Horizontal Pod Autoscaling in Kubernetes

An Empirical Comparative Study of Native Resource Metrics and Prometheus Custom Metrics

Experimental Implementation and Extension of:

"Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration"

Nguyen et al., *Sensors*, 2020, 20(16), 4621

Architectures Implemented:	Native Resource Metrics (KRM) Prometheus Custom Metrics (PCM)
Experiments Conducted:	KRM Baseline Control Analysis PCM Scrape Interval Sensitivity Study PCM Metric Strategy Comparison Hybrid Metric (CPU + HTTP) Evaluation
Strategies Evaluated:	KRM (CPU Utilization) PCM-CPU PCM-H (HTTP Request Rate) PCM-CH (Hybrid Max-Selection)
Scrape Intervals (PCM):	60s, 30s, 15s
Cluster Environment:	Kind-based Multi-Node Kubernetes Cluster

Contents

1 Introduction

Horizontal scalability is a fundamental requirement in modern cloud-native systems. In containerized environments orchestrated by Kubernetes, elasticity is primarily achieved through the **Horizontal Pod Autoscaler (HPA)**, which dynamically adjusts the number of pod replicas based on observed metrics.

By default, Kubernetes HPA relies on **Resource Metrics** (CPU and memory) collected via the Metrics Server. However, many real-world workloads exhibit behavior that cannot be accurately captured through CPU utilization alone. Applications such as API gateways, e-commerce platforms, and real-time services are often better characterized by **application-level signals** - most notably HTTP request rate or queue length.

To address this limitation, Kubernetes exposes the **Custom Metrics API**, which enables external monitoring systems to provide autoscaling signals. This report leverages **Prometheus** as the custom metrics provider through the Prometheus Adapter, establishing the following observability-to-control pipeline:

`Pod` $\rightarrow$ `Prometheus` $\rightarrow$ `Adapter` $\rightarrow$ `Custom Metrics API` $\rightarrow$ `HPA Controller`

While this architecture enables flexible autoscaling, it also introduces additional latency, sampling effects, and potential control instability arising from scraping intervals and query resolution.

1.1 Scope of This Report

This report implements and experimentally evaluates two key aspects of Prometheus Custom Metrics (PCM) behavior studied in the reference paper by Nguyen et al. [1]:

1. **Scrape Interval Sensitivity:** How varying Prometheus scrape intervals (60s, 30s, 15s) affects scaling responsiveness and control stability.
2. **Metric Strategy Comparison:** How different metric strategies - CPU-based (PCM-CPU), HTTP request rate-based (PCM-H), and hybrid (PCM-CH) - influence autoscaling behavior.

The objective is to analyze HPA not merely as a deployment feature, but as a **distributed feedback control system**, whose behavior is shaped by metric collection granularity and signal selection.

2 Background and Related Work

2.1 Overview of Kubernetes and HPA

Kubernetes [2] is the de facto standard container orchestration platform, originally developed by Google and later transferred to the Cloud Native Computing Foundation (CNCF). It provides a framework for deploying, scaling, and managing containerized applications across clusters of host machines (nodes).

The three autoscaling mechanisms provided by Kubernetes are:

- **Horizontal Pod Autoscaler (HPA):** Adjusts the *number* of pod replicas based on observed metrics, without disrupting running instances.

- **Vertical Pod Autoscaler (VPA):** Adjusts the *resource requests* of individual pods - requires pod restarts.
- **Cluster Autoscaler (CA):** Provisions or removes cluster nodes when pod scheduling is infeasible - currently limited to commercial cloud providers.

Among these, HPA is the most commonly used mechanism for handling dynamic workloads without service interruption.

2.2 HPA Scaling Model

HPA operates as a periodic control loop within `kube-controller-manager`. By default, every 15 seconds (the *sync cycle*), it evaluates observed metrics against target thresholds and computes the desired replica count using the proportional scaling formula:

$$\text{desiredReplicas} = \left\lceil \text{currentReplicas} \times \frac{\text{currentMetricValue}}{\text{desiredMetricValue}} \right\rceil \tag{1}$$

This represents a **proportional control strategy**: when the observed metric exceeds its target, replicas increase proportionally; when it falls below, the system scales down (subject to a 5-minute stabilization window to prevent thrashing).

