Vertically Autoscaling with Cassandra

Karla Saur - Principal Research SDE
Microsoft - Gray Systems Lab (GSL)

With thanks to:
Anna Pavlenko (VASIM creator), GSL Team, German Eichberger

About me

I am a Principal Research SDE in Gray Systems Lab (GSL) at Microsoft, an applied research lab within Azure Data.

Before Microsoft, I completed my PhD ~10 years ago, and worked as researcher on Telco/5G autoscaling

My main research focus is *optimizing cloud infrastructure for database and machine learning workloads* from a general perspective.

I hope that some of the techniques I mention are useful to you, and I am excited to learn more about Cassandra while at this conference!

About me

I am a Principal Research SDE in Gray Systems Lab (GSL) at Microsoft, an applied research lab within Azure Data.

Before Microsoft, I completed my PhD ~10 years ago, and worked as researcher on Telco/5G autoscaling

My main research focus is *optimizing cloud infrastructure for database and machine learning workloads* from a general perspective.

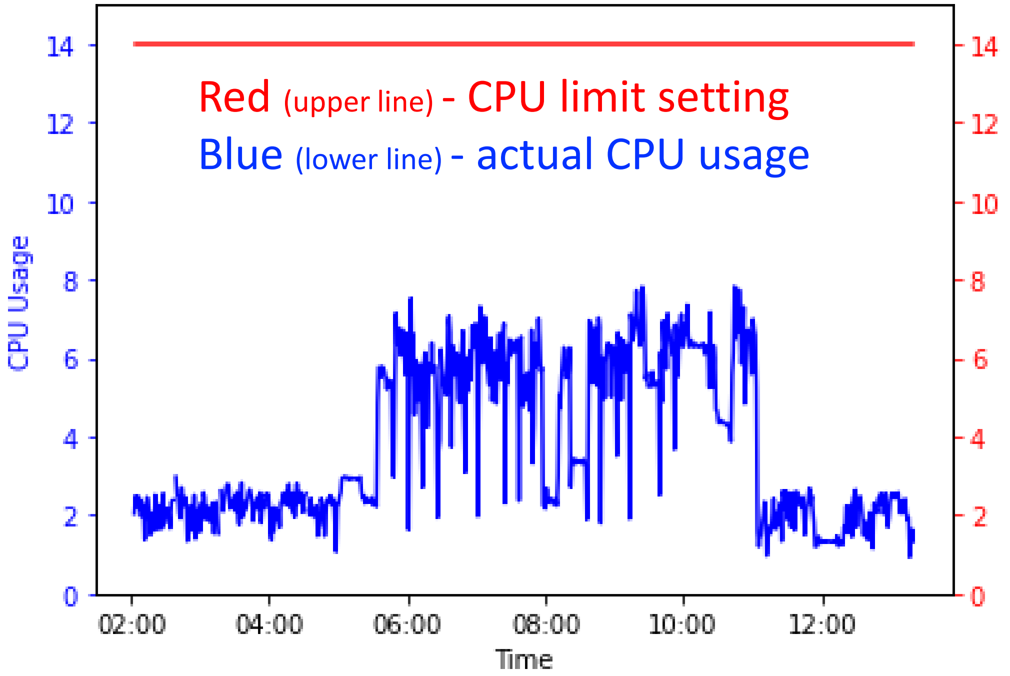
I hope that some of the techniques I mention are useful to you, and I am excited to learn more about Cassandra while at this conference!

Initial Project: Monolithic DB's on K8s

- Originally, our team was tasked with **optimizing deployments of monolithic databases** (ex: 1 primary, 2 secondaries, fixed) running on Kubernetes.

- We found that **many users were overprovisioned** in terms of CPU allocation, which is how we bill (#CPUs/hour)

- This began our **vertical scaling journey**.

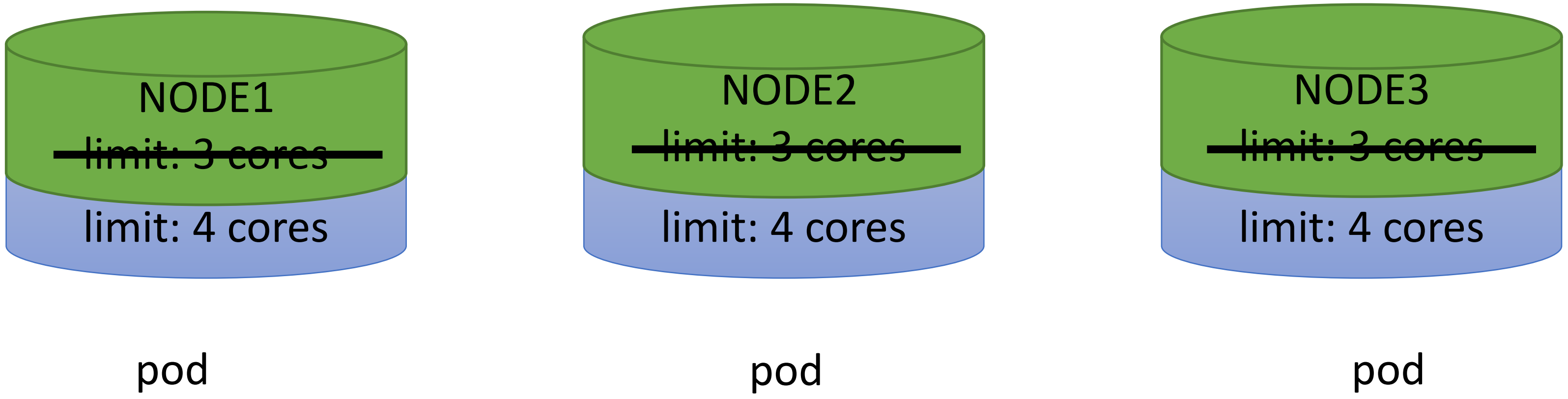


Vertically autoscaling with Cassandra

- Cassandra is famous for its linear scaling, seamlessly adds more nodes.
  - However, bringing up a new node (horizontal scaling) involves data movement and can take a significant amount of time
- Vertical scaling does not involve data movement.
  - It can provide an additional mechanism to right-size resources.
- There are many ways to run Cassandra (on VMs, on containers, on Kubernetes, K8ssandra, etc), and techniques are generally applicable.
  - We can often scale the cores in-place without restart

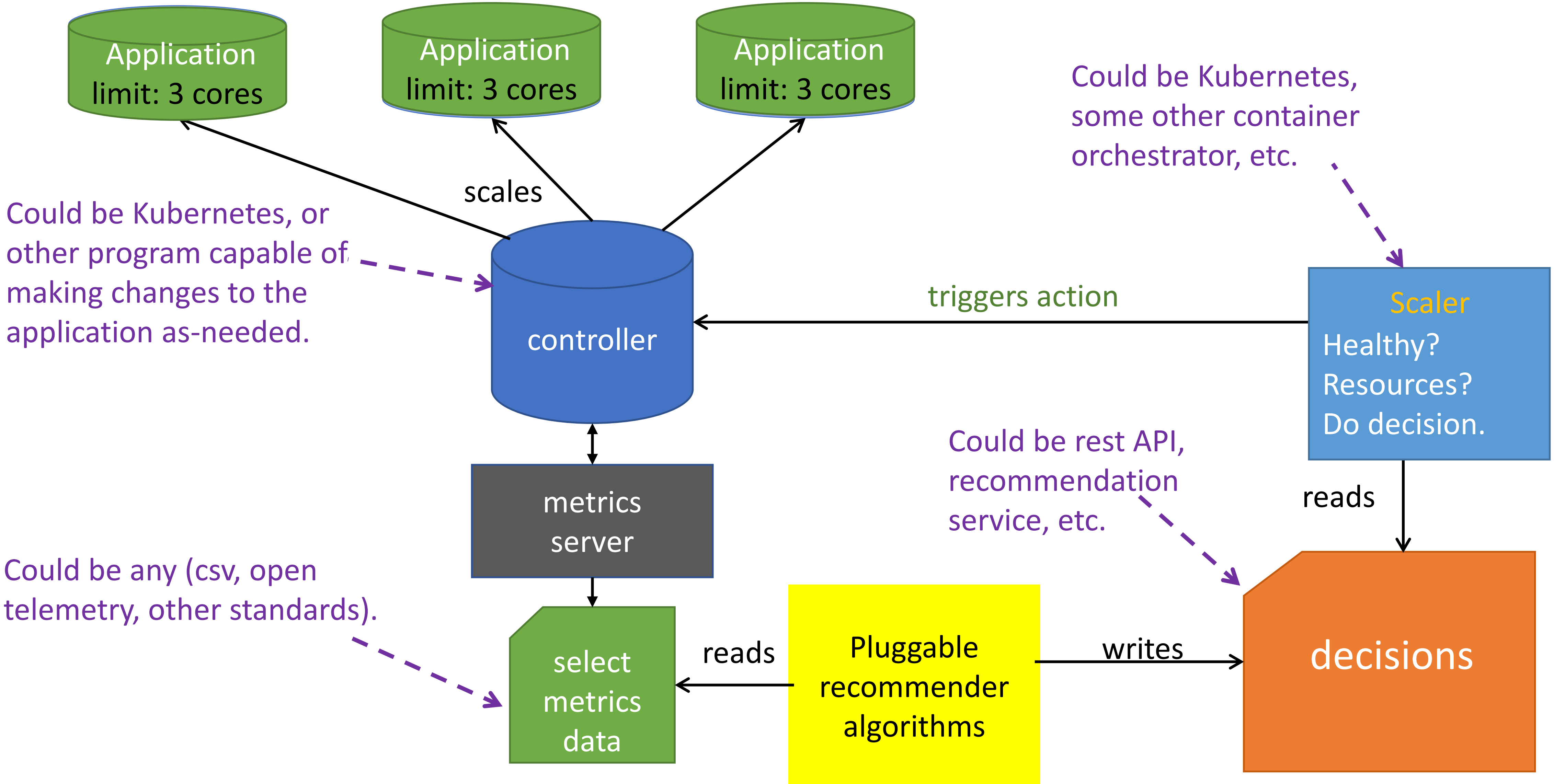
Scenario: Vertically Scaling Generic DBs

- Example: a database runs as a set of Kubernetes Pods with n cores each.
  - Users are billed based on a max CPU limit they specify



- Kubernetes excels at HORIZONTAL pod autoscaling, but our database use case is a fixed number of replicas. But we can scale VERTICALLY!

