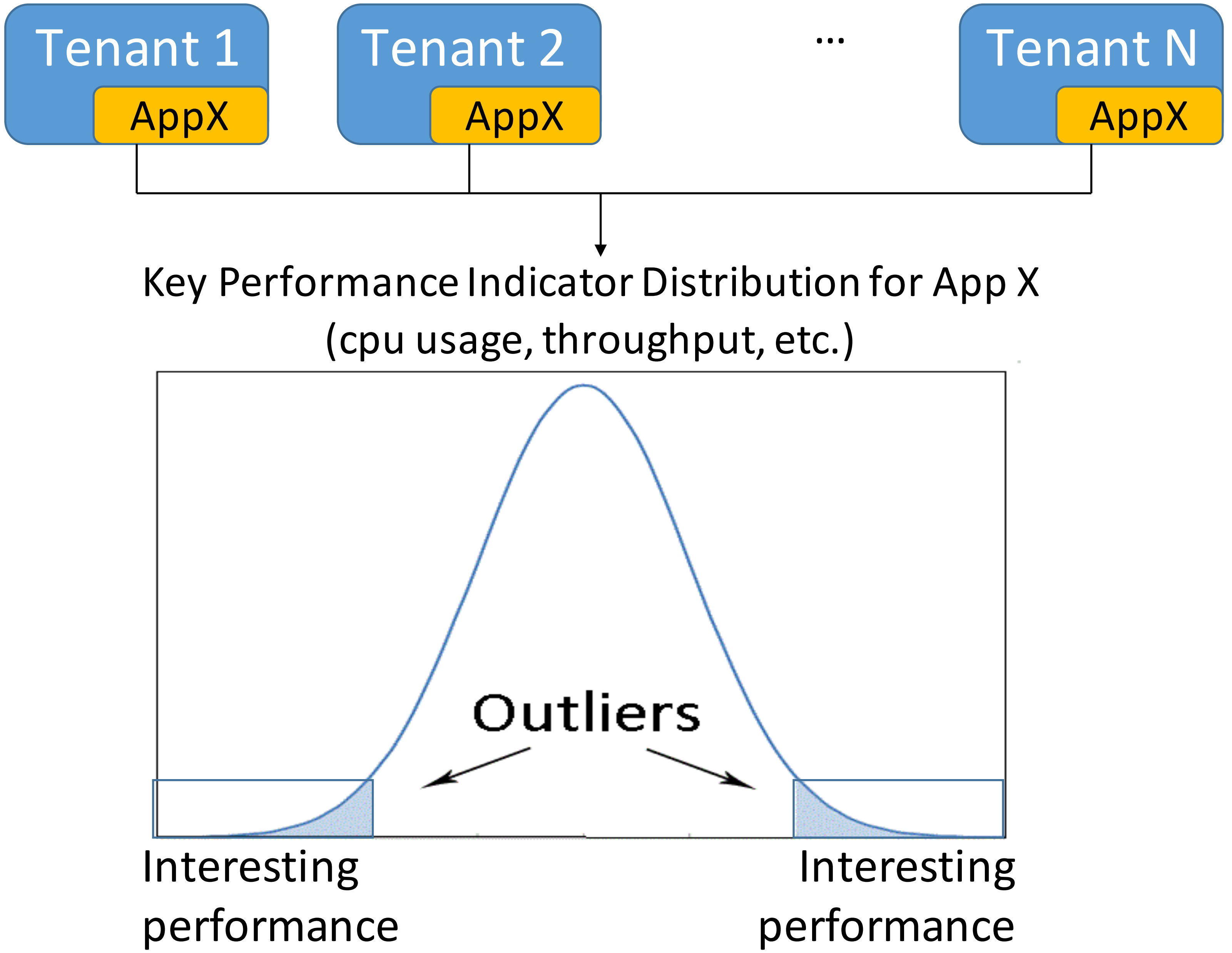
Motivation (Use Cases): Exploiting VM relationships

- Automatic assignment of Distributed Resource Scheduler (DRS) affinity / anti-affinity rules based on VM workload changes
- Automatic grouping of VMs based on their behavior (metric patterns)



Motivation (Use Cases): Explaining Performance

- Automatic detection and explanation of performance problems or differences across deployments using Conditional Probability Distributions



Key Performance Indicator Distribution for App X
(cpu usage, throughput, etc.)

Outline

- Goals: sense-making at scale
- An interesting anomaly detection problem
- Conceptual Steps (4 Example Pipelines)
- Related Work
- Data Sources
- Pipeline Details and Results
- Conclusion

Related Work: Fingerprinting the Datacenter

- Paper: **Fingerprinting the Datacenter**: Automated Classification of Performance Crisis (*Peter Bodík et al., EuroSys ‘10*)
- Goal: rapid identification of performance crises in a datacenter and rapid recovery from a crisis
- Uses a fingerprint to represent the state of the datacenter (classification problem)
  - The fingerprint is easy to compute (scales linearly with the number of performance metrics considered not the number of machines)
  - The fingerprint captures the most relevant metrics that can be used to describe or diagnose a crisis

Related Work: Using Correlated Surprise to Infer Shared Influence

- Paper: **Using Correlated Surprise to Infer Shared Influence** (*Adam Oliner et al., DSN ’10*)
- Goal: design a method for identifying the sources of problems in complex production systems based on influence
- Influence: two component share an influence if the exhibit surprising behavior around the same time
- This approach motivates the use of VM correlations in our analysis pipeline

Related Work: Online detection of Multi-Component Interactions

- Paper: **Online detection of Multi-Component Interactions in Production Systems** (*Adam Oliner et al., DSN ’11*)
- Goal: online identification of sources of problems in complex systems based on historical data
- The paper shows that understanding complex relationships between heterogeneous components reduces to studying the variance in a set of signals
- This approach motivates the use of VM metric variations in our analysis pipeline

Related Work: Dominant Resource Fairness

- Paper: **Dominant Resource Fairness: Fair Allocation of Multiple Resource Types** (*Ali Ghodsi et al., NSDI '11*)
- Goal: fair resource allocation in a system containing different resource types and different demands
- The paper focuses on fairness policies, but we can reuse the notion of a dominant resource to group VMs and get a possible fingerprint (e.g. the dominant resource is the resource a VM cares about the most)
- This approach motivates the use of dominant resource patterns in our pipeline

Related Work: Carat: Collaborative Detection of Energy Bugs

- Paper: **Carat: Collaborative Detection of Energy Bugs** (*Adam Oliner et al., carat.cs.berkeley.edu, SenSys '13*)
- Goal: use the idea of an Application Community to do collaborative detection of energy bugs for smart phones
- Application Community: collection of multiple instances of the same application (or similar applications) running in different environments
- Battery usage data is collected from the entire community and used to identify possible energy hogs (applications or settings that lead to a higher discharge rate) – P(drain rate = x | wireless off, AppA running, …)
- This approach motivates our use of Conditional Probability Distributions to identify performance problems across deployments

Outline

- Goals: sense-making at scale
- An interesting anomaly detection problem
- Conceptual Steps (4 Example Pipelines)
- Related Work
- Data Sources
- Pipeline Details and Results
- Conclusion

Data Sources

- Data Collection Clusters
  - AppRM (DRM Cluster) – smaller controlled environment (~10 VMs), research workloads - 2 small mongoDB clusters and unrelated other workloads
  - ViewPlanner (Perf Team Cluster) – medium size controlled environments (100’s of VMs), controlled workload
  - Nimbus – large number of VMs (~1000 VMs), heterogeneous workloads

Pipeline Input

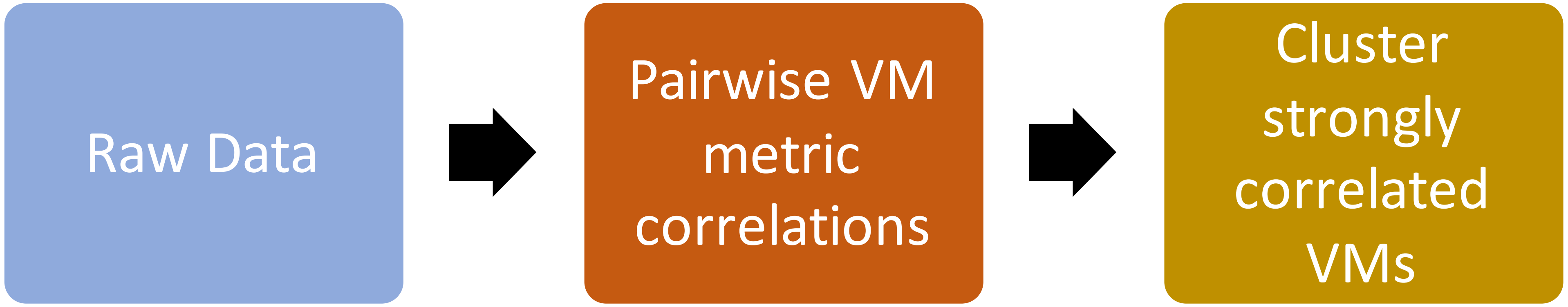
- Collected ~300 metrics per VM
  - Includes CPU, Memory, Disk, Network statistics at the VM, hypervisor and Guest OS level. Also includes hypervisor statistics, e.g., memory ballooning and hypervisor swap data
- On average ~100 metrics per VM were non-constant

Data Exploration and Preparation

- Algorithmic tools
  - Metric correlations
  - Percentile summarization
  - K-Means clustering + Bayesian Information Criterion + Cluster Stability Measures + Cluster Spread/Diffusion metrics
  - Logistic Regression (Classification)
  - Principal Component Analysis (PCA)
  - Multi-modal Distribution Analysis (Silverman's test)
  - Entropy and Mutual Information computations

Pipeline A – VM Correlations

- Use VM correlations for a rough approximation of VM relationships
- Based on the results, assess whether it is worth applying more advanced statistical clustering techniques