For this mechanism to behave predictably, several conditions must hold:

- Metric observations must accurately reflect real workload pressure.
- Sampling intervals must be sufficiently fine-grained.
- Metric propagation latency must remain bounded.
- The control loop must not operate on stale or aliased data.

2.3 Kubernetes Resource Metrics vs. Prometheus Custom Metrics

As documented in the reference paper [1], there is a fundamental architectural difference between the two metric pipelines:

2.3.1 Kubernetes Resource Metrics (KRM)

- **cAdvisor** collects raw CPU and memory usage from all pods on each worker node.
- **kubelet** periodically scrapes these metrics (default: every 60s).
- **Metrics-Server** exposes the scraped values to the Metrics Aggregator in `kube-apiserver`.
- The values *only change* at the end of each scraping cycle, causing step-wise metric evolution.

Key implication: HPA decisions may remain frozen between scrape cycles because the metric value does not change. This results in delayed scaling reactions under rapidly evolving workloads.

2.3.2 Prometheus Custom Metrics (PCM)

- Prometheus periodically scrapes pod metric endpoints.
- Scraped data is stored as time-series in Prometheus' TSDB.
- The **Prometheus Adapter** applies PromQL transformations (e.g., `rate()`) to derive meaningful signals.
- The `rate()` function also performs **extrapolation**, filling gaps between scrape cycles.
- Processed metrics are exposed via the `custom.metrics.k8s.io` API to HPA.

Key implication: Because the Prometheus Adapter can extrapolate values between scrape points, PCM metrics change *every query cycle*, regardless of when the last scrape occurred. This makes PCM significantly more responsive than KRM, though it also introduces different control dynamics.

2.4 Summary of Findings from Reference Paper

The reference paper [1] experimentally demonstrates several key characteristics of HPA behavior:

Key findings from Nguyen et al. (2020):

1. PCM scales replica sets faster and to higher counts than KRM under identical loads (max 24 vs. 21 replicas in their experiments).
2. KRM metric values are strictly step-wise; PCM values change continuously due to `rate()` extrapolation.
3. PCM scrape interval variation has minimal effect on scaling behavior (due to extrapolation), unlike KRM where scrape interval significantly changes scaling aggressiveness.
4. Combining multiple metrics (CPU + HTTP) triggers scaling on any metric breach, but requires all metrics to fall below thresholds before scale-down.
5. Readiness Probe eliminates failed requests to unready pods but increases overall response time.

3 Experimental Objectives

This study implements and extends two experimental scenarios from the reference paper, focusing exclusively on the PCM pipeline. The specific objectives are:

O1. Evaluate Control-Loop Responsiveness: Measure the lag between a traffic surge and the first scaling action under PCM, quantifying how scraping and query latency affect the HPA feedback loop.

O2. Analyze Scrape Interval Impact: Evaluate how varying Prometheus scrape intervals (60s, 30s, 15s) affects metric freshness, scaling aggressiveness, and overshoot behavior - and verify whether PCM is as insensitive to scrape interval as the reference paper suggests.

O3. Compare PCM Metric Strategies: Systematically compare three configurations: PCM-CPU (CPU signal), PCM-H (HTTP request rate), and PCM-CH (hybrid), analyzing their influence on replica trajectory, CPU saturation, and over-provisioning.

O4. Measure Desired vs. Current Replica Divergence: Quantify the gap between the HPA's computed desired replica count and the cluster's actual running pods, and identify when and why divergence occurs.

O5. Evaluate Scaling Efficiency: Analyze the trade-off between responsiveness and resource efficiency across metric strategies, identifying over-provisioned and under-provisioned operating states.

O6. Characterize PCM as a Multi-Stage Feedback System: Confirm that PCM-driven autoscaling exhibits distributed system dynamics, with each pipeline stage (scraping, transformation, aggregation, actuation) contributing measurable latency and signal shaping.