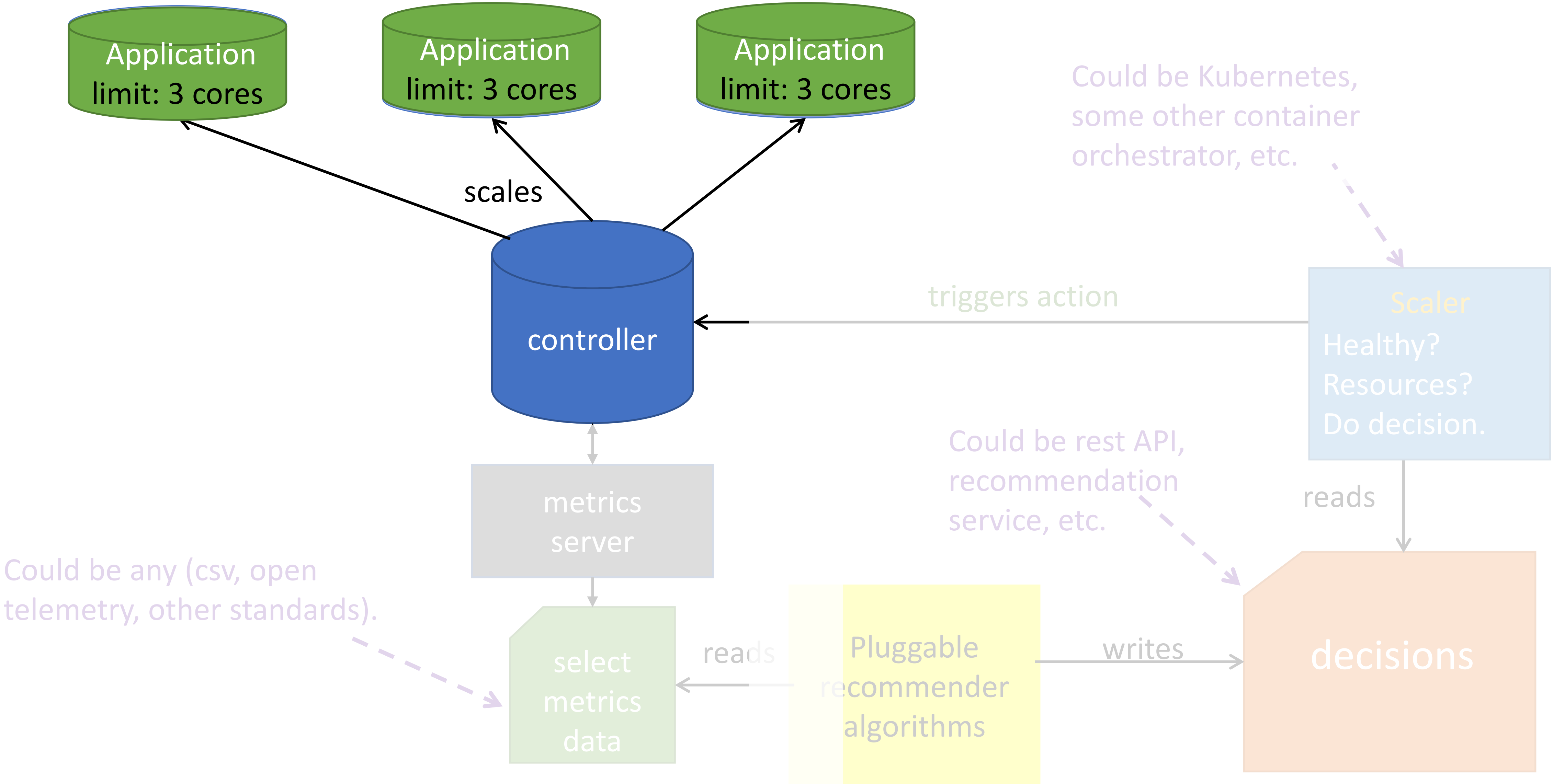
Generic Vertical Autoscaling (end-to-end)



Outline

- Mechanism
  - Vertically scaling in-place
  - Cassandra perf impacts?
- Policy
  - CaaSPER: Proactive/Reactive algorithm for balancing price-perf trade-off
- Code/demo: VASIM - Vertical Autoscaling SIMulator
  - Try your own autoscaling algorithm!
  - Autotuning: parameter tune your own algorithm
  - How to get started with Cassandra

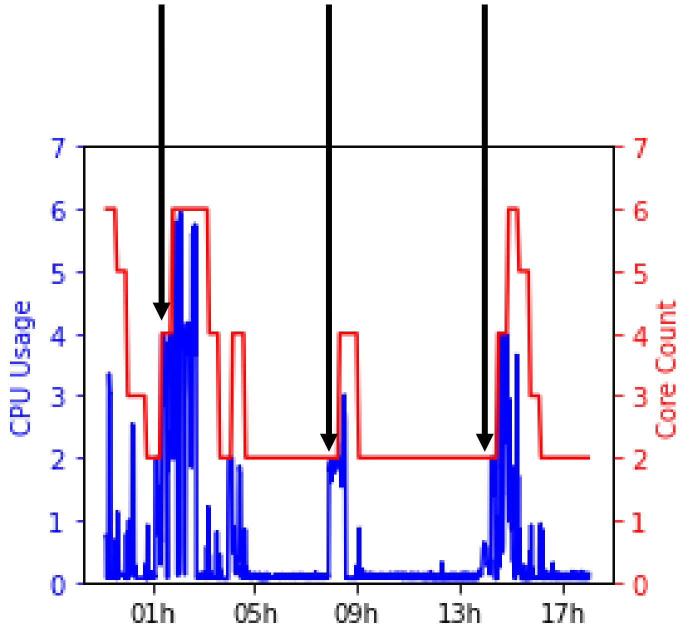
Mechanism: scaling in-place

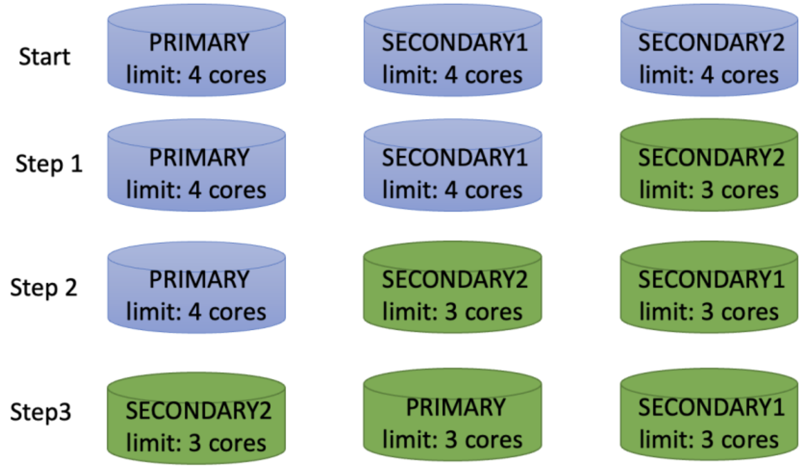


Restarts hinder scaling nimbly in stateful workloads

(even with containers!)

Rolling restart HA process (~10-15 min) makes scaling perf much worse than necessary due to delay.





Rolling upgrade process

With no restarts, we could minimize throttling and optimize scaling further and more safely by reacting faster

No restart!

In-place/no-restart scaling of CPUs

- Docker: `docker update some-cassandra --cpus` or `--cpu-quota`

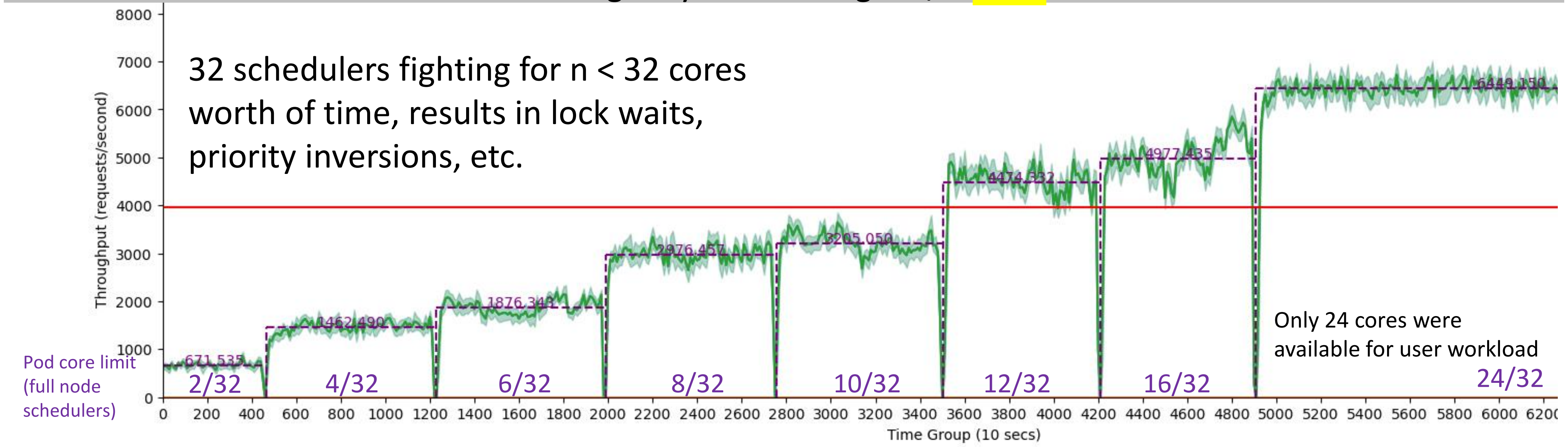
- K8s: In-place updates officially an alpha feature in the 1.27 release (~April 2023) under the feature gate `InPlacePodVerticalScaling`
  - Changed simply by patching the running pod spec
  - Default behavior is "in-place" unless `resizePolicy` is set to `RestartContainer`

```
resizePolicy:
- resourceName: memory
  restartPolicy: RestartContainer
- resourceName: cpu
  restartPolicy: NotRequired
```