Raw Data

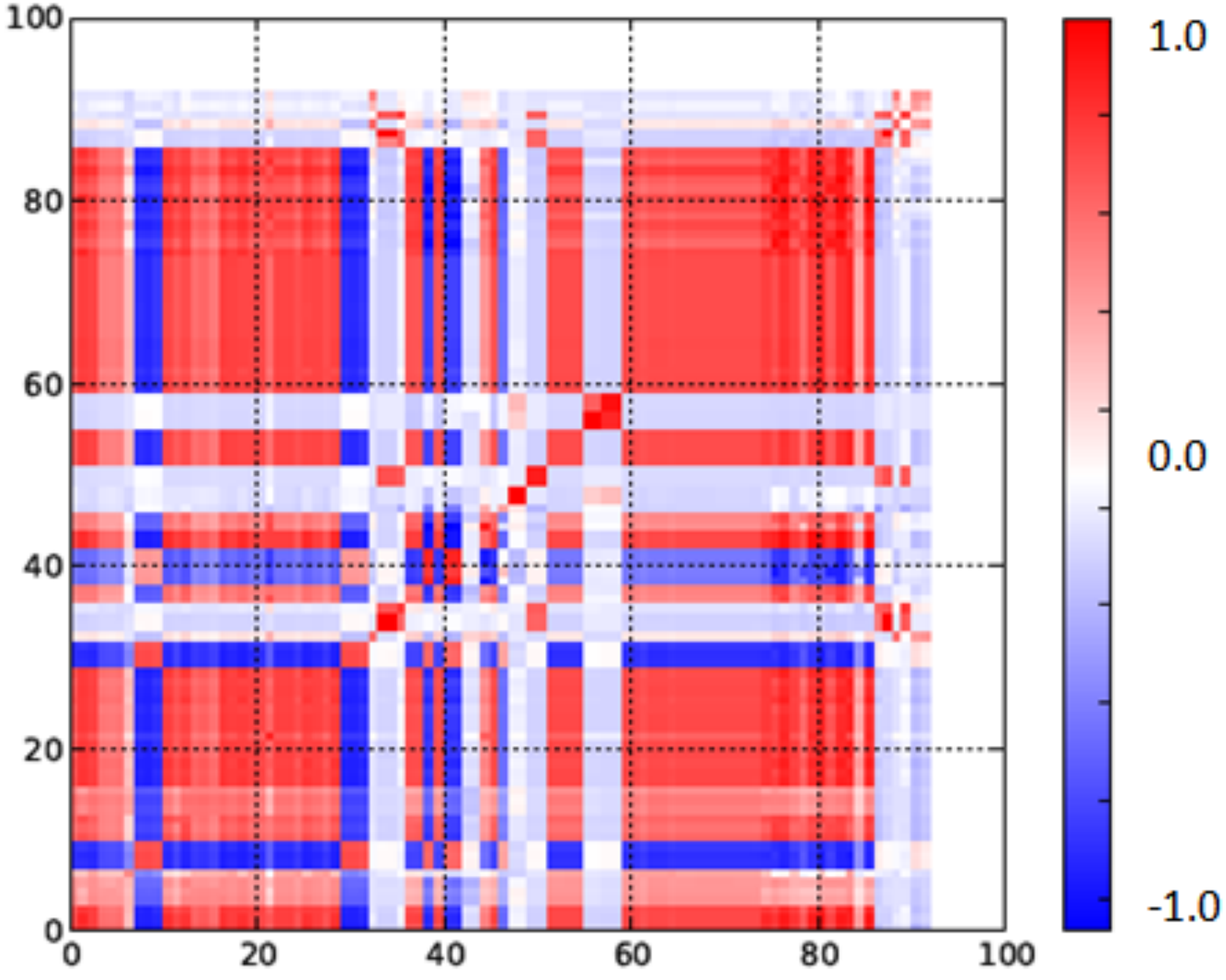
Pairwise VM metric correlations

Cluster strongly correlated VMs

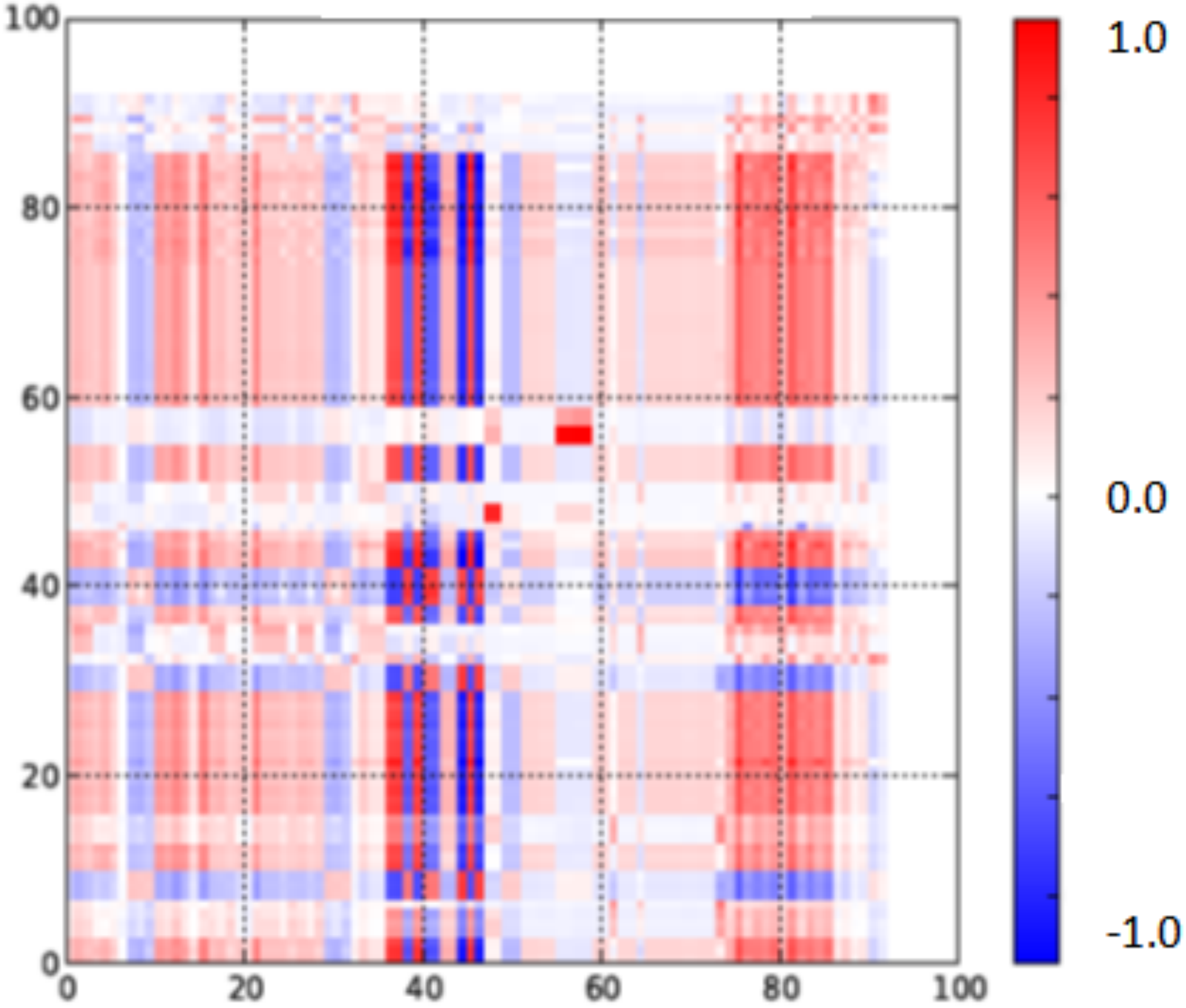
Pipeline A Results – Correlations across VM metrics

- Choose a set of n relevant metrics (via correlation coefficient thresholds) for VM_1 and VM_2
- Build an NxN correlation matrix M, such that $M_{i,j}$ is the correlation between the values of the i^{th} metric for VM_1 and the j^{th} metric for VM_2 (using a hour’s worth of raw data)
- Look for patterns

Pipeline A Results – Correlations across VM metrics (AppRM)