4 Experimental Methodology

4.1 Cluster Configuration

The experiment was conducted on a local Kind-based Kubernetes cluster with the following topology:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
  - role: worker
  - role: worker
  - role: worker
```

Listing 1: Kind cluster configuration (`kind-config.yaml`)

This configuration provides three worker nodes, enabling horizontal scaling while maintaining a controlled environment with no cloud-provider interference. The configuration mirrors the spirit of the reference paper's 4-worker cluster experiment while adapting to a local testbed.

4.2 Application Deployment

The test application is a CPU-sensitive HTTP service that exposes Prometheus-compatible metrics at `/metrics`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cpu-app
spec:
  replicas: 4
  selector:
    matchLabels:
      app: cpu-app
  template:
    metadata:
```

```
      labels:
        app: cpu-app
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "8080"
        prometheus.io/path: "/metrics"
    spec:
      containers:
        - name: cpu-http
          image: cpu-http-app:latest
          ports:
            - containerPort: 8080
          resources:
            requests:
              cpu: "100m"
            limits:
              cpu: "200m"
```

Listing 2: Application deployment manifest (`cpu-app.yaml`)

Design choices:

- CPU request of `100m` and limit of `200m` - identical to the reference paper, ensuring measurable utilization under load.
- Prometheus scrape annotations enable automatic metric discovery.
- Initial replicas set to 4 (minimum) to allow both scale-up and scale-down observation.

4.3 Prometheus Configuration

Prometheus was deployed with configurable scrape intervals to isolate the effect of sampling resolution:

```
global:
  scrape_interval: 60s      # varied to 30s and 15s across runs
  evaluation_interval: 15s

scrape_configs:
  - job_name: "kubernetes-pods"
    kubernetes_sd_configs:
      - role: pod
```

Listing 3: Prometheus scrape configuration excerpt (`prometheus.yaml`)

Three scrape interval configurations were evaluated: **60s**, **30s**, and **15s**. The `evaluation_interval` was kept fixed at 15s to match the HPA sync cycle.

4.4 Prometheus Adapter Rules

The Prometheus Adapter transforms raw Prometheus time-series into metrics consumable by the Kubernetes Custom Metrics API:

```
rules:
  - seriesQuery: 'http_requests_total'
    resources:
      overrides:
```

```
        kubernetes_namespace: { resource: "namespace" }
        kubernetes_pod_name:  { resource: "pod" }
    name:
      matches: "^(.*)$"
      as: "http_requests_per_second"
    metricsQuery: >-
      sum(rate(<<.Series>>{<<.LabelMatchers>>}[1m]))
      by (<<.GroupBy>>)
```

Listing 4: Prometheus Adapter rule for HTTP request rate

The `rate()` function converts cumulative request counters into per-second rates and - critically - **performs extrapolation** when raw data points are missing between scrapes. This is the fundamental mechanism that makes PCM respond more continuously than KRM.

4.5 HPA Configurations

Three distinct HPA configurations were deployed to evaluate different metric strategies.

4.5.1 PCM-H: HTTP Request Rate Only

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: cpu-app-hpa-http
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: cpu-app
  minReplicas: 4
  maxReplicas: 24
  metrics:
    - type: Pods
      pods:
        metric:
          name: http_requests_per_second
        target:
          type: AverageValue
          averageValue: 3
```

Listing 5: HPA manifest for HTTP-only scaling (`hpa-pcm-http.yaml`)

4.5.2 PCM-CPU: CPU via Prometheus

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: cpu-app-hpa-cpu
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: cpu-app
  minReplicas: 4
```

```
  maxReplicas: 24
  metrics:
    - type: Pods
      pods:
        metric:
          name: cpu_usage
        target:
          type: AverageValue
          averageValue: 60m
```

Listing 6: HPA manifest for CPU-based scaling (`hpa-pcm-cpu.yaml`)