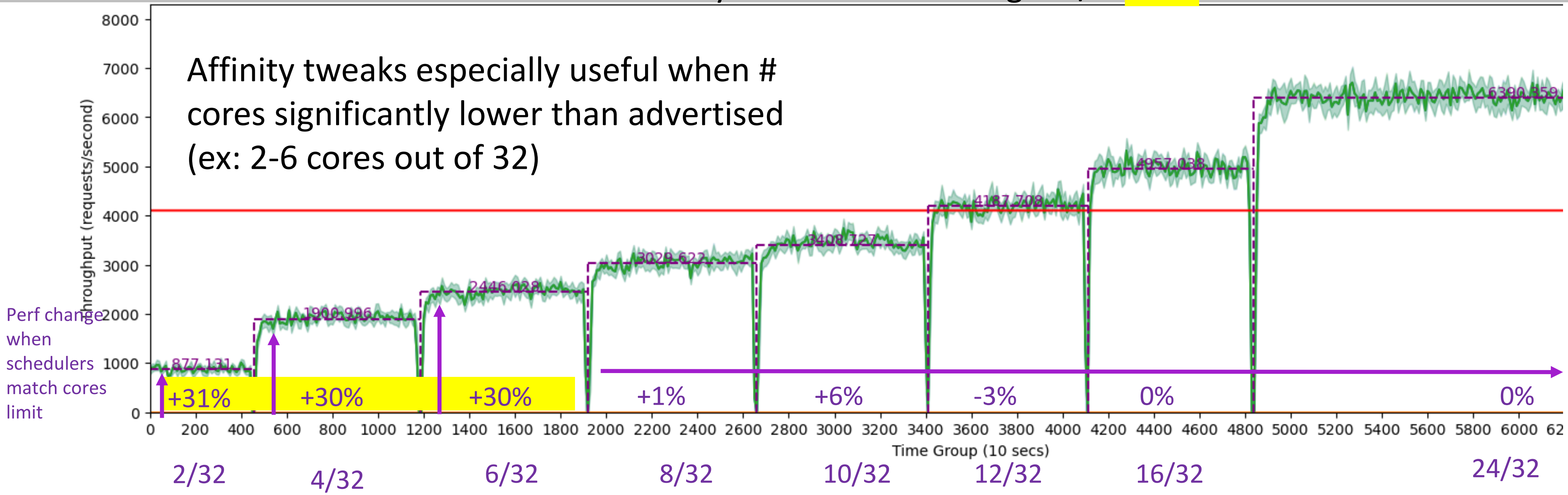
What is the perf impact if *the DB thinks it has n cores, but we actually give it m cores?*

- 3 replica SQL (**on Linux**) deployed on *32 core* K8s nodes
- Start scaling op every 200 seconds
  - Because the machine has 32 cores, SQL Server thinks it has 32 cores
  - Scale from 2 → 4 → 6 → 8 → 10 → 12 → 16 → 24
  - Plot the average throughput during each segment
- 2 tests
  - Scaling only (i.e. SQL activates 32 SqlOS schedulers)
  - Coordination via affinity changes prior to scale

Scaling only. Overall avg txn/s: **3960**

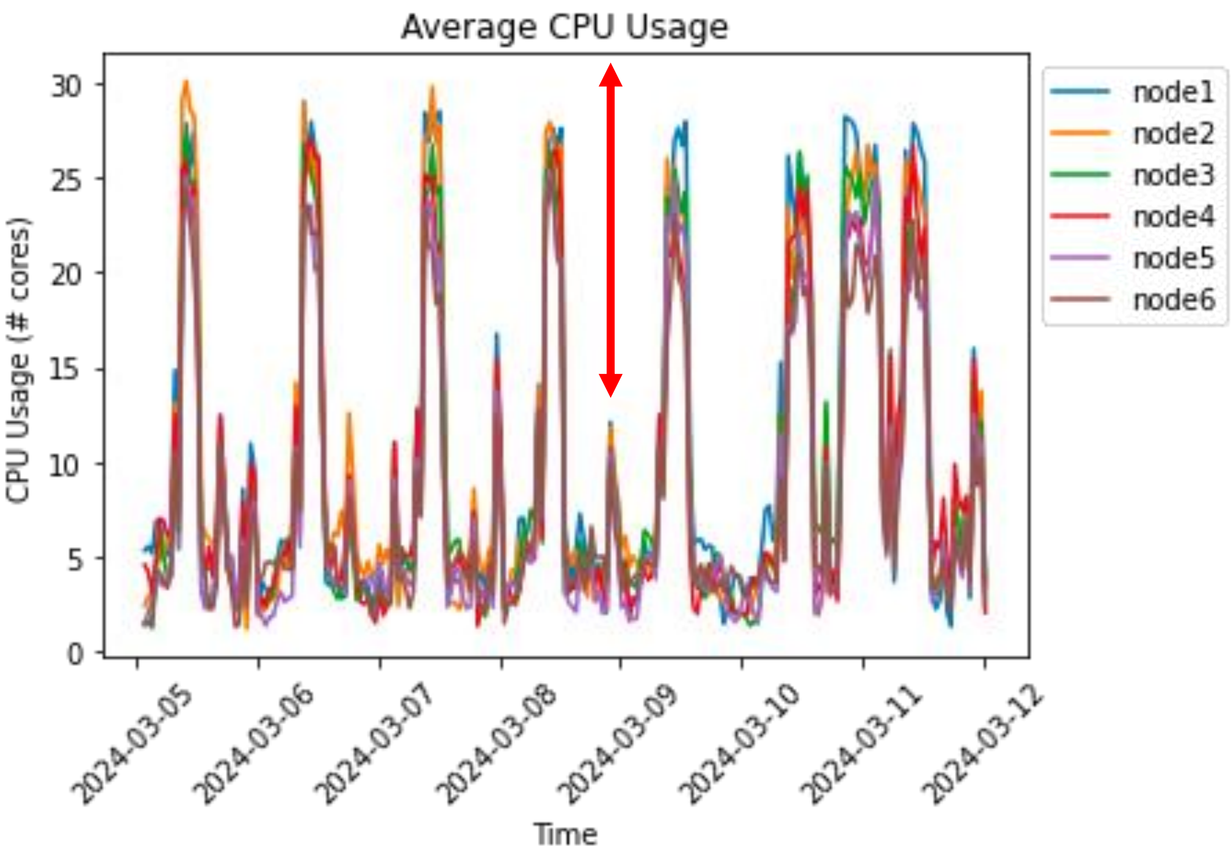


Coordination via affinity tweaks. Overall avg txn/s: **4102**

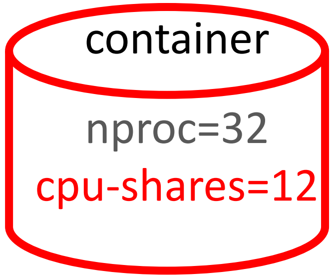


IMPORTANT: Many SQL instances are often idle and could be scaled down to 2-4 cores. The longer we can keep them there, the longer we have cores to use in other places. But we don't want to lose perf!

Perf impact of restart-free core scaling with Cassandra?



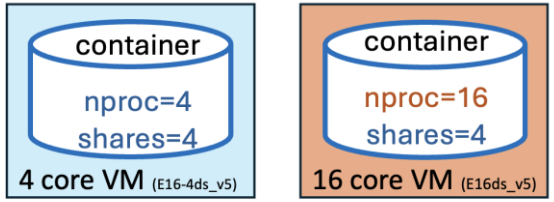
- For this customer provisioned at 32 cores, we could scale down by ~20 cores each night
  - But what is impact to the locks/buffers/threads/etc if the DB thinks it has 32 cores but only has 12?



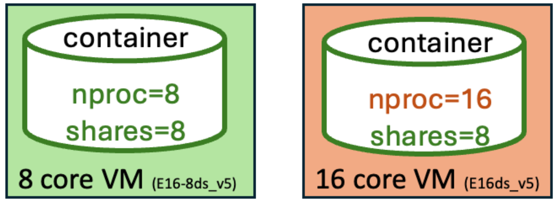
Testing perf impact of restart-free core scaling

Quick experiment with Cassandra:

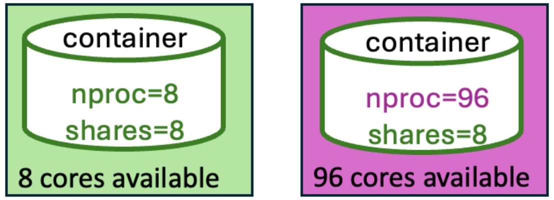
- Ran tried matched/mismatched on a customer's workload: *pleasantly boring.*



calls/second diff:
in the noise/nearly identical



calls/second diff:
in the noise/nearly identical



calls/second diff:
in the noise/nearly identical

- Java's `public int availableProcessors()` - apps must poll this explicitly
  - Cassandra doesn't, so we expected to see some mismatch due to threadpools, etc.
  - However: to use the new cores, Cassandra needs to be aware of the max cores at startup.
- Some JVM-weirdness related to List of Processors…*to be continued!*

What about memory??

- Resizing memory without a restart is challenging, regardless of platform or environment (Python/C/JVM/etc).
- Memory resizing likely requires application changes.
- For now, restart/rolling-update when memory needs to be resized.

Outline

- Mechanism
  - Vertically scaling in-place
  - Cassandra perf impacts?
- Policy
  - CaaSPER: Proactive/Reactive algorithm for balancing price-perf trade-off

Vertical autoscaling: CaaSPER Algorithm

Vertically Autoscaling Monolithic Applications with CaaSPER:

Scalable Container-as-a-Service Performance Enhanced Resizing Algorithm for the Cloud

Anna Pavlenko, Joyce Cahoon, Yiwen Zhu, Brian Kroth, Michael Nelson,
Andrew Carter, David Liao, Travis Wright, Jesús Camacho-Rodríguez, Karla Saur
{firstname}.{lastname}@microsoft.com
Microsoft

ABSTRACT

Kubernetes has emerged as a prominent open-source platform for managing cloud applications, including stateful databases. These monolithic applications rely on vertical scaling, adjusting CPU cores based on load fluctuations. However, our analysis of Kubernetes-based Database-as-a-Service (DBaaS) offerings at Microsoft revealed that many customers consistently over-provision resources for peak workloads, neglecting cost-saving opportunities through resource scale-down. We found that there is a gap in the ability of existing vertical autoscaling tools to minimize resource slack and respond promptly to throttling, leading to increased costs and impacting crucial metrics such as throughput and availability.

To address this challenge, we propose CaaSPER, a vertical autoscaling algorithm that blends reactive and proactive strategies. By dynamically adjusting CPU resources, CaaSPER minimizes resource slack, maintains optimal CPU utilization, and reduces throttling. Importantly, customers have the flexibility to prioritize either cost savings or high performance based on their preferences. Extensive testing demonstrates that CaaSPER effectively reduces throttling and keeps CPU utilization within target levels. CaaSPER is designed to be application-agnostic and platform-agnostic, with potential for extension to other applications requiring vertical autoscaling.

1 INTRODUCTION

Cloud computing [6, 27, 53] has transformed the landscape of application development, deployment, and management by providing organizations with access to on-demand resources and scalability. However, during provisioning, users are often required to specify the amount of resources they will initially require among a large number of cloud offerings (e.g., VM configuration and size). It is challenging to estimate resource requirements upfront, and the initial settings can become irrelevant with the dynamic nature of the workloads. One of the most common scaling approaches to address some of these issues has been *horizontal autoscaling*, in which additional service replicas are added and removed based on utilization, thus adjusting overall system resource usage in fixed-sized quantities. Although this has worked for some services [59], this approach is not well suited for stateful monolithic systems (e.g., traditional RDBMS) that either have a fixed number of total instances (e.g., single writable primary) or cannot quickly scale horizontally due to size of data copy operations inherent to creating new replicas. In such cases, *vertical scaling* capabilities become crucial, allowing expanding or contracting resources of existing replicas. Additional benefits to vertical scaling include simplicity, performance, and reliability [41].