2 MongoDB VMs

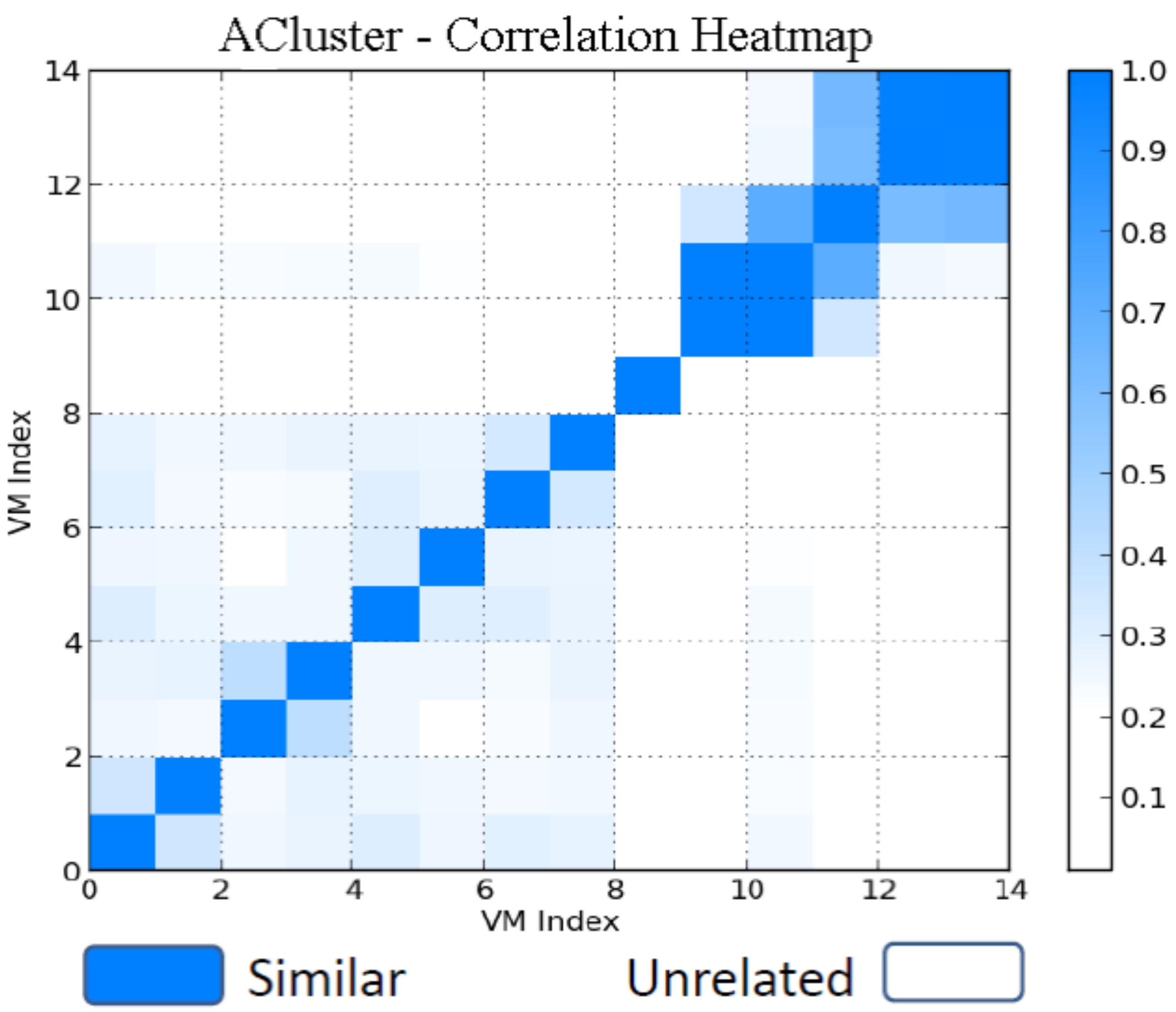


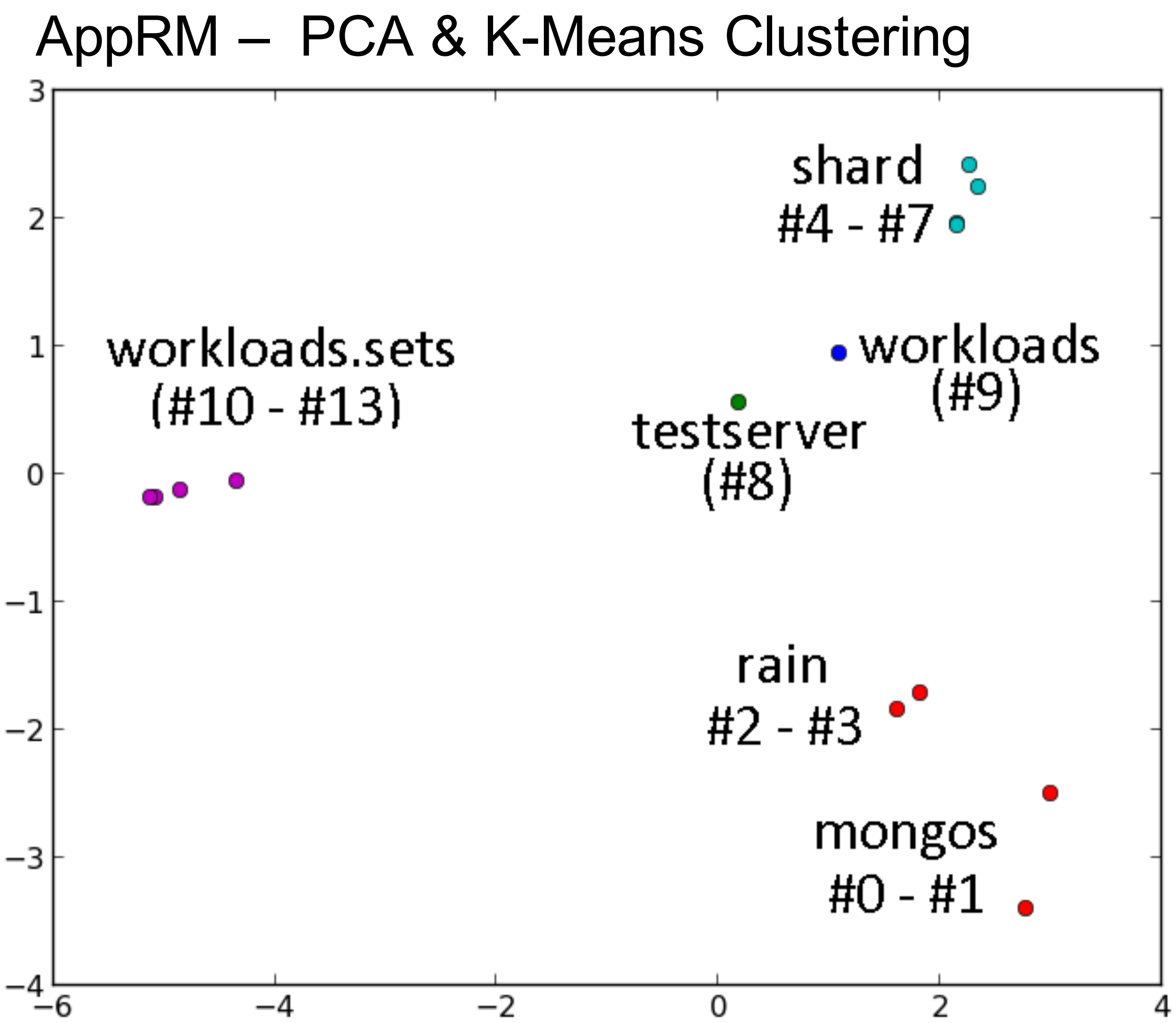
1 MongoDB VM and
1 unrelated testserver VM

Pipeline A Results – From Correlations to Clustering

- Given a pair of VMs, count for how many metrics the correlation coefficient is above a given threshold (0.4)
- Normalize using the highest count
- Step 1: Naïve clustering based on normalized count:
  - Count close to 1: high fraction of correlated metrics
  - Count close to 0: low fraction of correlated metrics
- Step 2: Cluster based on K-Means if Step 1 looks promising
  - Each value used as a coordinate component for a set of metrics $<m_1, m_2, \ldots, m_n>$ is its k-percentile value (e.g., median/50th percentile) over a 1 hr period

Pipeline A Results – From Correlations to Clustering


ACluster - Correlation Heatmap


AppRM – PCA & K-Means Clustering

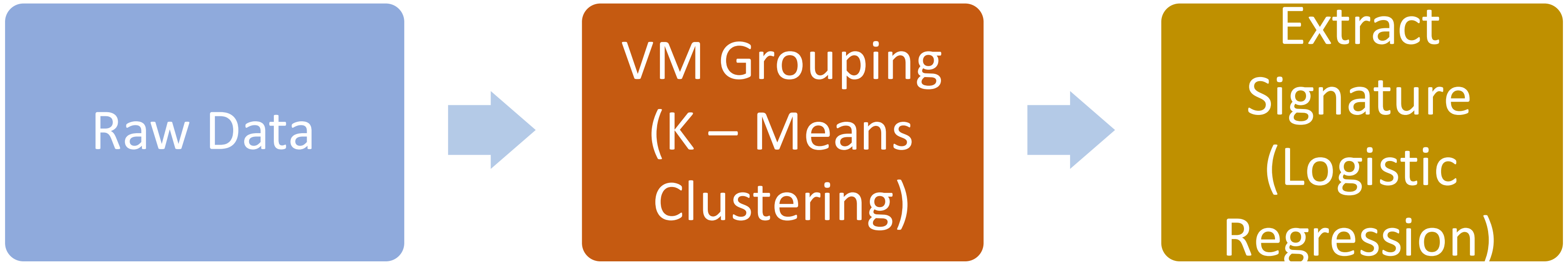
VM 0 – 7: MongoDB cluster + rain load generators
VM 8: unrelated test server
VM 9: workload A
VM 10 – 13: workload B

Pipeline B - Background

- K-means clustering
  - Unsupervised Machine Learning technique to identify structure in a dataset
  - Used to identify the VMs that are "similar" (group together)
- Logistic regression
  - Classification technique used to identify the features that describe a labeled set of datapoints
  - Used to describe the structure (statistical clusters) from K-Means Clustering algorithm by identifying the relevant features for each cluster
- Principal Component Analysis (PCA)
  - Data summarization technique used to explain the variance of a set of signals using the variance of a subset of them
  - Can be used to identify the key (principal) components and eliminate redundant features

Pipeline B – Patterns of Metric Variations

Show good results using the raw data



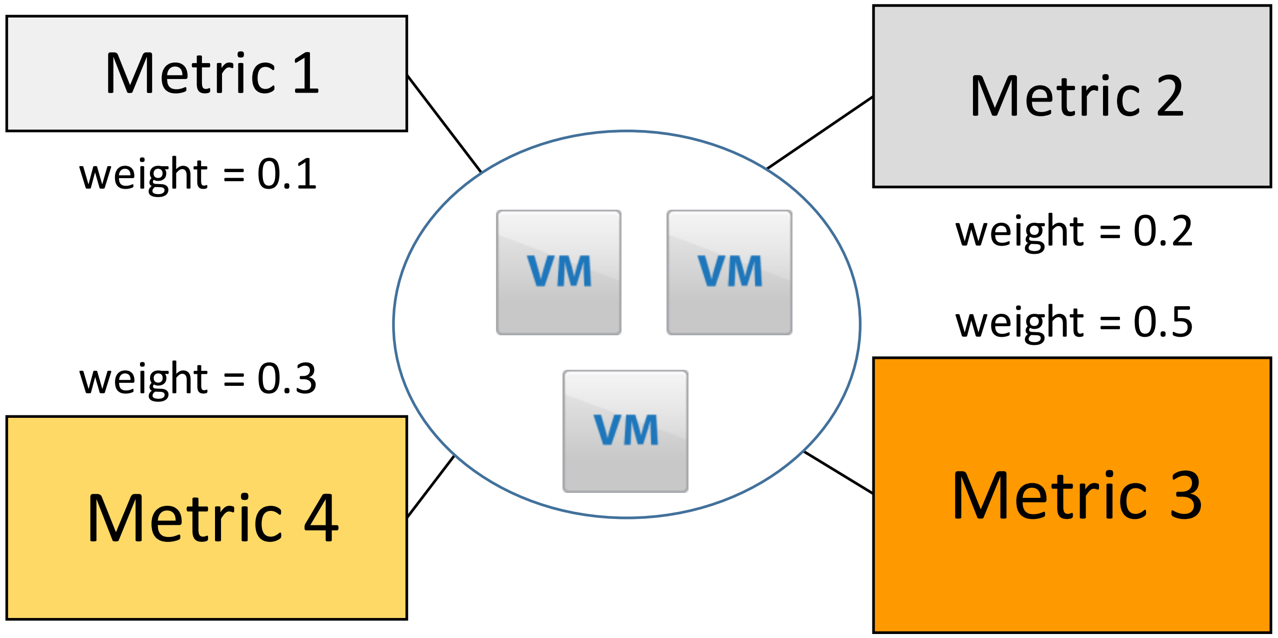
Raw Data

VM Grouping (K – Means Clustering)