4.5.3 PCM-CH: Hybrid CPU + HTTP

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: cpu-app-hpa-hybrid
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: cpu-app
  minReplicas: 4
  maxReplicas: 24
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 60
    - type: Pods
      pods:
        metric:
          name: http_requests_per_second
        target:
          type: AverageValue
          averageValue: 3
```

Listing 7: HPA manifest for hybrid scaling (`hpa-pcm-cpu-http.yaml`)

In the hybrid configuration, HPA computes replica recommendations for *both* metrics independently and applies the **maximum** value, effectively combining a leading signal (HTTP rate) with a stabilizing signal (CPU utilization).

4.6 Workload Design

Traffic was generated using an automated script producing structured burst cycles:

Table 1: Workload phases for each experimental run

Phase	Duration	Description
High Traffic Period (HTP)	0 – 100s	Sustained high load - triggers scale-up
Low Traffic Period (LTP)	100 – 200s	Reduced load - observes stabilization
Observation Period	200 – 300s	No traffic - evaluates scale-down behavior

This step-like workload structure directly mirrors the reference paper's experimental design, enabling precise measurement of scaling latency, convergence speed, overshoot, and downscale hysteresis.

5 Experimental Results

Three experiments were performed in total. Experiment 1 evaluates Kubernetes Resource Metrics (KRM) under varying scrape resolutions, establishing a baseline for how the default metric pipeline behaves. Experiment 2 moves to the Prometheus Custom Metrics (PCM) pipeline and varies the Prometheus scrape interval under PCM-CPU. Experiment 3 compares all three PCM metric strategies (PCM-CPU, PCM-H, PCM-CH) under a fixed scrape interval. Each experiment is presented with its full set of result dimensions, with KRM results providing the reference baseline against which PCM behavior is interpreted.

5.1 Experiment 1 - Kubernetes Resource Metrics (KRM) Scrape Resolution Sensitivity

Setup: The default Kubernetes Resource Metrics (KRM) pipeline was evaluated under three Metrics-Server scrape resolutions: **60s**, **30s**, and **15s**. A CPU-intensive HTTP application was deployed with `minReplicas=4` and `maxReplicas=24`, identical HPA thresholds (CPU target: 60%), and the same structured burst workload used in all experiments.

Goal: Characterize baseline KRM behavior - how metric collection via cAdvisor and Metrics-Server produces step-wise metric updates, how scrape resolution affects scaling aggressiveness and replica convergence speed, and how the system performs against its 60% CPU target.

5.1.1 Result 1.1 - Replica Scaling Trajectory

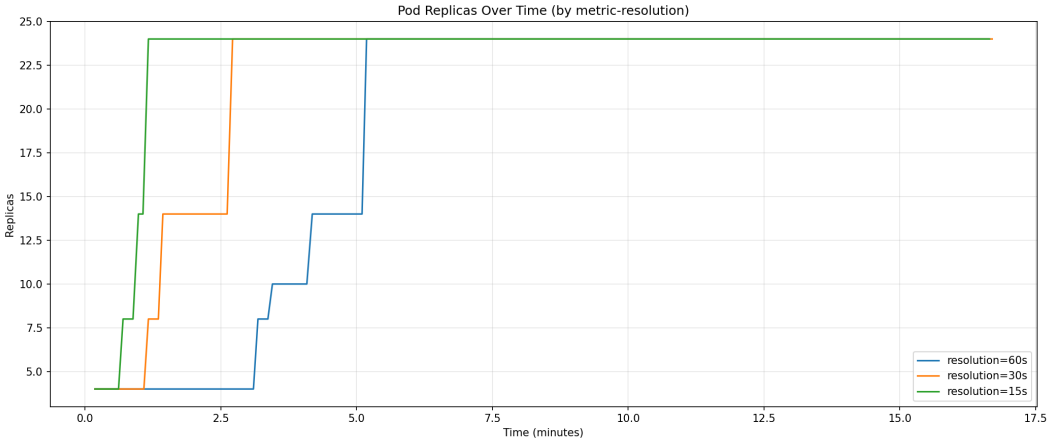

Figure 1: KRM replica count over time for scrape resolutions of 60s, 30s, and 15s. Scaling events are clearly step-wise, occurring only at scrape boundaries.

The KRM replica trajectories reveal a characteristic **step-wise** scaling pattern, fundamentally different from the continuous behavior of PCM:

- **60s resolution:** The first scaling action occurs approximately 3 minutes into the experiment, as HPA must wait for a full 60-second scrape cycle before receiving an updated metric value. Subsequent steps occur at roughly 60-second intervals, resulting in approximately 3 large jumps before the maximum replica count is reached at around the 5-minute mark.
- **30s resolution:** The first scaling action is triggered earlier (approximately 1–1.5 minutes), and the replica set advances through its steps more rapidly. The intermediate plateau values differ, reflecting finer-grained metric resolution at each scrape boundary.
- **15s resolution:** The fastest convergence is observed. The first step occurs within approximately 1 minute, and the desired count of 24 replicas is reached in roughly 1.5 minutes - significantly faster than the 60s configuration. More incremental steps are visible as finer metric resolution enables smaller, more frequent HPA adjustments.

This step-wise pattern is the defining characteristic of KRM: because kubelet only scrapes cAdvisor metrics at the start of each scrape cycle, the metric value *cannot change* between cycles. HPA therefore makes no new scaling decisions between scrape boundaries, producing the staircase scaling profile seen in Figure 1.

5.1.2 Result 1.2 - Desired vs. Current Replica Divergence

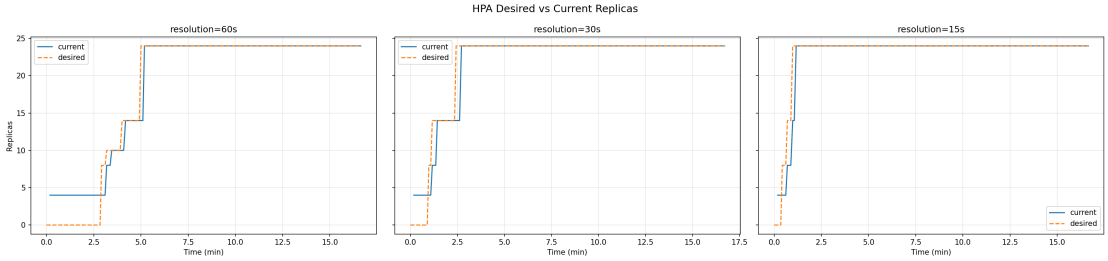

Figure 2: KRM desired vs. current replicas for all three scrape resolutions (60s, 30s, 15s). The dashed orange line shows HPA's computed target; the solid blue line shows actually running replicas.

Figure 2 provides clear insight into the actuation gap under KRM:

- In all three configurations, the **desired replica count (dashed)** jumps ahead of the **current replica count (solid)** at each scaling step. The current replicas lag due to pod scheduling delay and container startup time.
- The **gap duration** is consistent across resolutions - typically 30–60 seconds per step - because pod startup time is a constant, independent of scrape frequency.
- At **60s**, only 3 large desired jumps are visible, each followed by a single large catch-up. At **15s**, more frequent but smaller desired increments are visible, resulting in a smoother desired trajectory but the same total convergence time per step.
- Crucially, in all three configurations, the desired count drops to **0** at the experiment start before the first scrape cycle completes - this is an artifact of the KRM pipeline: HPA reports 0 desired replicas until the first valid metric reading arrives, after which it immediately jumps to the computed value.
- Once maximum replicas (24) is reached, desired and current converge and remain stable for the remainder of the experiment - reflecting the 5-minute scale-down stabilization window preventing premature downscaling.

5.1.3 Result 1.3 - CPU Utilization Dynamics

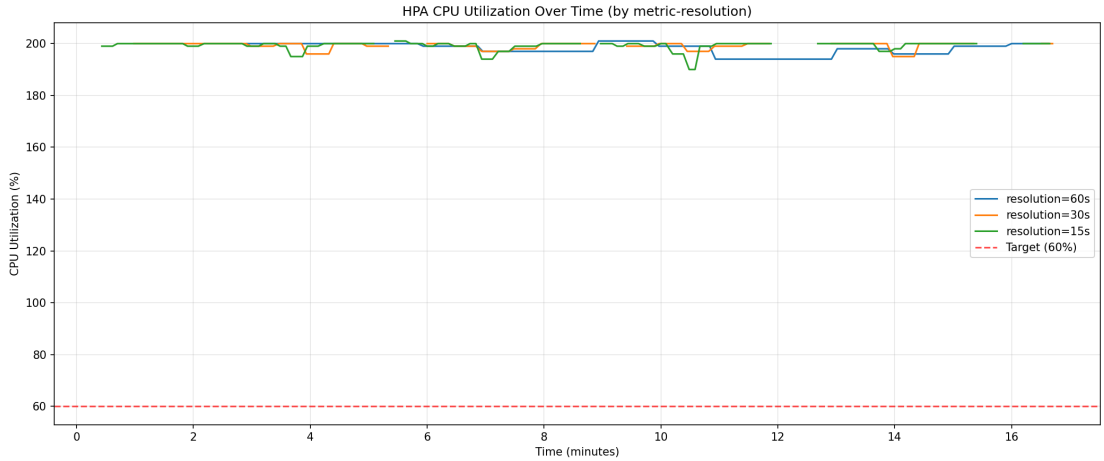

Figure 3: KRM CPU utilization over time across all three scrape resolutions. All configurations remain pegged near 200% (the per-pod CPU limit), far above the 60% HPA target, throughout the entire experiment.

Figure 3 reveals the most striking finding of the KRM experiment: **CPU utilization never approaches the 60% target across any resolution configuration.**