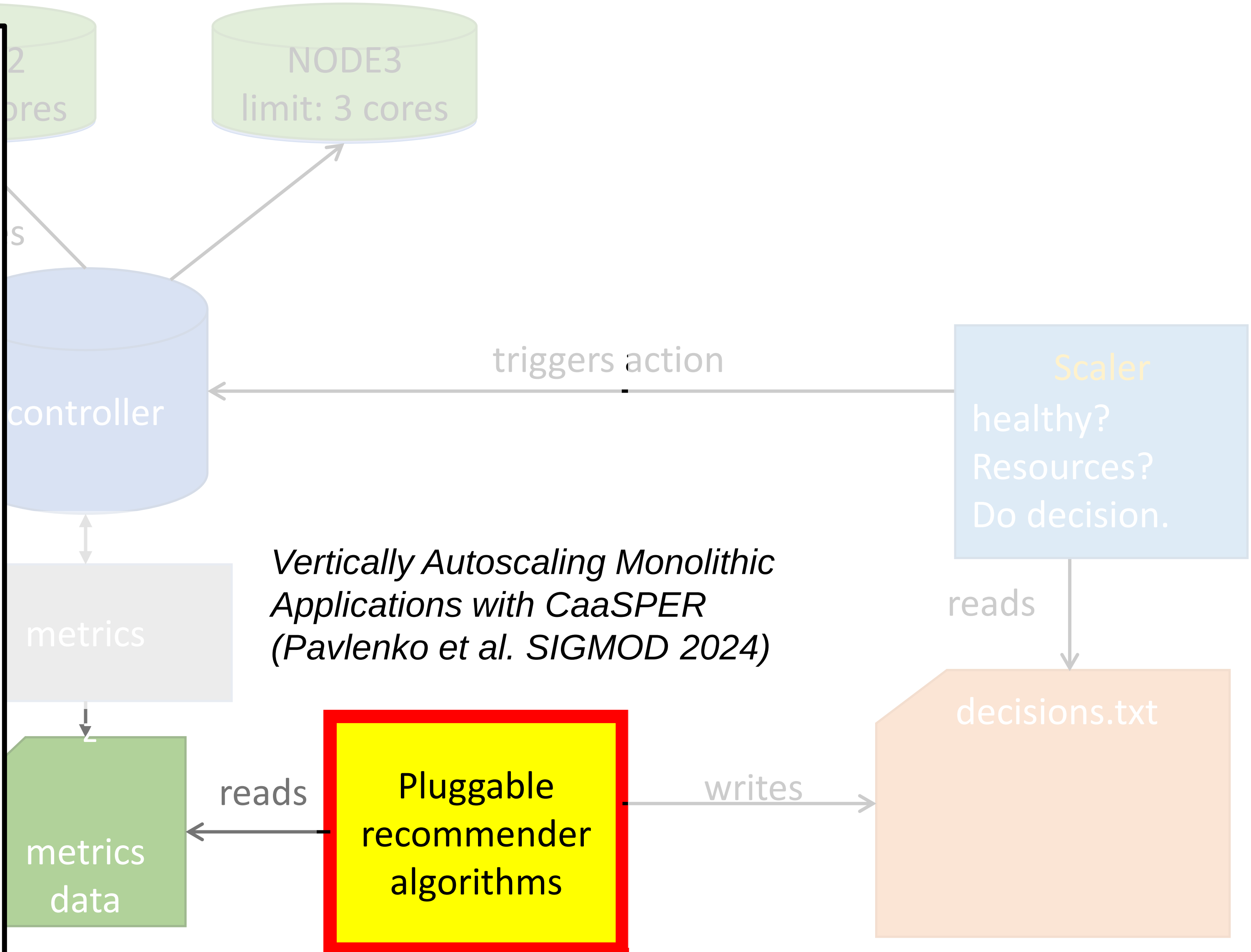
Modern platforms such as Kubernetes (K8s) [45], which has become a popular platform choice for implementing Database-as-a-Service (DBaaS) and other stateful service offerings [20, 39, 54, 66], facilitate vertical resource scaling. K8s provides two essential mechanisms, `requests` and `limits`, to define *guaranteed* and *burstable* CPU resource allocation for applications. At Microsoft and elsewhere, applications needing predictable performance, such as databases [24, 37], often set `requests` and `limits` to the same value [13, 40, 48] to ensure that the application will be scheduled on a node with enough resources to always provide the `limits`.[1] Despite resizing support, we find that users in our database offerings at Microsoft rarely scale their deployments and usually over-provision for the worst-case. In fact, during a sampling of first-party DBaaS deployments, we found cases of CPU *over-provisioning* that exceeded peak load by a significant factor, up to 20x in certain workloads, resulting in under-utilized resources (*i.e.*, increased costs for idle resources), as well as instances of *under-provisioning*, which leads to performance impacts due to "throttling" (*i.e.*, when an application lacks enough resources to meet its load demands).

To automate this process, the Vertical Pod Autoscaler (VPA) [33] in K8s can dynamically adjust the `requests` and `limits` values according to a pluggable algorithm. However, when tested, the default VPA algorithm and other existing approaches proved inadequate in our scenario for effectively addressing cases involving throttling detection and scaling down when over-provisioned. Additionally, these methods were oblivious of the billing model in use, leading to suboptimal cost-performance tradeoffs during scaling decisions, an important consideration for customers. Other recent works in K8s [73] leverage machine learning to support predictive autoscale. However, there is a significant drawback in purely relying on a machine learning algorithm in predictive autoscaling especially using time-series forecasting, as it lacks the ability to effectively detect throttling and instead assumes that future usage will remain consistent until retrained. Moreover, for throttled workloads, the usage forecast does not align with the true amount of resources required for the workload, leading to under-estimated `limits` (see §3.3). When aiming for optimal performance, there is a strong need to quickly identify and respond to throttling to meet SLA objectives.

SIGMOD '24, June 2024, Santiago, Chile

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

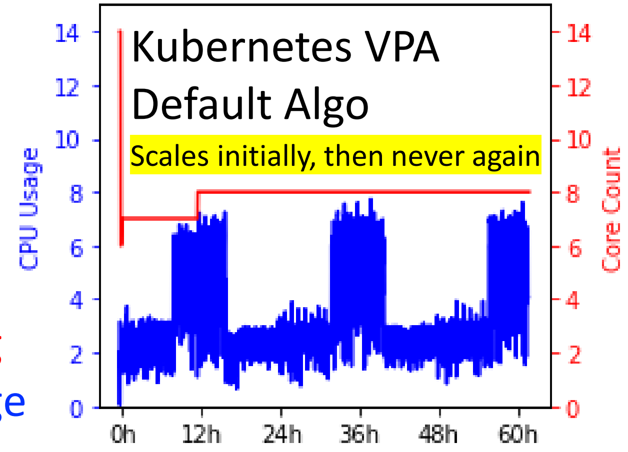
[1] Note that service level agreements (SLAs) provided by DBaaS further emphasize the importance of predictability and may have penalties for violations [57, 58].

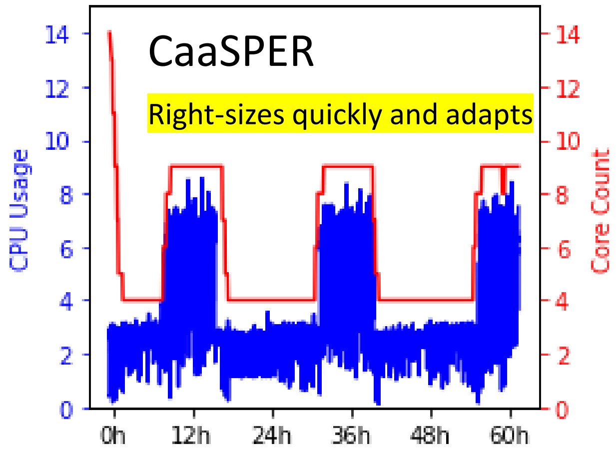


Vertically Autoscaling Monolithic Applications with CaaSPER (Pavlenko et al. SIGMOD 2024)

Why not K8s built-in Vertical Pod Autoscaler (VPA)?

- For billing, we scale only at whole-cores, with `limits=requests`
  - In Kubernetes, `requests` and `limits` define guaranteed and burstable CPU resource allocation for applications. Setting these equal ‘breaks’ VPA.
- Must consider customer preferences when scaling
  - Most existing VPA tools are for optimal scheduling, not cost-perf preferences