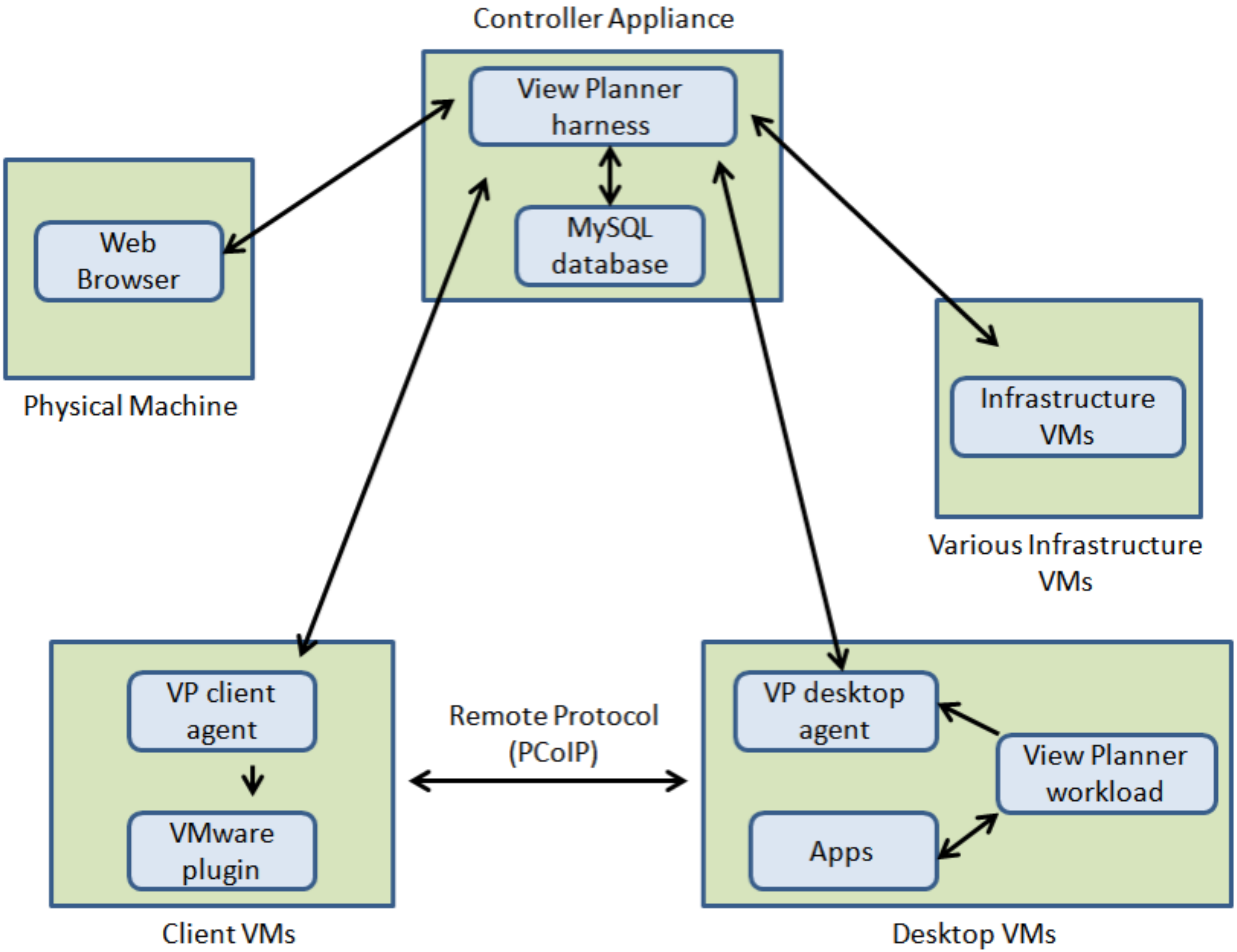
Extract Signature (Logistic Regression)

Pipeline B – Fingerprint Example

- One-vs-all Logistic Regression is used to identify the most important metrics for each cluster
- The resulting fingerprint is a weighted expression of common metrics of importance