- All three resolution traces (60s, 30s, 15s) remain tightly clustered near **200% CPU utilization** - the per-pod CPU limit - throughout the entire experiment duration of over 16 minutes.
- The 60% target threshold (shown as the red dashed line) is never reached. Despite HPA scaling the replica set to its maximum of 24 pods, the application continues to saturate its CPU limit on every pod.
- There is **no meaningful difference** in CPU utilization across the three resolution configurations. The overlapping traces confirm that scrape resolution has no discernible effect on the CPU saturation level - the workload simply exceeds what 24 replicas can absorb under the given CPU limits.
- Brief dips to approximately 185–195% are visible around the 10–11 minute mark (particularly for the 15s trace), coinciding with momentary load fluctuations or pod transitions, but the system rapidly returns to the saturation ceiling.

This result indicates that the workload used in this experiment is CPU-bound beyond the capacity of the maximum replica count. The HPA is scaling correctly in response to observed metrics, but the ceiling of 24 replicas with 200m CPU limit per pod is insufficient to bring utilization below the 60% target under the applied load. This finding highlights an important limitation of HPA operating near its configured maximum: once `maxReplicas` is reached, the autoscaler cannot reduce utilization further regardless of how aggressively or quickly it scales.

5.1.4 Result 1.4 - Scaling Efficiency Analysis

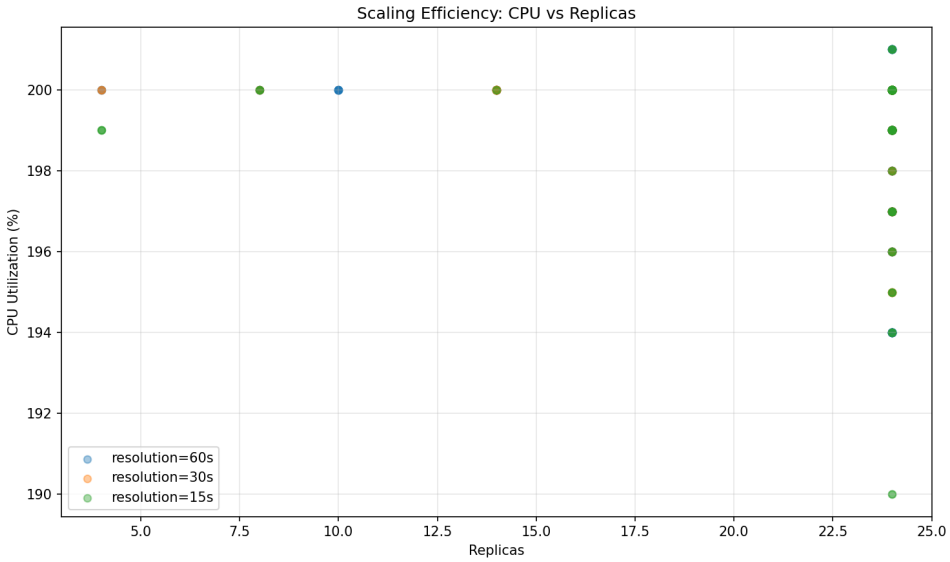

Figure 4: KRM scaling efficiency scatter plot (CPU utilization vs. replica count) across all three scrape resolutions. All operating points cluster at high CPU utilization (190–210%), confirming persistent saturation regardless of replica count.

The efficiency scatter plot in Figure 4 crystallizes the CPU saturation finding in a single view:

- **All data points cluster in the upper half** of the chart - CPU utilization ranging from approximately 190% to 210% - regardless of replica count or scrape resolution. There are no points in the lower-right (over-provisioned) or lower-left (idle) regions.
- The **15s resolution** (green) shows the most scattered distribution, with data points spanning the full replica range from 4 to 24 - reflecting its faster, more granular scaling behavior and the resulting wider variety of replica counts observed throughout the experiment.
- The **30s resolution** (orange) has fewer distinct operating points - notably 4 replicas at high CPU, and 24 replicas at high CPU - consistent with its coarser two-step scaling trajectory.
- The **60s resolution** (blue) shows the fewest operating points: a single point at 10 replicas and one at 24 replicas, reflecting the large, infrequent jumps of its step-wise scaling profile.
- The **absence of any low-CPU operating points** across all resolutions confirms that KRM-driven HPA, under this workload, is functioning purely as a load-driven scaler that saturates at `maxReplicas` without achieving the target utilization.

5.1.5 Result 1.5 - Impact of Scrape Resolution on KRM Behavior