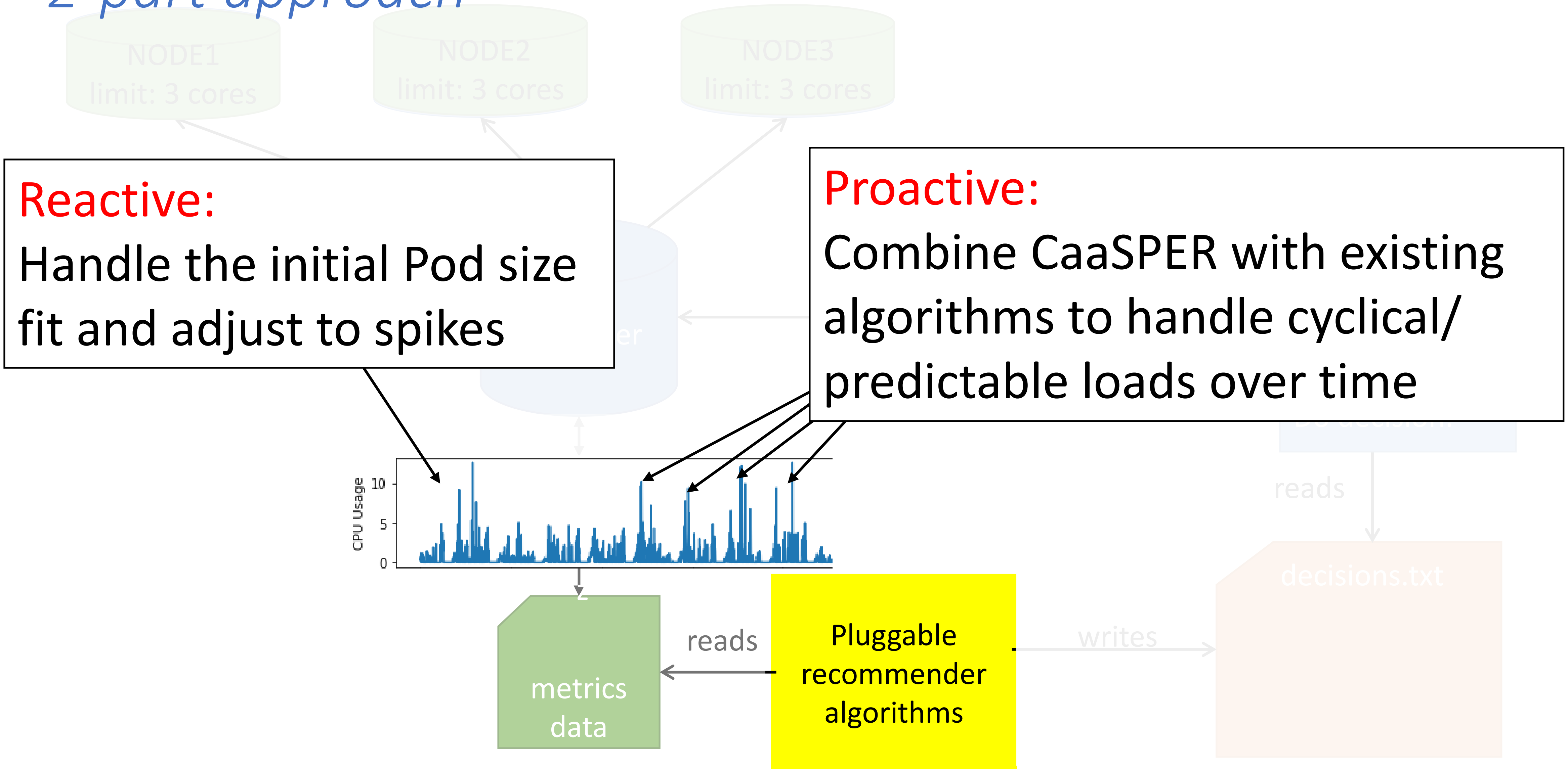




Red - CPU limit setting
Blue - actual CPU usage

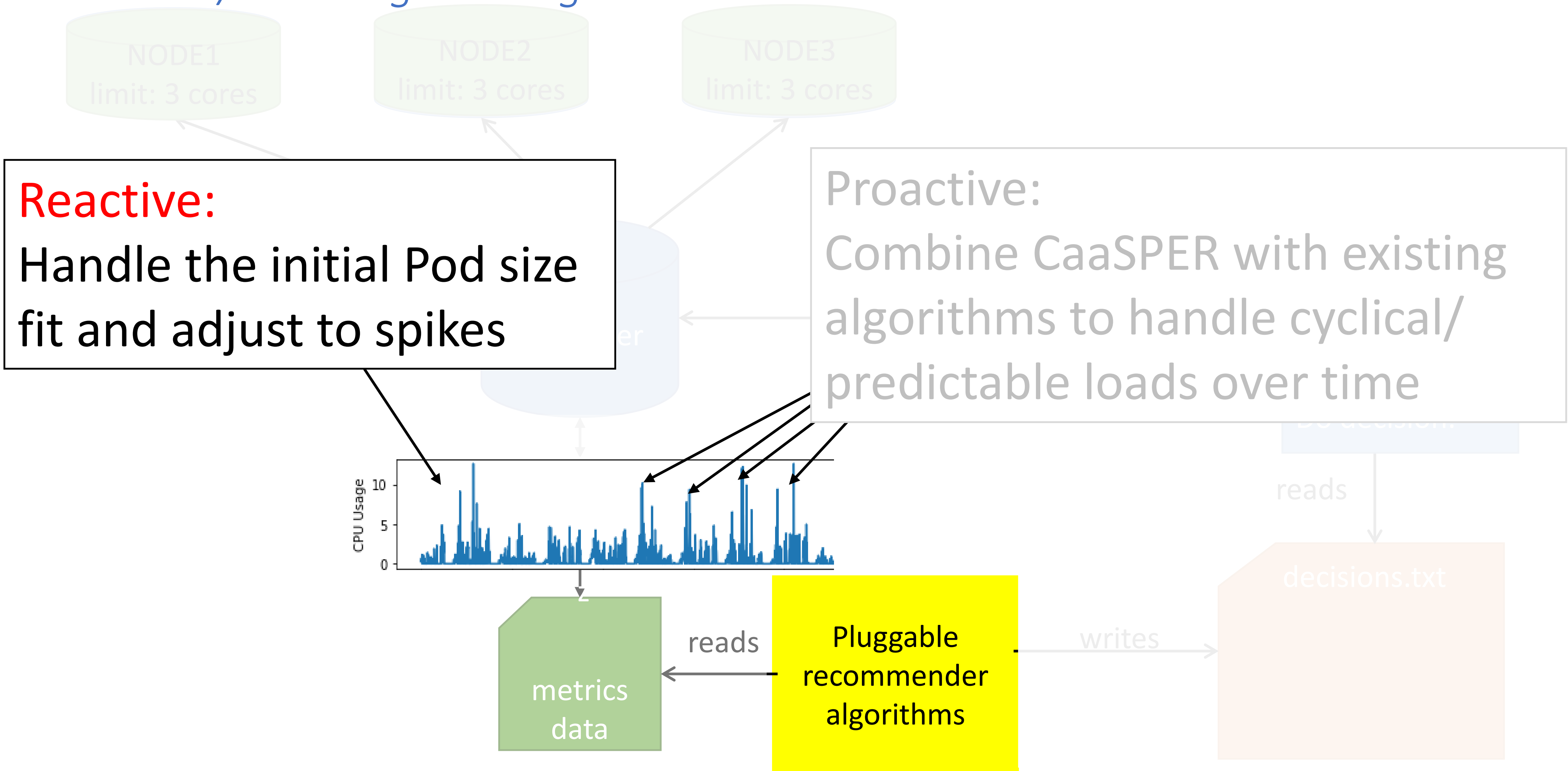
Vertical auto-scaling approach:

2-part approach



Vertical auto-scaling: Part 1

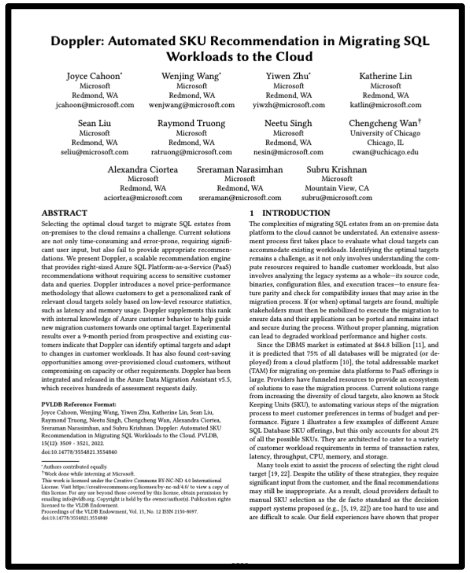
Reactive/initial right-sizing with no data



Reactive CaaSPER

Doppler (prior work) provides initial SKU (#cores/#mem) selection *offline* for SQL Server based on personalized price-perf curve

- We adapted this price-perf curve for our container scenario by monitoring the change in slope over time, instead of focusing on a static price-perf curve for migration



Doppler: Automated SKU Recommendation in Migrating SQL Workloads to the Cloud

Joyce Cahoon*, Microsoft, Redmond, WA, jcahoon@microsoft.com
Wenjing Wang*, Microsoft, Redmond, WA, wenjwang@microsoft.com
Yiwen Zhu*, Microsoft, Redmond, WA, yiwzh@microsoft.com
Katherine Lin, Microsoft, Redmond, WA, katlin@microsoft.com
Sean Liu, Microsoft, Redmond, WA, seliu@microsoft.com
Raymond Truong, Microsoft, Redmond, WA, ratruong@microsoft.com
Neetu Singh, Microsoft, Redmond, WA, nesin@microsoft.com
Chengcheng Wan†, University of Chicago, Chicago, IL, cwan@uchicago.edu
Alexandra Ciortea, Microsoft, Redmond, WA, aciortea@microsoft.com
Sreraman Narasimhan, Microsoft, Redmond, WA, sreraman@microsoft.com
Subru Krishnan, Microsoft, Mountain View, CA, subru@microsoft.com

ABSTRACT

Selecting the optimal cloud target to migrate SQL estates from on-premises to the cloud remains a challenge. Current solutions are not only time-consuming and error-prone, requiring significant user input, but also fail to provide appropriate recommendations. We present Doppler, a scalable recommendation engine that provides right-sized Azure SQL Platform-as-a-Service (PaaS) recommendations without requiring access to sensitive customer data and queries. Doppler introduces a novel price-performance methodology that allows customers to get a personalized rank of relevant cloud targets solely based on low-level resource statistics, such as latency and memory usage. Doppler supplements this rank with internal knowledge of Azure customer behavior to help guide new migration customers towards one optimal target. Experimental results over a 9-month period from prospective and existing customers indicate that Doppler can identify optimal targets and adapt to changes in customer workloads. It has also found cost-saving opportunities among over-provisioned cloud customers, without compromising on capacity or other requirements. Doppler has been integrated and released in the Azure Data Migration Assistant v5.5, which receives hundreds of assessment requests daily.

PVLDB Reference Format:
Joyce Cahoon, Wenjing Wang, Yiwen Zhu, Katherine Lin, Sean Liu, Raymond Truong, Neetu Singh, Chengcheng Wan, Alexandra Ciortea, Sreraman Narasimhan, and Subru Krishnan. Doppler: Automated SKU Recommendation in Migrating SQL Workloads to the Cloud. PVLDB, 15(12): 3509 - 3521, 2022.
doi:10.14778/3554821.3554840

*Authors contributed equally.
†Work done while interning at Microsoft.

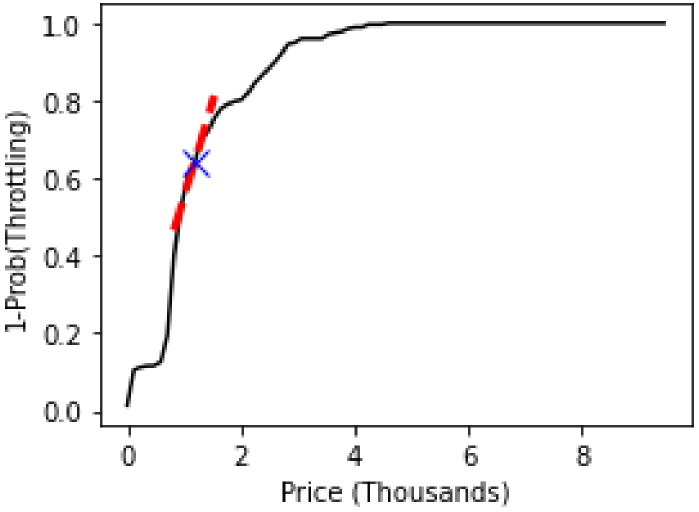
Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554840

1 INTRODUCTION

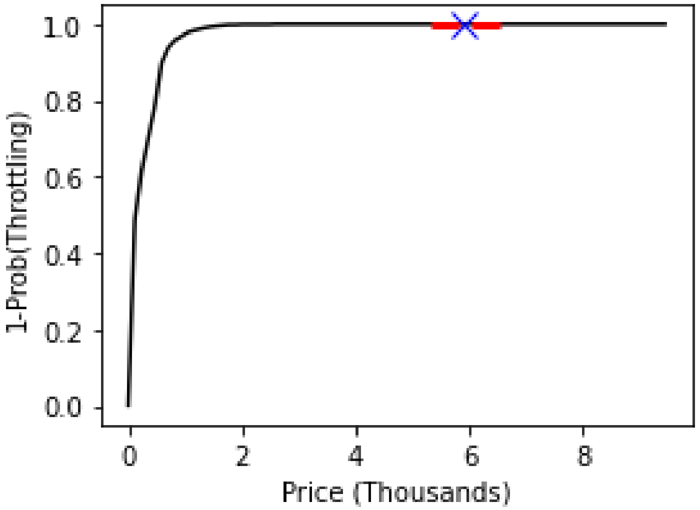
The complexities of migrating SQL estates from an on-premise data platform to the cloud cannot be understated. An extensive assessment process first takes place to evaluate what cloud targets can accommodate existing workloads. Identifying the optimal targets remains a challenge, as it not only involves understanding the compute resources required to handle customer workloads, but also involves analyzing the legacy systems as a whole—its source code, binaries, configuration files, and execution traces—to ensure feature parity and check for compatibility issues that may arise in the migration process. If (or when) optimal targets are found, multiple stakeholders must then be mobilized to execute the migration to ensure data and their applications can be ported and remains intact and secure during the process. Without proper planning, migration can lead to degraded workload performance and higher costs.