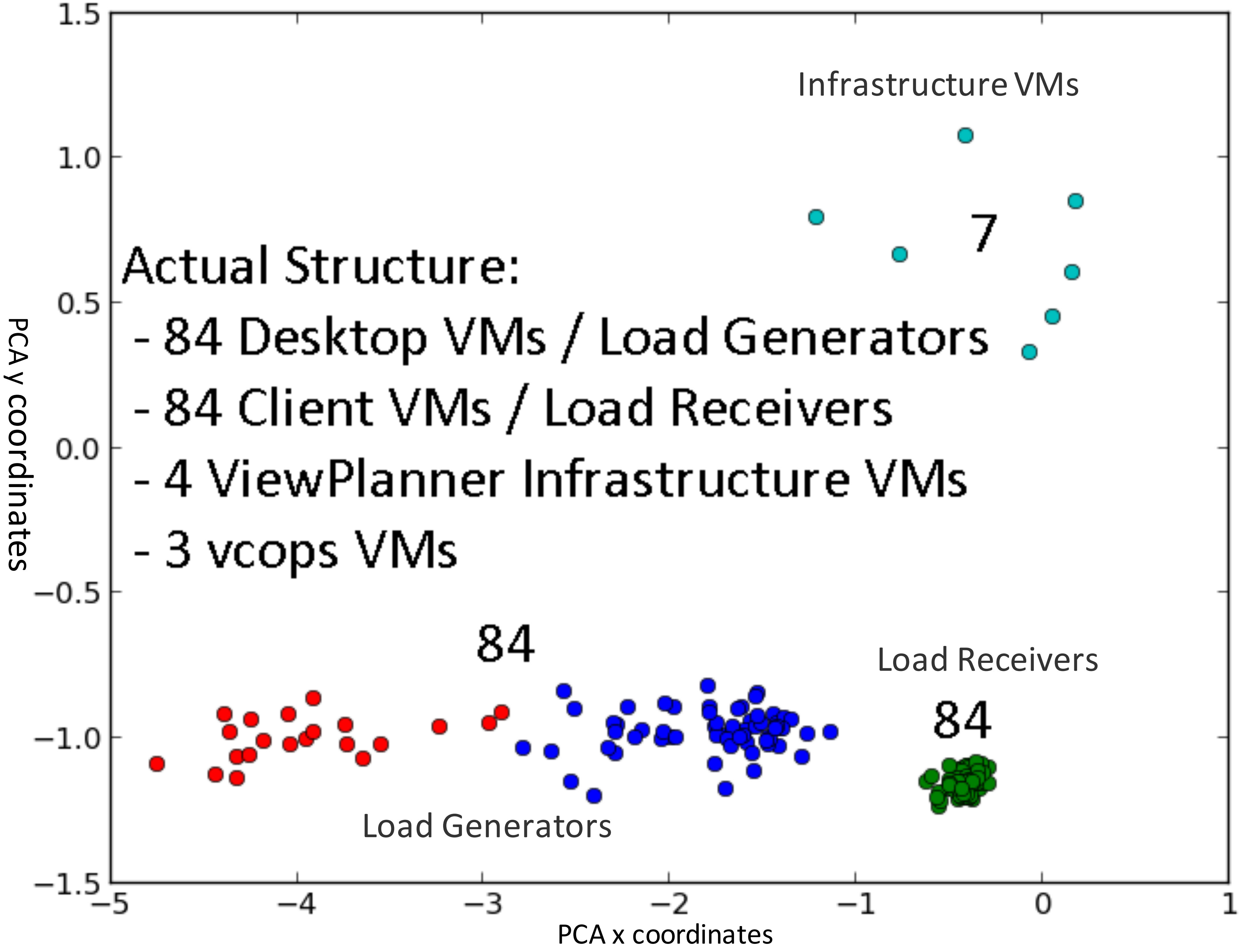


Pipeline B – ViewPlanner Setup



Pipeline B Results – ViewPlanner

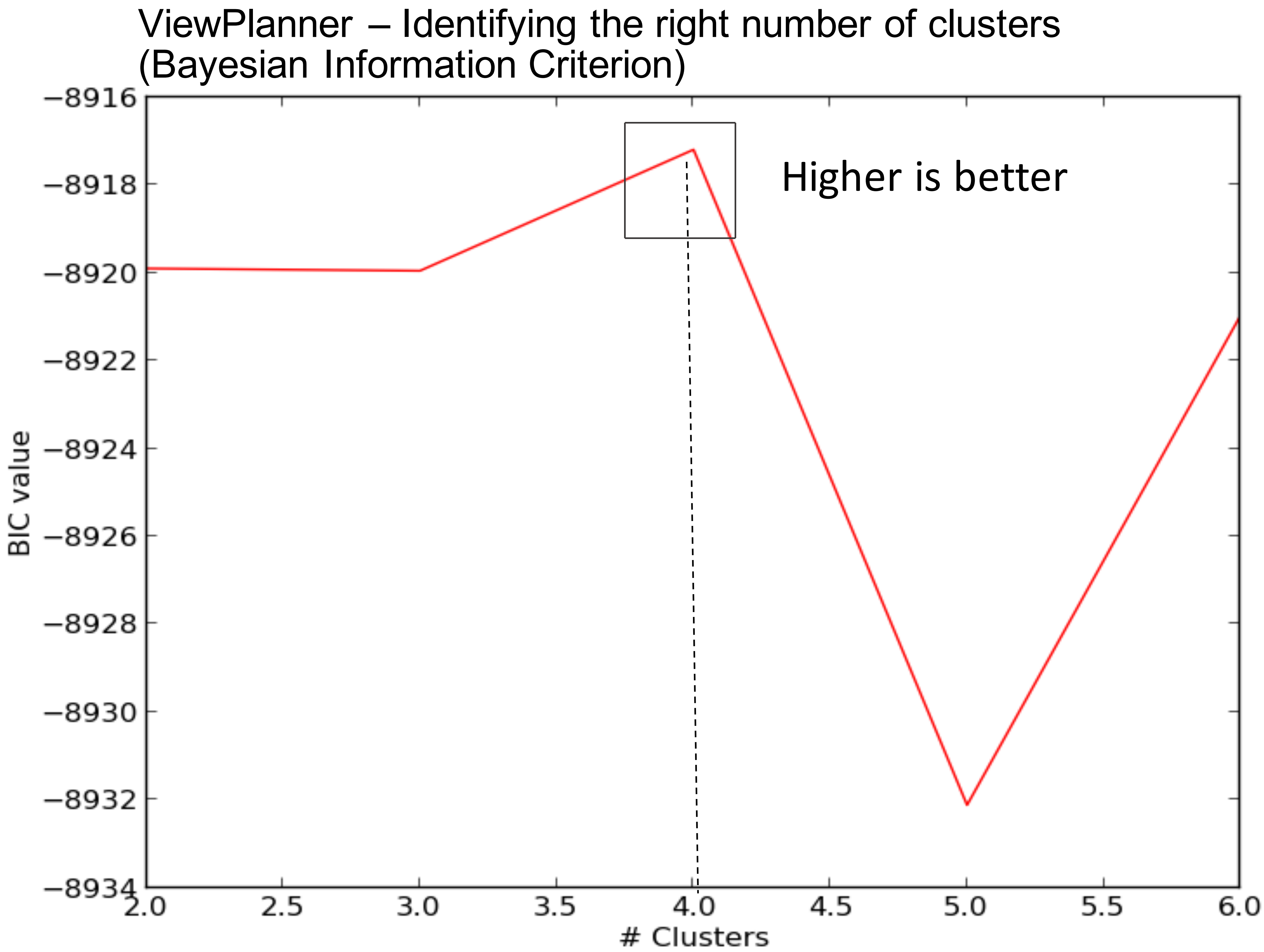
ViewPlanner – PCA & K-Means Clustering



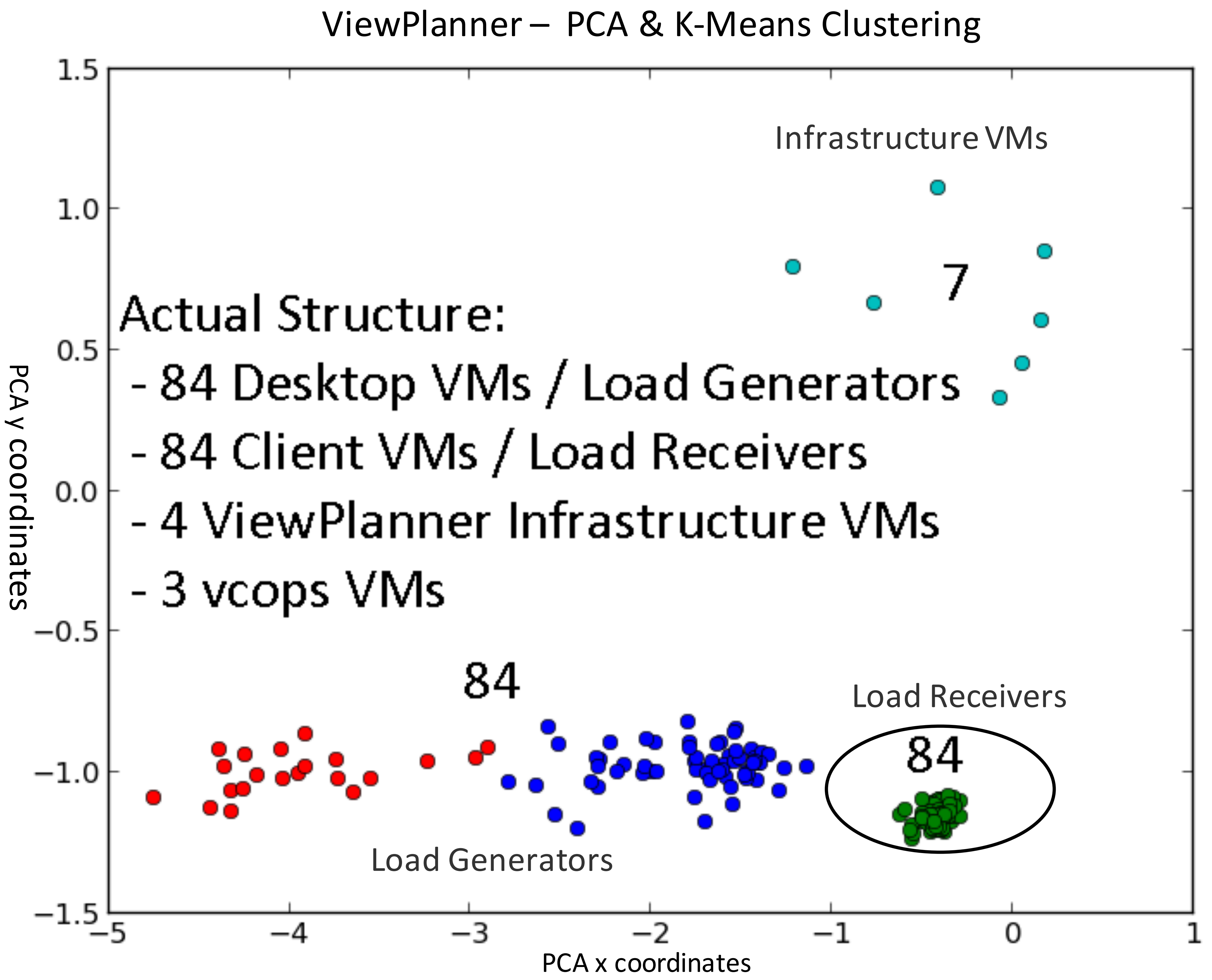
Actual Structure:

- 84 Desktop VMs / Load Generators
- 84 Client VMs / Load Receivers
- 4 ViewPlanner Infrastructure VMs
- 3 vcops VMs

Pipeline B Results – Choosing the number of clusters



Pipeline B Results – ViewPlanner



Actual Structure:

- 84 Desktop VMs / Load Generators
- 84 Client VMs / Load Receivers
- 4 ViewPlanner Infrastructure VMs
- 3 vcops VMs

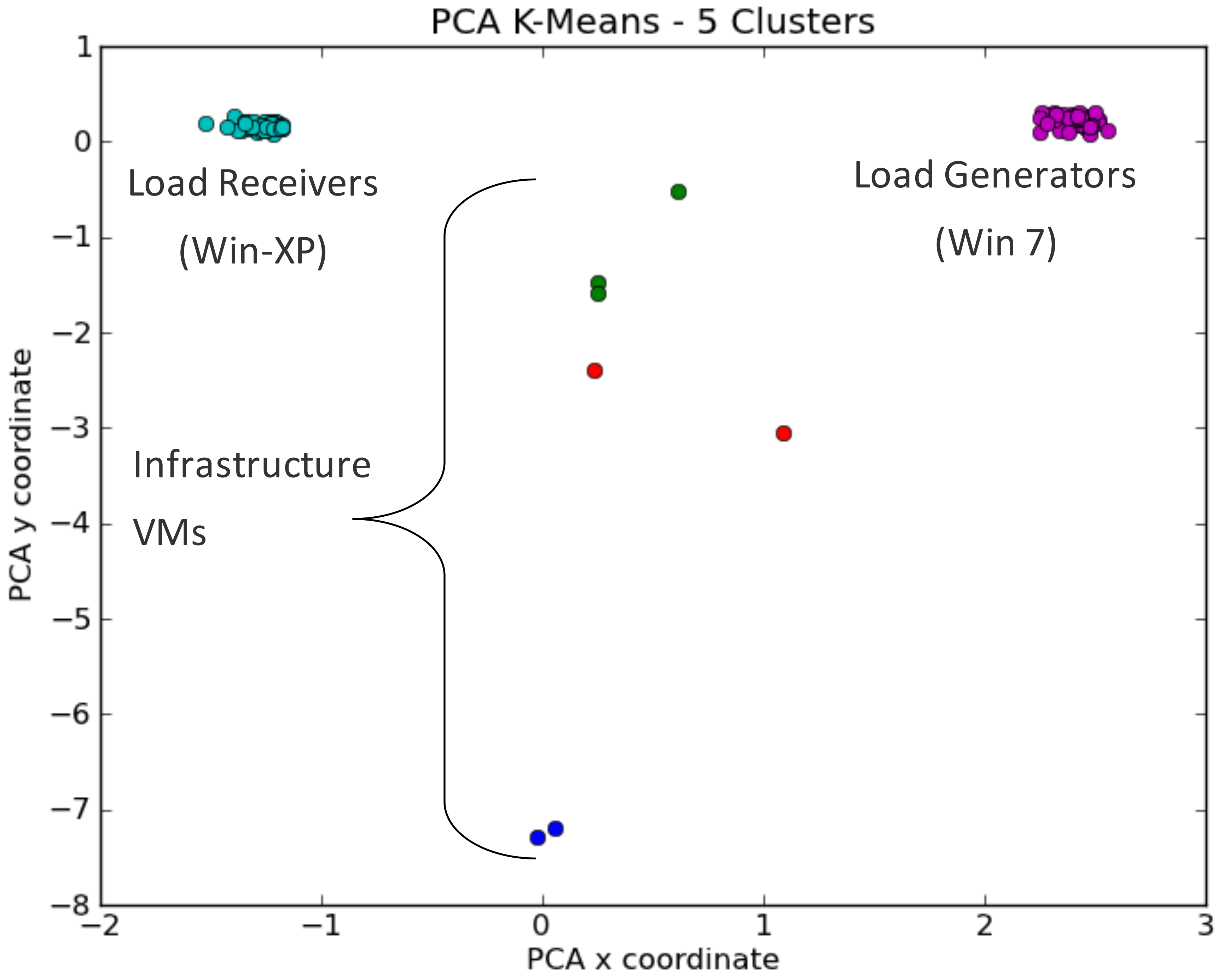
Pipeline B Results – Extracting the Fingerprint

ViewPlanner Desktop VMs (Green) Cluster Signature
Load Receivers

Metric	Coefficient
rescpu.runpk1.latest	1.20
cpu.system.summation	1.10
mem.usage.none	0.89
net.broadcastRx	0.78
cpu.usagemhz.none	0.69

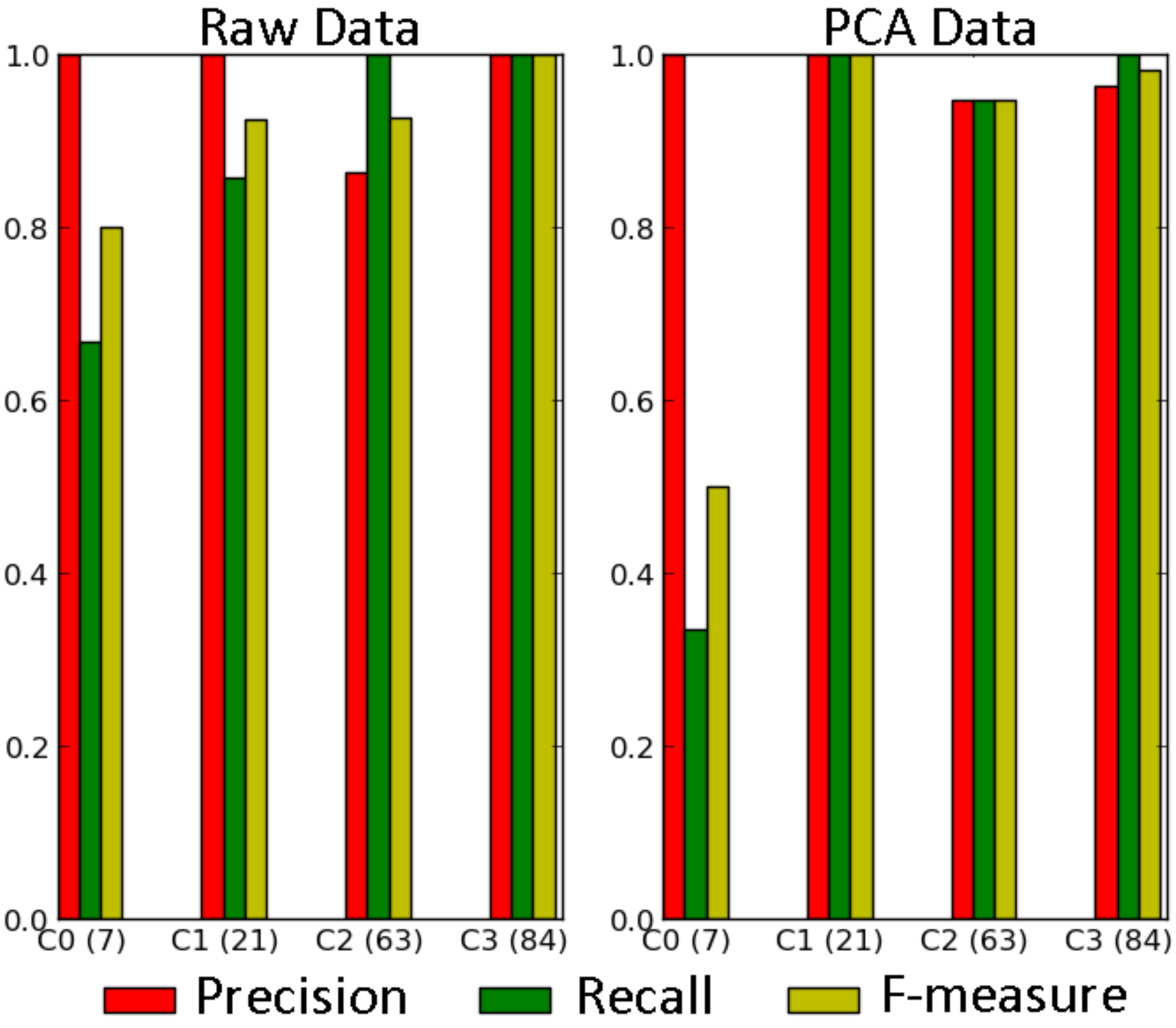
Pipeline B Results – ViewPlanner (idle state)