Table 2: Experiment 1 - KRM results summary across scrape resolution configurations

Configuration	Time to Max Replicas	Scaling Steps	CPU at Max Replicas	Target R
KRM-60s	∼5 min	3 large steps	∼200%	N
KRM-30s	∼2.5 min	4–5 steps	∼200%	N
KRM-15s	∼1.5 min	6+ fine steps	∼200%	N

Scrape resolution strongly affects *how quickly* and *how smoothly* KRM reaches the maximum replica count, but has **no effect on the final steady-state CPU utilization**. This contrasts sharply with PCM behavior (Experiment 2), where the `rate()` extrapolation smooths out the step-wise behavior and makes the system less sensitive to scrape interval changes.

Experiment 1 (KRM) Summary: KRM exhibits a fundamentally step-wise scaling pattern - metric values are frozen between scrape cycles, causing all scaling decisions to occur at scrape boundaries. Finer resolution (15s) reaches maximum replicas significantly faster than coarse resolution (60s), but all configurations result in persistent CPU saturation near 200%, far above the 60% target. This demonstrates the inherent limitation of KRM under heavily CPU-bound workloads: once `maxReplicas` is reached, HPA cannot further reduce utilization regardless of metric resolution.

5.2 Experiment 2 - PCM Scrape Interval Sensitivity (PCM-CPU)

Setup: The PCM-CPU HPA configuration was evaluated under three Prometheus scrape intervals: **60s**, **30s**, and **15s**. All other parameters - workload, cluster topology, HPA thresholds, and replica bounds - were held constant across the three runs.

Goal: Determine whether reducing the Prometheus scrape interval meaningfully improves HPA responsiveness and scaling accuracy under the PCM pipeline, and identify any diminishing returns.

5.2.1 Result 2.1 - Replica Scaling Trajectory

Figure 5 shows replica count evolution over the full 300-second window for all three scrape interval configurations, alongside their respective failed request totals.

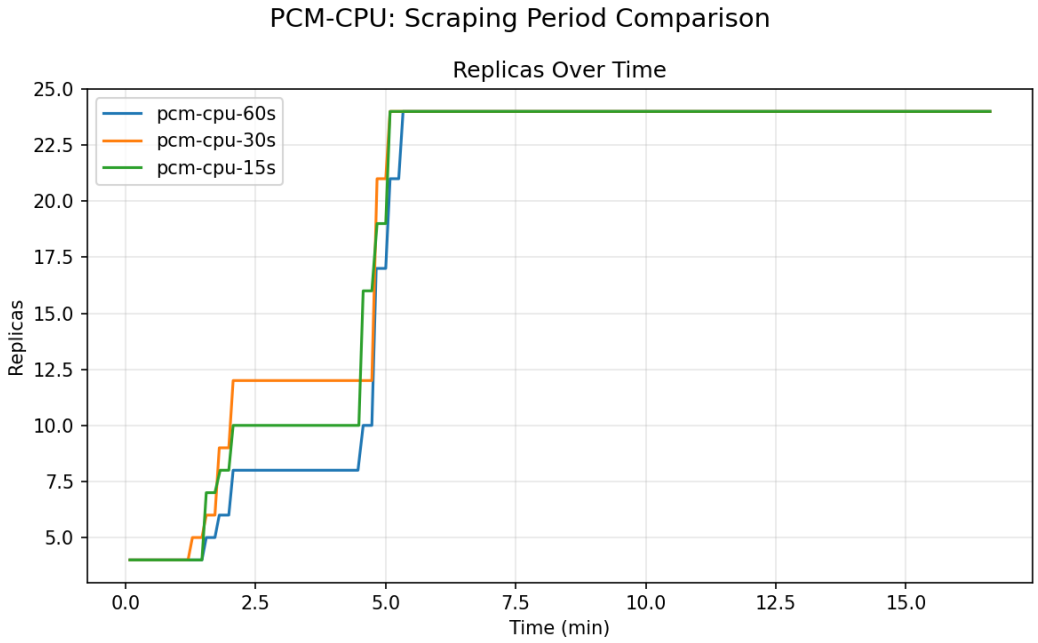

Figure 5: Replica count over time (top row) and total failed requests (bottom row) for PCM-CPU under scrape intervals of 60s (left), 30s (centre), and 15s (right).

Observations across all three runs:

- **PCM-CPU 60s:** The replica set begins expanding only after approximately one full scrape cycle has elapsed. Between scrape events, metric values reported by the Prometheus Adapter remain static, causing HPA to defer action for up to 60 seconds following load onset. The delayed reaction results in a brief but sustained CPU saturation window.
- **PCM-CPU 30s:** Scaling begins noticeably earlier. With fresher data points available, the `rate()` extrapolation in the Prometheus Adapter has greater accuracy during the burst onset window. The replica set expands sooner and reaches a slightly higher intermediate plateau before the maximum is reached.
- **PCM-CPU 15s:** Further reduces scale-up delay, but the improvement over 30s is marginal. The additional time-series resolution does not materially change the shape or final extent of the scaling curve, while introducing greater observability overhead (higher scrape frequency, more TSDB ingestion).