Since the DBMS market is estimated at $64.8 billion [11], and it is predicted that 75% of all databases will be migrated (or deployed) from a cloud platform [10], the total addressable market (TAM) for migrating on-premise data platforms to PaaS offerings is large. Providers have funneled resources to provide an ecosystem of solutions to ease the migration process. Current solutions range from increasing the diversity of cloud targets, also known as Stock Keeping Units (SKU), to automating various steps of the migration process to meet customer preferences in terms of budget and performance. Figure 1 illustrates a few examples of different Azure SQL Database SKU offerings, but this only accounts for about 2% of all the possible SKUs. They are architected to cater to a variety of customer workload requirements in terms of transaction rates, latency, throughput, CPU, memory, and storage.

Many tools exist to assist the process of selecting the right cloud target [19, 22]. Despite the utility of these strategies, they require significant input from the customer, and the final recommendations may still be inappropriate. As a result, cloud providers default to manual SKU selection as the de facto standard as the decision support systems proposed (e.g., [5, 19, 22]) are too hard to use and are difficult to scale. Our field experiences have shown that proper



if > threshold, scale up



if < threshold scale down

Doppler: Automated SKU Recommendation in Migrating SQL Workloads to the Cloud. PVLDB 15, 12 (2022).

Reactive CaaSPER

Steepness of price-perf curve determines how MUCH to scale

$$\text{Scaling Factor}(s) = \log(bs + c)$$

- s: Slope of the PP curve at the existing number of cores
- b: Skew estimate of the distribution of existing slopes
- c: Minimum number of cores needed to operate

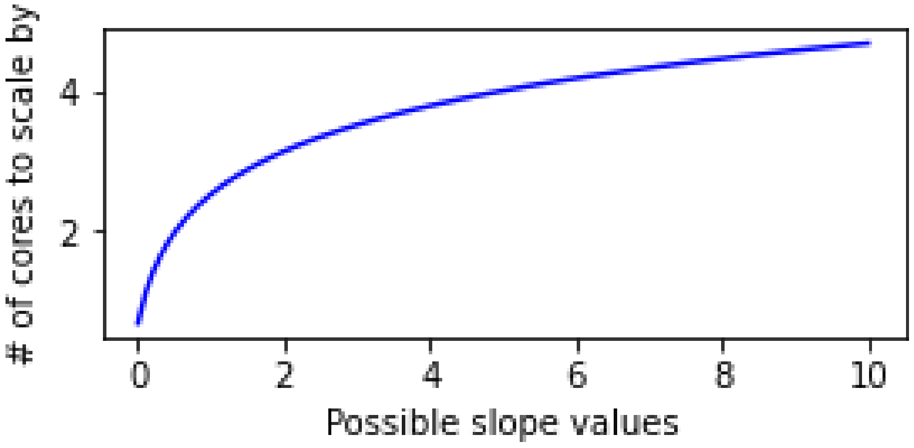


Figure 6: Example shape of scaling-factor function $SF(s)$ of *PvP-curve* slope s. Scale-ups happen more aggressively for large s (more throttling), than small s (less throttling).

Vertical auto-scaling approach:

Pluggable proactive portion

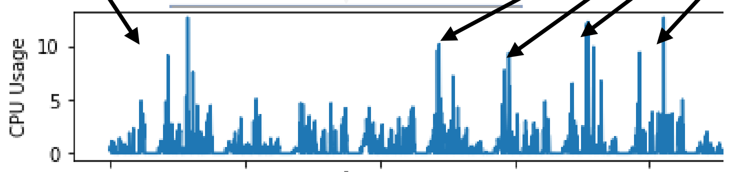
NODE1
limit: 3 cores

NODE2
limit: 3 cores

NODE3
limit: 3 cores

Reactive:
Handle the initial Pod size fit and adjust to spikes

Proactive:
Combine CaaSPER with existing algorithms to handle cyclical/ predictable loads over time



reads

metrics data

reads

Pluggable recommender algorithms

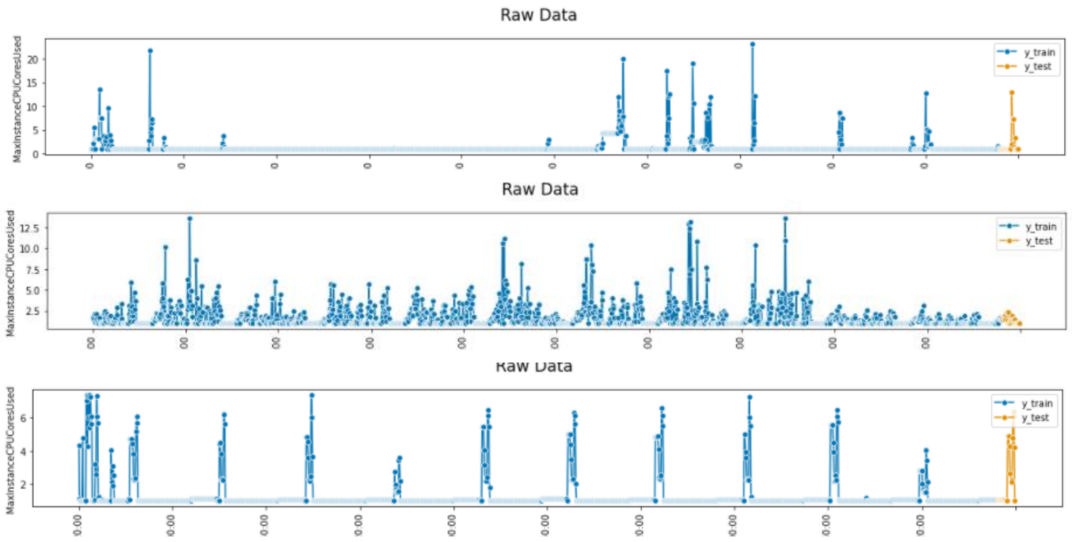
writes

decisions.txt

Proactive:

Real cyclical workload + Time series

- Started by looking at simulated + real workload CPU traces



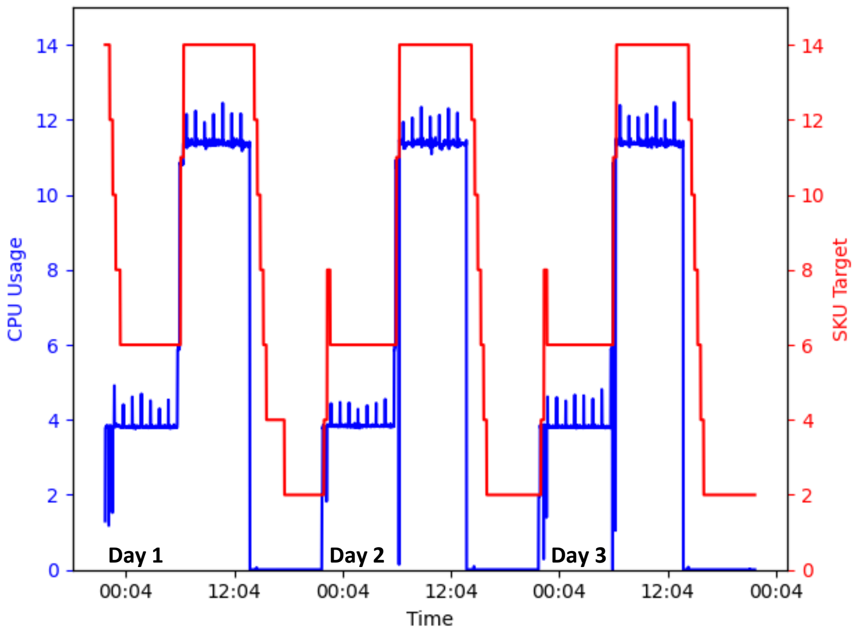
- Experimented with many different algorithms and data preprocessing for prediction and measured throttling/fit/etc
- Naïve worked well for most of our scenarios, but can easily swap out
- Trade-off: complexity/robustness+debuggability

Combining Reactive + Proactive

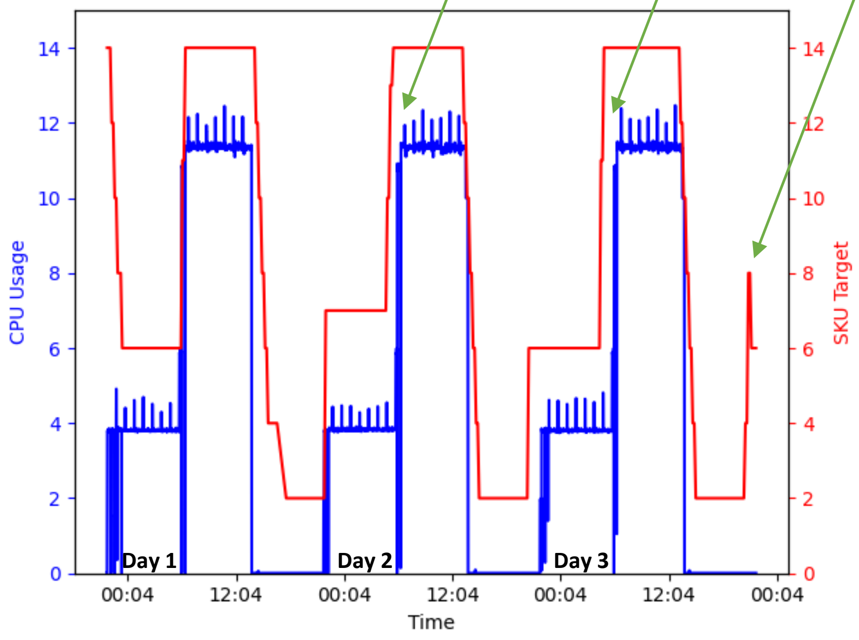
Predictive based on 1 day

Predictive based on 2 days

Predictive scale-up but with load change, reactive CaaSPER takes back over



Reactive only



Reactive + Proactive

Red - CPU limit setting

Blue - actual CPU usage

CaaSPER parameters

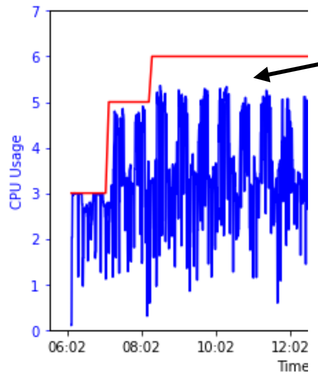
Users can specify preferences on a slider, or we can autotune in our simulator (next):