- We can still identify the VMs even though they are idle



Pipeline B Results– Fingerprint Accuracy (ViewPlanner)

- Use One-VS-All Logistic regression to measure the information loss due to PCA by comparing precision, recall, f-measure values



$$\text{Precision} = \frac{tp}{tp+fp}$$

$$\text{Recall} = \frac{tp}{tp+fn}$$

$$\text{F-measure} = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

tp = true positives

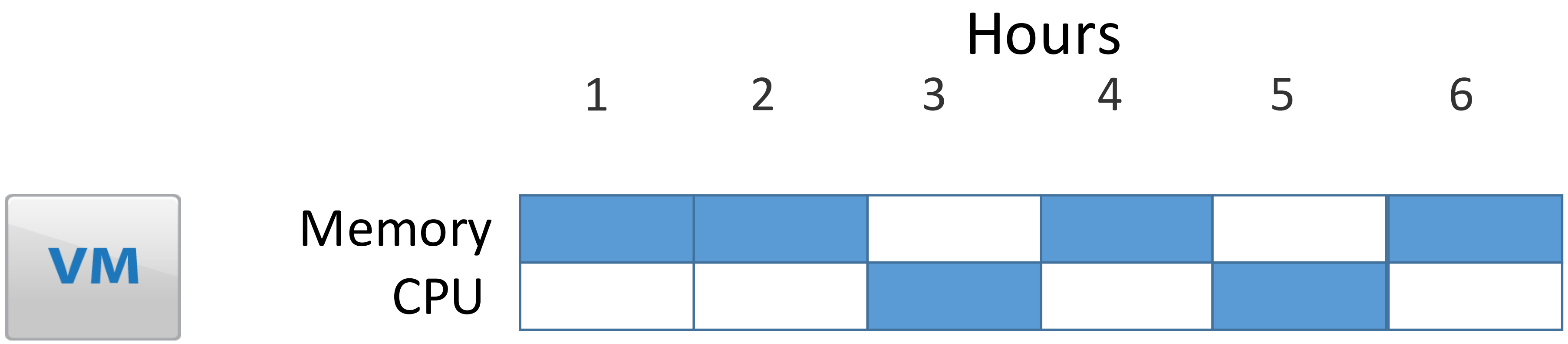
fp = false positives

tn = true negatives

fn = false negatives

Pipeline C – Dominant Resources

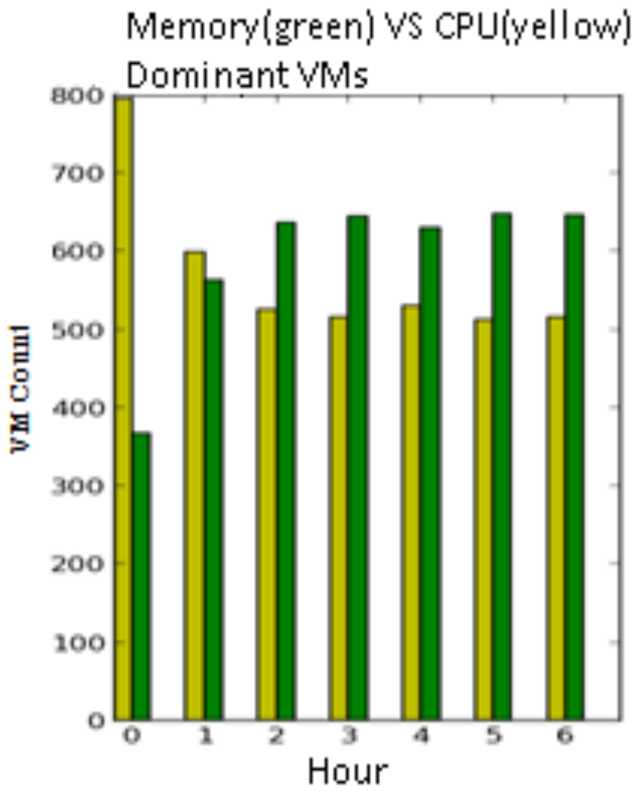
- Given a time interval, compute for each VM the percentage of CPU/Memory usage with respect to cluster's capacity
- The resource for which the VM requests the highest fraction of cluster resources is its dominant resource
- Fingerprint Example



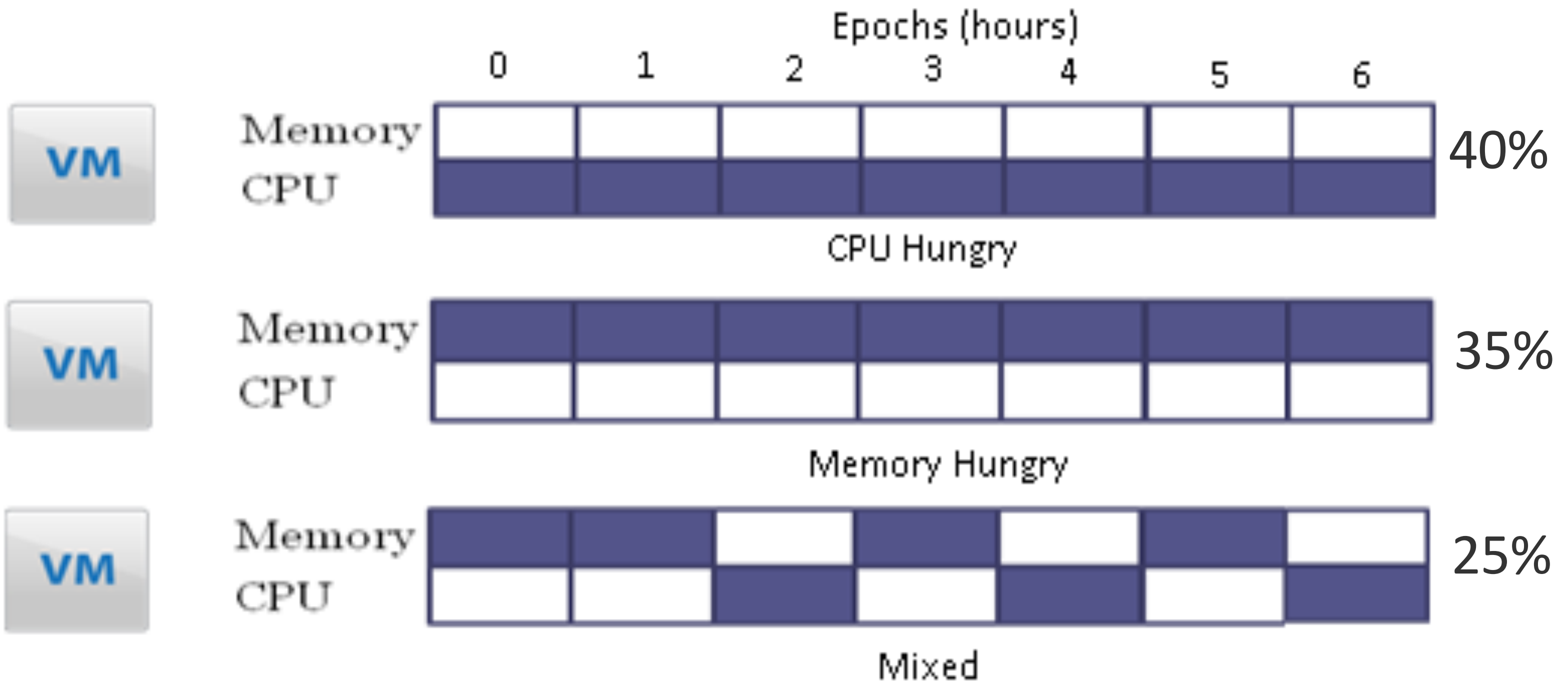
Hours

	1	2	3	4	5	6
Memory	■	■		■		■
CPU			■		■	

Pipeline C Results – Dominant Resources (Nimbus)



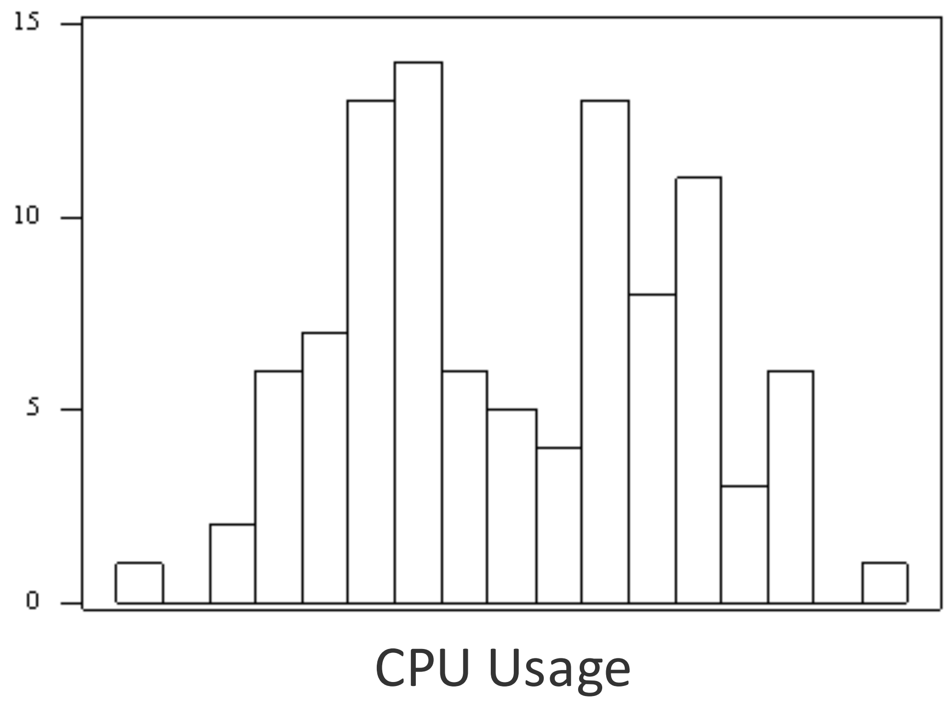
Memory(green) VS CPU(yellow)
Dominant VMs