More performant/More expensive

slider of tradeoffs

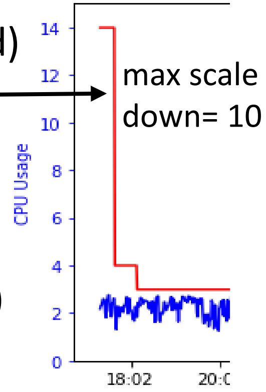
Less performant/More cost-effective

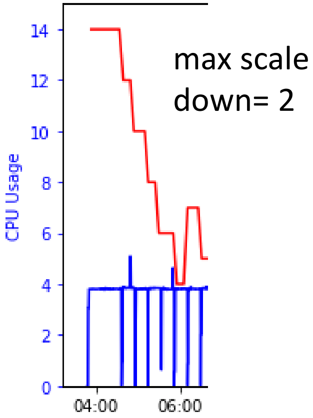
Impact of our parameters:



"slack" parameter is configurable. Here: 15%

- Slack (buffer between resources used and resources allocated)
- Max amount to auto-scale up/down
- How frequently to scale
- How much historical usage data to look at
- Balance between reactive vs proactive algorithm
- How early to be proactive (scale up 5 min early or 1 hour, etc)
- How frequently to scale
- Guardrails (ex: giant burst of traffic, how to behave)





At this point: panicking.

- We had a paper deadline. We built an awesome algorithm, but tuning the 20+ parameters was challenging
- We needed to demonstrate our autoscaling algorithm for about 30 7-day long experiments to run, but we only had 3 functioning K8s clusters, and 10 days.
- Enter: VASIM

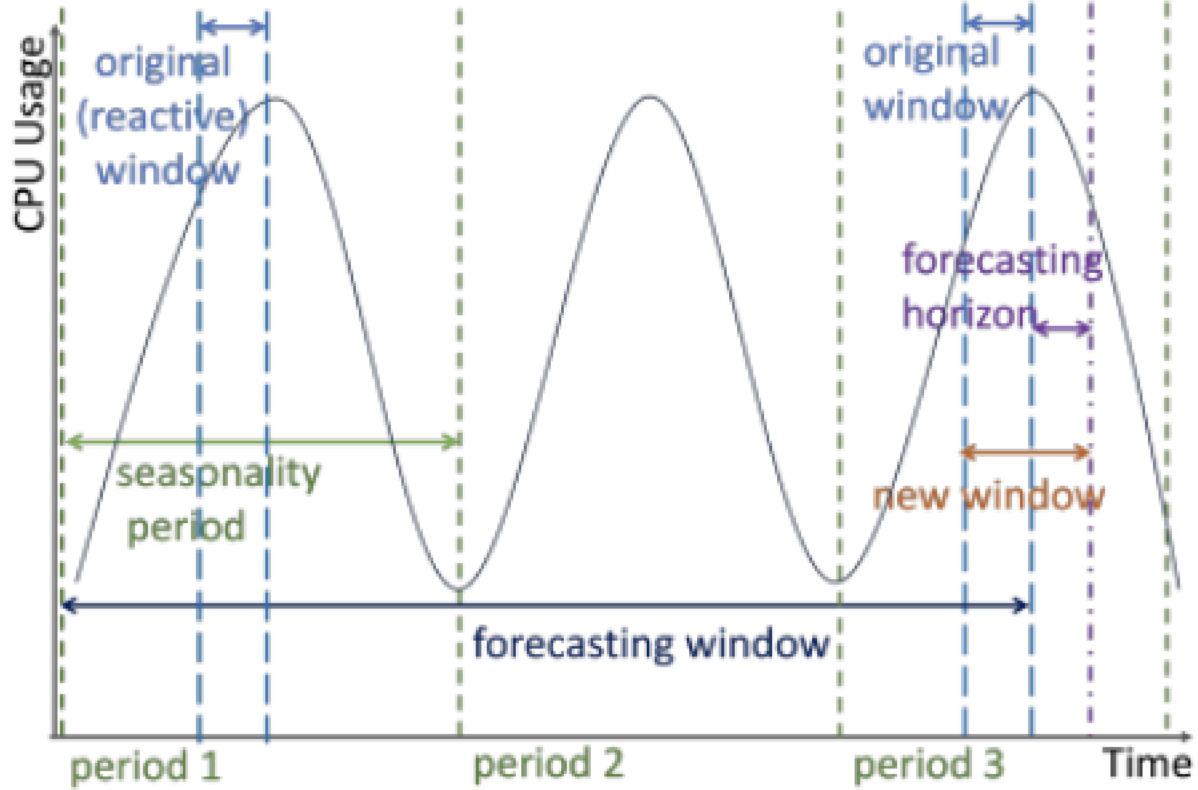
Algorithm 1 CaaSPER autoscaling decision algorithm.

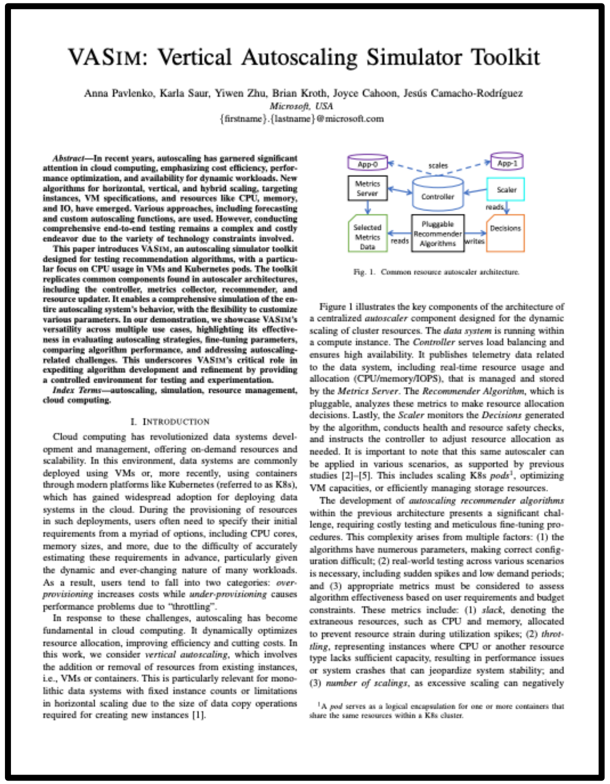
Require: x_c: CoreCount$_{cur}$
Require: $\{X_t\}$: Vector of workload CPU usage indexed by time (observed and/or predicted)
Require: R: System inputs (e.g., resource limit such as max CPU, price per core, granularity per core)
Require: s_h: High slope threshold
Require: s_l: Low slope threshold
Require: m_h: High slack threshold as percentage of capacity
Require: m_l: Low slack threshold as percentage of capacity
Require: SF$_h$: Maximum single step scale-up amount
Require: SF$_l$: Maximum single step scale-down amount
Require: c_{min}: Minimum resource requirements (scale-down lower bound)

```
function AUTOSCALE(x_c, {X_t})
    normalized cpu ← PREPROCESS CPU({X_t})
    PvP curve ← SKU RECOMMENDATION TOOL(normalized CPU, R)
    PvP slopes ← CALCULATE SLOPES(PvP curve)
    skew ← CALCULATE SKEW(PvP slopes)
    s ← GET CURRENT SLOPE(PvP slopes, x_c)
    SF ← CALCULATE SCALING FACTOR, SF(s, skew)
    if s ≥ s_h or Quantile({X_t}) ≥ (1 − m_h) * x_c then
        SF ← min (SF, SF_h)
    else if s ≤ s_l or Quantile({X_t}) ≤ m_l * x_c then
        SF ← max (−SF, −SF_l)
    else if s == 0 and x_c at top of PvP curve then
        SF ← UPDATE SCALING FACTOR(PvP curve, x_c)
    SF ← APPLY GUARDRAILS(SF, SF_h, SF_l, c_min, R)
    return SF
```

VASIM: Vertical Autoscaling Simulator

VASIM replicates common components found in autoscaler architectures and replays CPU traces (real and estimated) with tunable parameters





VASIM: Vertical Autoscaling Simulator Toolkit

Anna Pavlenko, Karla Saur, Yiwen Zhu, Brian Kroth, Joyce Cahoon, Jesús Camacho-Rodríguez
Microsoft, USA
{firstname}.{lastname}@microsoft.com

***Abstract*—In recent years, autoscaling has garnered significant attention in cloud computing, emphasizing cost efficiency, performance optimization, and availability for dynamic workloads. New algorithms for horizontal, vertical, and hybrid scaling, targeting instances, VM specifications, and resources like CPU, memory, and IO, have emerged. Various approaches, including forecasting and custom autoscaling functions, are used. However, conducting comprehensive end-to-end testing remains a complex and costly endeavor due to the variety of technology constraints involved.**

This paper introduces VASIM, an autoscaling simulator toolkit designed for testing recommendation algorithms, with a particular focus on CPU usage in VMs and Kubernetes pods. The toolkit replicates common components found in autoscaler architectures, including the controller, metrics collector, recommender, and resource updater. It enables a comprehensive simulation of the entire autoscaling system's behavior, with the flexibility to customize various parameters. In our demonstration, we showcase VASIM's versatility across multiple use cases, highlighting its effectiveness in evaluating autoscaling strategies, fine-tuning parameters, comparing algorithm performance, and addressing autoscaling-related challenges. This underscores VASIM's critical role in expediting algorithm development and refinement by providing a controlled environment for testing and experimentation.

***Index Terms*—autoscaling, simulation, resource management, cloud computing.**

I. INTRODUCTION

Cloud computing has revolutionized data systems development and management, offering on-demand resources and scalability. In this environment, data systems are commonly deployed using VMs or, more recently, using containers through modern platforms like Kubernetes (referred to as K8s), which has gained widespread adoption for deploying data systems in the cloud. During the provisioning of resources in such deployments, users often need to specify their initial requirements from a myriad of options, including CPU cores, memory sizes, and more, due to the difficulty of accurately estimating these requirements in advance, particularly given the dynamic and ever-changing nature of many workloads. As a result, users tend to fall into two categories: *over-provisioning* increases costs while *under-provisioning* causes performance problems due to "throttling".

In response to these challenges, autoscaling has become fundamental in cloud computing. It dynamically optimizes resource allocation, improving efficiency and cutting costs. In this work, we consider *vertical autoscaling*, which involves the addition or removal of resources from existing instances, i.e., VMs or containers. This is particularly relevant for monolithic data systems with fixed instance counts or limitations in horizontal scaling due to the size of data copy operations required for creating new instances [1].

Fig. 1. Common resource autoscaler architecture.

Figure 1 illustrates the key components of the architecture of a centralized *autoscaler* component designed for the dynamic scaling of cluster resources. The *data system* is running within a compute instance. The *Controller* serves load balancing and ensures high availability. It publishes telemetry data related to the data system, including real-time resource usage and allocation (CPU/memory/IOPS), that is managed and stored by the *Metrics Server*. The *Recommender Algorithm*, which is pluggable, analyzes these metrics to make resource allocation decisions. Lastly, the *Scaler* monitors the *Decisions* generated by the algorithm, conducts health and resource safety checks, and instructs the controller to adjust resource allocation as needed. It is important to note that this same autoscaler can be applied in various scenarios, as supported by previous studies [2]–[5]. This includes scaling K8s *pods*[1], optimizing VM capacities, or efficiently managing storage resources.

The development of *autoscaling recommender algorithms* within the previous architecture presents a significant challenge, requiring costly testing and meticulous fine-tuning procedures. This complexity arises from multiple factors: (1) the algorithms have numerous parameters, making correct configuration difficult; (2) real-world testing across various scenarios is necessary, including sudden spikes and low demand periods; and (3) appropriate metrics must be considered to assess algorithm effectiveness based on user requirements and budget constraints. These metrics include: (1) *slack*, denoting the extraneous resources, such as CPU and memory, allocated to prevent resource strain during utilization spikes; (2) *throttling*, representing instances where CPU or another resource type lacks sufficient capacity, resulting in performance issues or system crashes that can jeopardize system stability; and (3) *number of scalings*, as excessive scaling can negatively

[1] A *pod* serves as a logical encapsulation for one or more containers that share the same resources within a K8s cluster.

VASIM: Vertical Autoscaling Simulator Toolkit.
In IEEE International Conference on Data Engineering (ICDE 2024)

VASIM: Vertical Autoscaling Simulator

You need 3 things: CPU Data, Autoscaling Algo, Parameters

```
TIMESTAMP,CPU_USAGE_ACTUAL
2023.04.02-00:09:00:000,7.2
2023.04.02-00:10:00:000,7.04
2023.04.02-00:11:00:000,6.88
2023.04.02-00:12:00:000,6.72
2023.04.02-00:13:00:000,6.48
2023.04.02-00:14:00:000,6.50
2023.04.02-00:15:00:000,6.52
2023.04.02-00:16:00:000,6.54
2023.04.02-00:17:00:000,6.56
```

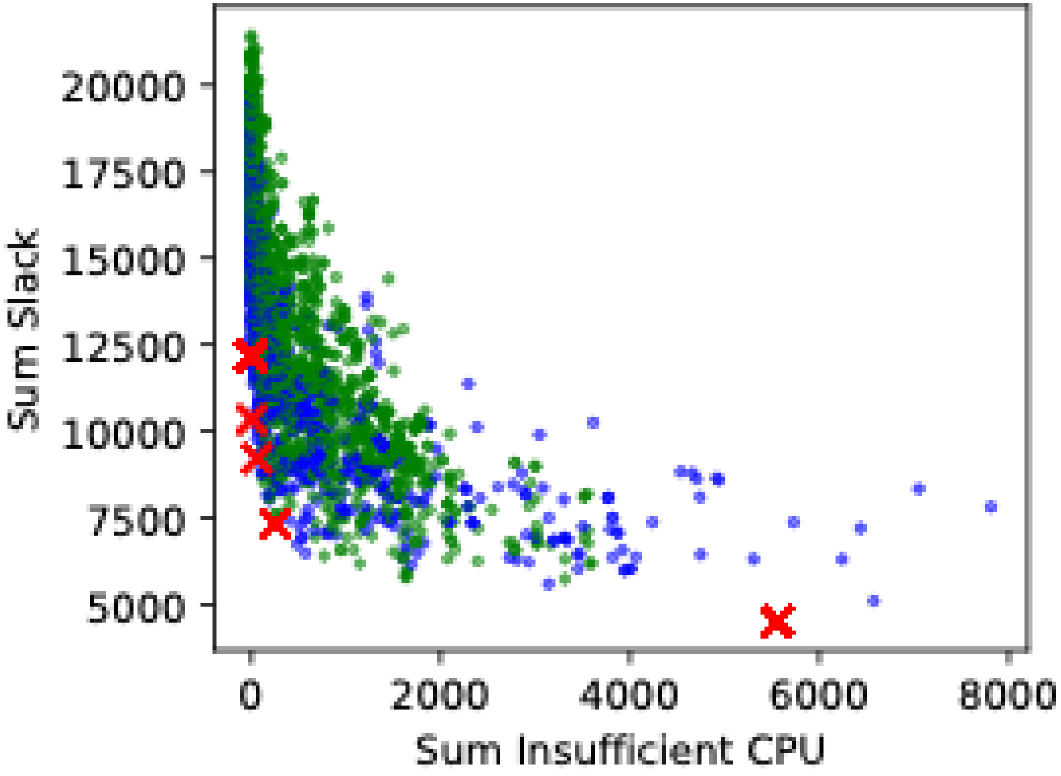
```python
class SimpleAdditiveRecommender(Recommender):
    def __init__(self, cluster_state_provider, save_metadata=True):
        # Copy the code at the top of this function as-is.

        # Put your parameters here hard-coded, or pass them in to your
        # `metadata.json` file in the `algo_specific_config` section.
        self.my_param = self.algo_params.get("myparam", 2)

    def run(self, recorded_data):
        """
        This method runs the recommender algorithm and returns the new number of
              cores to scale to (new limit).

        Inputs:
            recorded_data (pd.DataFrame): The recorded metrics data for the current time window to
simulate
        Returns:
            latest_time (datetime): The latest time of the performance data.
            new_limit (float): The new number of cores to scale to.
        """

        # Your logic goes here! Look at the data in the `recorded_data` dataframe,
        #   do a calculation, and return the number of cores to scale to.

        return new_limit
```

```json
"algo_specific_config": {
    "addend": 2
},
"general_config": {
    "window": 20,
    "lag": 10,
    "max_cpu_limit": 25,
    "min_cpu_limit": 2.0
},
```

Simulating & Tuning parameters

When selecting parameters, we must find the ideal balance between:

- slack (resources wasted)
- insufficient CPU (throttling)

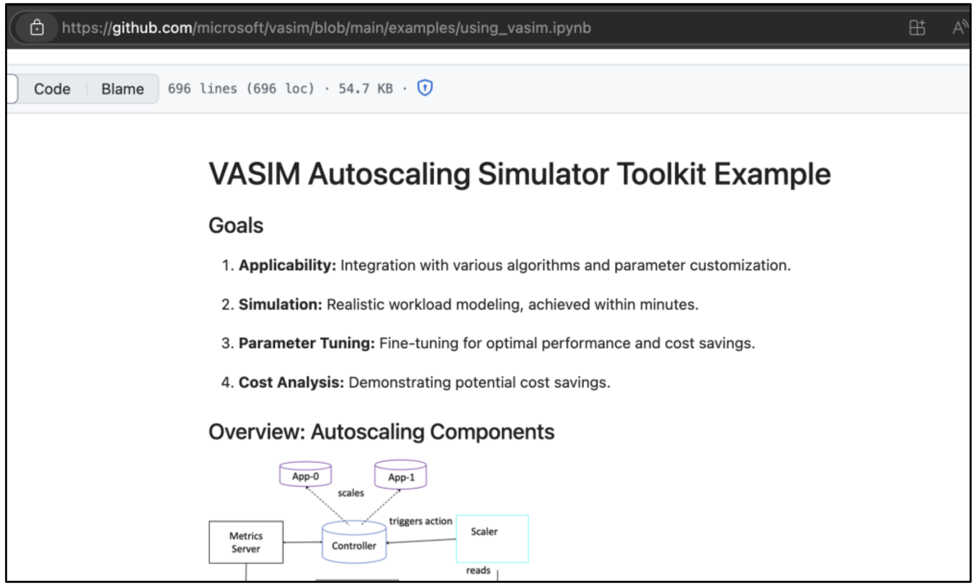


Outline

- Mechanism
  - Vertically scaling in-place
  - Cassandra perf impacts?
- Policy
  - CaaSPER: Proactive/Reactive algorithm for balancing price-perf trade-off
- Code/demo: VASIM - Vertical Autoscaling SIMulator
  - Try your own autoscaling algorithm!
  - Autotuning: parameter tune your own algorithm
  - How to get started with Cassandra

(Go to GitHub…)

https://github.com/microsoft/vasim


https://github.com/microsoft/vasim/blob/main/examples/using_vasim.ipynb

Code Blame 696 lines (696 loc) · 54.7 KB

VASIM Autoscaling Simulator Toolkit Example

Goals

1. **Applicability:** Integration with various algorithms and parameter customization.
2. **Simulation:** Realistic workload modeling, achieved within minutes.
3. **Parameter Tuning:** Fine-tuning for optimal performance and cost savings.
4. **Cost Analysis:** Demonstrating potential cost savings.

Overview: Autoscaling Components

VASIM web demo

- Our notebook
  https://github.com/microsoft/vasim/blob/main/examples/using_vasim.ipynb

- Together with Cassandra
  https://github.com/microsoft/vasim/tree/kasaur/e2e-livedemo/examples/cassandra

- And the web front-end
  https://github.com/microsoft/vasim/tree/main/examples/streamlit

References

- Code repo: https://github.com/microsoft/vasim
  - Simulator demo: examples -> streamlit
  - Cassandra demo: examples -> cassandra
- Papers:

  - **VASIM: Vertical Autoscaling Simulator Toolkit** Anna Pavlenko, Karla Saur, Yiwen Zhu, Brian Kroth, Joyce Cahoon, Jesús Camacho Rodríguez. ICDE, 2024. [pdf]

  - **Vertically Autoscaling Monolithic Applications with CaaSPER: Scalable Container-as-a-Service Performance Enhanced Resizing Algorithm for the Cloud** Anna Pavlenko, Joyce Cahoon, Yiwen Zhu, Brian Kroth, Michael Nelson, Andrew Carter, David Liao, Travis Wright, Jesús Camacho Rodríguez, Karla Saur. SIGMOD, 2024. [pdf]