Nimbus (~1000 VMs) - 3 common patterns

Pipeline D – Conditional Probability Distributions from Multi-modal data

Bimodal Distribution

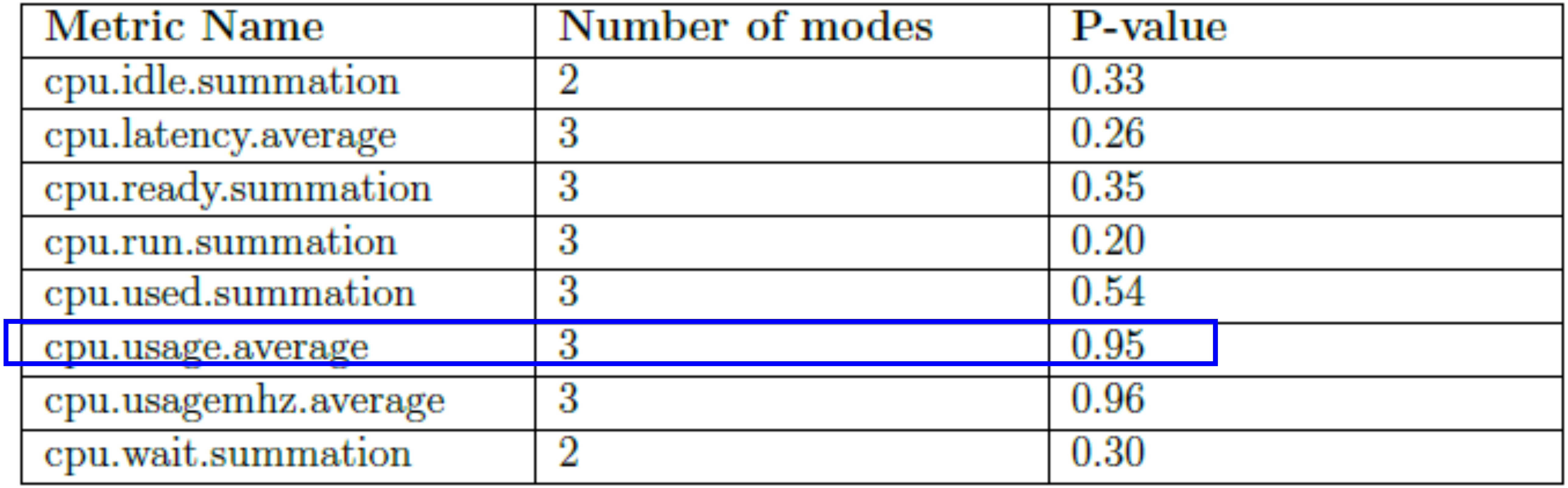


Pipeline D – Conditional Probability Distributions from Multi-modal data

- Two challenges (we want $P(X|Y_1, \ldots, Y_n)$)
  - Q1: How to find candidate interesting variables (X's)?
  - Q2: How to determine which variables to condition on (Y_i's)?
- Two possible strategies
  - A1: Multi-modal metrics may be interesting (use Silvermans Test)
  - A2: Use Mutual Information to exclude independent variables

Pipeline D – Conditional Probability Distributions from Multi-modal data

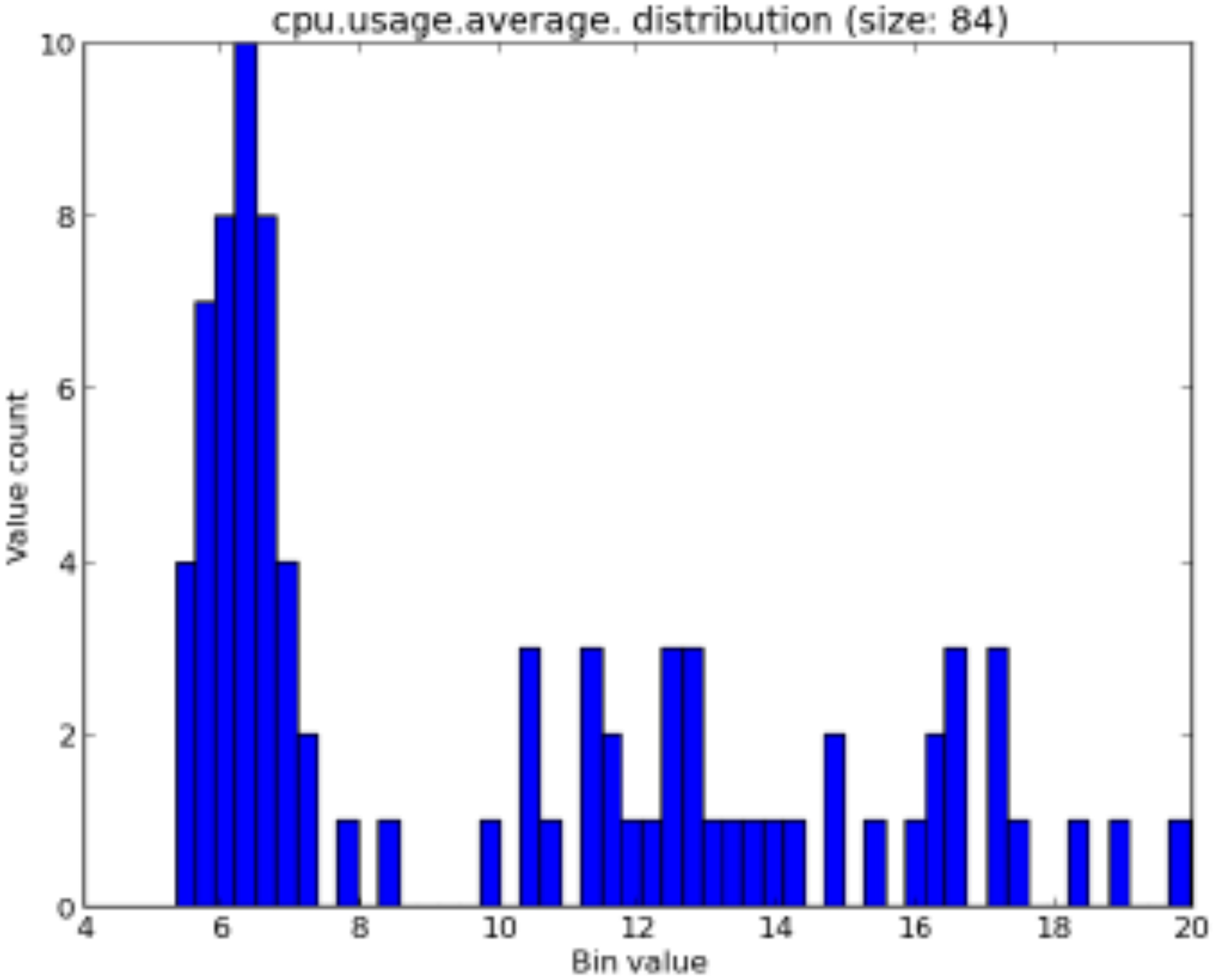
- Using data from other View Planner experiments we find example multi-modal metrics (using Silvermans test and a 0.1 significance level)



Metric Name	Number of modes	P-value
cpu.idle.summation	2	0.33
cpu.latency.average	3	0.26
cpu.ready.summation	3	0.35
cpu.run.summation	3	0.20
cpu.used.summation	3	0.54
cpu.usage.average	3	0.95
cpu.usagemhz.average	3	0.96
cpu.wait.summation	2	0.30

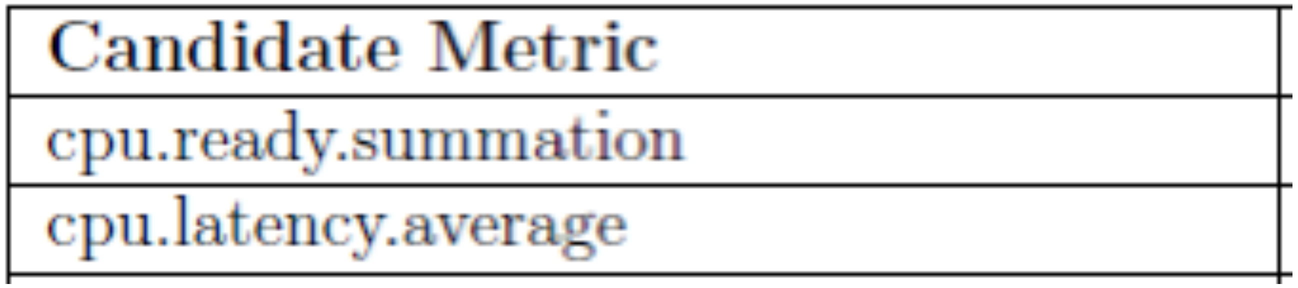
Pipeline D – Conditional Probability Distributions from Multi-modal data

- cpu.usage.average distrbution



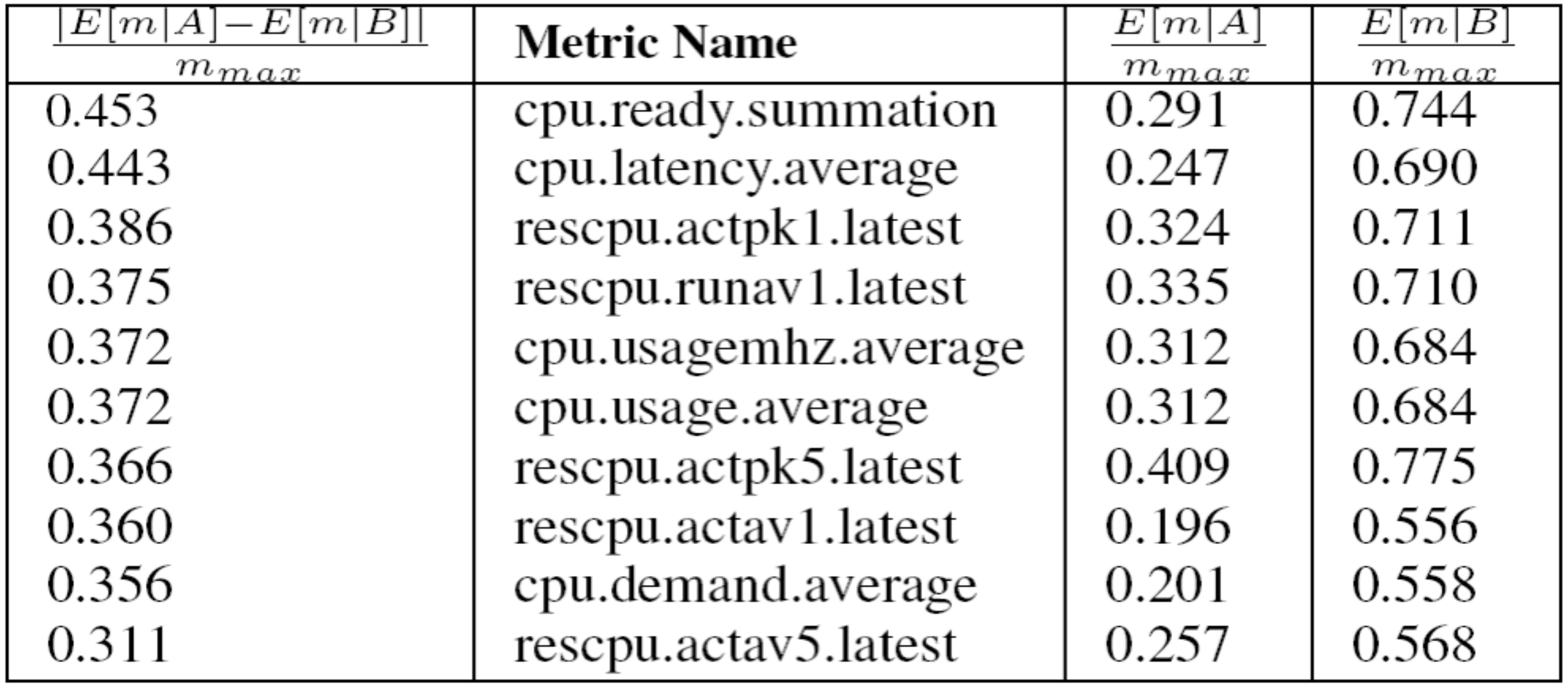
Pipeline D – Conditional Probability Distributions from Multi-modal data

- Identify candidate metrics to condition on via Mutual Information



Candidate Metric
cpu.ready.summation
cpu.latency.average

- We identify spread metrics from the normalized delta of expected values across View Desktop clusters (m_{max} = max median value over split cluster of View Desktops)

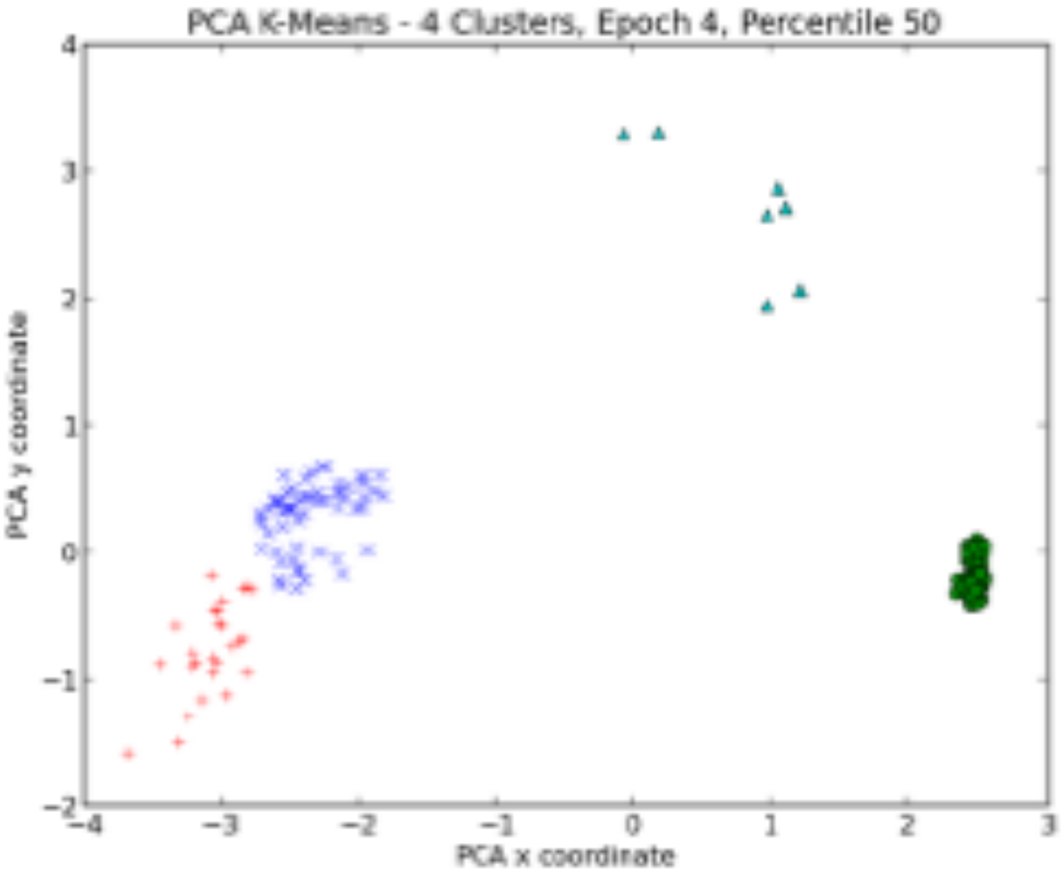


$\frac{\lvert E[m \mid A]-E[m \mid B]\rvert}{m_{max}}$	**Metric Name**	$\frac{E[m \mid A]}{m_{max}}$	$\frac{E[m \mid B]}{m_{max}}$
0.453	cpu.ready.summation	0.291	0.744
0.443	cpu.latency.average	0.247	0.690
0.386	rescpu.actpk1.latest	0.324	0.711
0.375	rescpu.runav1.latest	0.335	0.710
0.372	cpu.usagemhz.average	0.312	0.684
0.372	cpu.usage.average	0.312	0.684
0.366	rescpu.actpk5.latest	0.409	0.775
0.360	rescpu.actav1.latest	0.196	0.556
0.356	cpu.demand.average	0.201	0.558
0.311	rescpu.actav5.latest	0.257	0.568

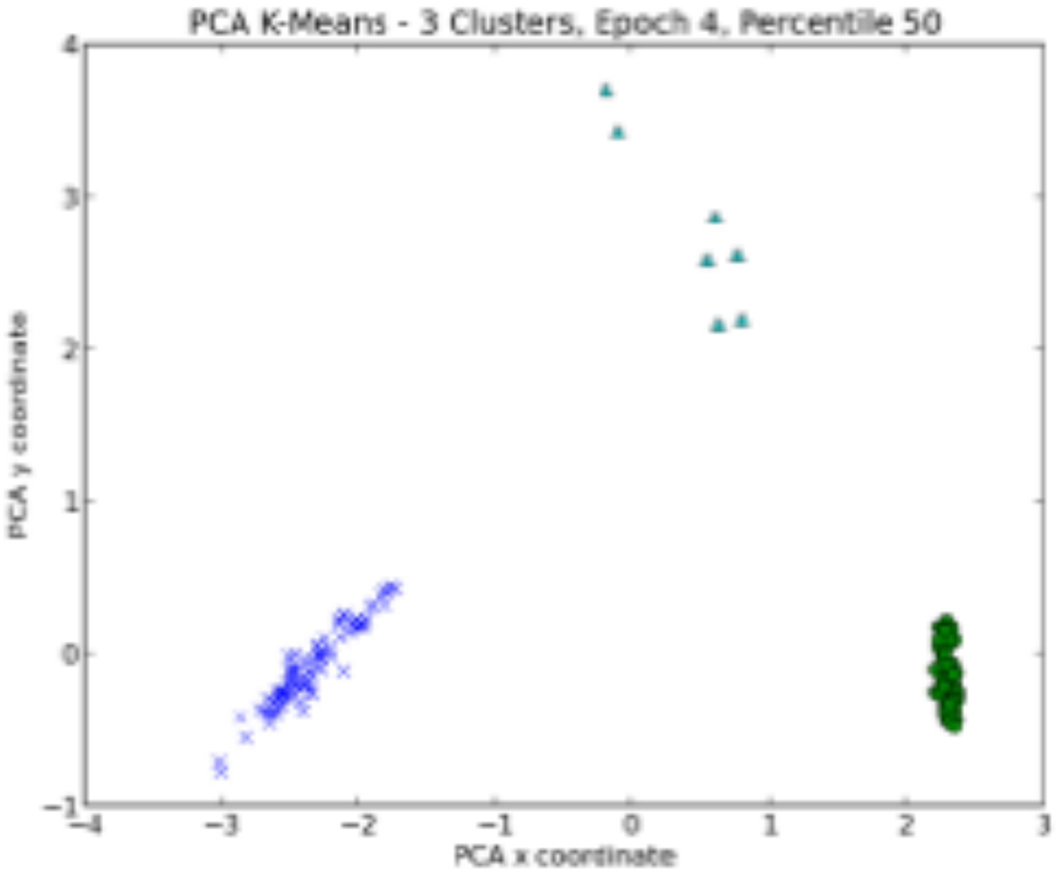
Pipeline D – Conditional Probability Distributions from Multi-modal data

- Removing Spread Metrics S(m) with deltas > theta (0.2) cause clusters to collapse (explaining the original diffusion)

$$S(m) = \frac{|E[m|\text{red cluster}] - E[m|\text{blue cluster}]|}{m_{max}} > \theta$$



PCA K-Means - 4 Clusters, Epoch 4, Percentile 50



PCA K-Means - 3 Clusters, Epoch 4, Percentile 50

Conclusion and Future Work

✓We are able to automatically identify similar VMs based on workloads/telemetry

✓We use classification techniques to fingerprint each group of similar VMs

✓ Within a group of similar VMs we have heuristics for finding potentially interesting metrics to build conditional probability models on to explain diffusion or split clusters

✓All of our techniques build on statistical or signal processing algorithms to create our eventual pipline

Summary

- Virtualized datacenters present the opportunity for flexible & efficient application performance management

- Appropriate resource scheduling, automatic scaling to maintain SLOs w/o over-provisioning, & relevant telemetry data are key to achieving flexibility & efficiency.

Backup

(Some) Related Work

- **Fingerprinting the Datacenter**: Automated Classification of Performance Crisis (*Peter Bodík et al., EuroSys '10*)
- **Using Correlated Surprise to Infer Shared Influence** (*Adam Oliner et al., DSN '10*)
- **Online detection of Multi-Component Interactions in Production Systems** (*Adam Oliner et al., DSN '11*)
- **Dominant Resource Fairness: Fair Allocation of Multiple Resource Types** (*Ali Ghodsi et al., NSDI '11*)
- **Carat: Collaborative Detection of Energy Bugs** (*Adam Oliner et al., carat.cs.berkeley.edu, SenSys '13*)

(Some) Related Work

- **VM and Workload Fingerprinting for Software Defined Datacenters** (*Dragos Ionescu, Masters Thesis MIT, 2012*)