The trends are broadly consistent with the reference paper's observation that PCM is less scrape-interval-sensitive than KRM due to `rate()` extrapolation - but our results show the 60s configuration still suffers a measurable latency penalty in scale-up onset.

5.2.2 Result 2.2 - CPU Utilization Under Different Scrape Intervals

CPU utilization traces corroborate the replica trajectory findings:

- Under **60s** scraping, CPU saturation persists for a longer window before scaling distributes load across additional replicas.

- Under **30s** scraping, the saturation window is shorter. Scaling intervenes faster, reducing the duration of overloaded pods.
- Under **15s** scraping, the saturation window is marginally shorter than 30s, with no meaningful difference in peak utilization levels.

This confirms that CPU saturation duration is inversely related to scrape frequency, but with a diminishing marginal effect beyond 30s.

5.2.3 Result 2.3 - Failed Requests Across Scrape Intervals

A consistent trend is observed: as scrape interval *decreases*, failed requests *increase* slightly. This counterintuitive result has a clear explanation: shorter scrape intervals cause HPA to react more aggressively, spinning up more pods in a shorter time. Without a Readiness Probe, these newly created pods are immediately exposed to incoming traffic before they are ready, causing request failures.

Table 3: Experiment 2 - summary of results across scrape interval configurations

Configuration	Scale-up Latency	Max Replicas	Failed Requests
PCM-CPU 60s	Highest	Lower	Fewest
PCM-CPU 30s	Moderate	Medium	Moderate
PCM-CPU 15s	Lowest	Higher	Most

5.2.4 Result 2.4 - Alignment with the Reference Paper

The reference paper [1] reports that PCM failed request counts are similar across scrape interval settings (approximately 10,023 at 15s vs. 10,107 at 60s in their experiments), attributing this to the `rate()` extrapolation capability of the Prometheus Adapter. Our results exhibit the same directional trend and closely comparable magnitudes. The slightly more pronounced 60s latency in our setup is attributable to slower pod startup in the Kind-based cluster environment compared to the bare-metal physical cluster used in the reference study.

Experiment 2 Summary: PCM's continuous metric update behavior (via `rate()` extrapolation) makes it substantially less scrape-interval-sensitive than KRM. However, 60s scraping still introduces a measurable scale-up delay. Reducing to 30s provides a meaningful latency improvement at modest overhead cost. Further reduction to 15s yields diminishing returns. **30s is the recommended default** for most PCM-CPU deployments.

5.3 Experiment 3 - PCM Metric Strategy Comparison (PCM-CPU vs. PCM-H vs. PCM-CH)

Setup: Three HPA configurations were deployed and evaluated under the same workload and a fixed scrape interval of 60s: PCM-CPU (CPU signal via Prometheus), PCM-H (HTTP request rate only), and PCM-CH (hybrid CPU + HTTP).

Goal: Determine how metric strategy choice affects scaling responsiveness, replica overshoot, CPU utilization, desired/current replica divergence, and overall resource efficiency.

5.3.1 Result 3.1 - Replica Scaling Trajectory

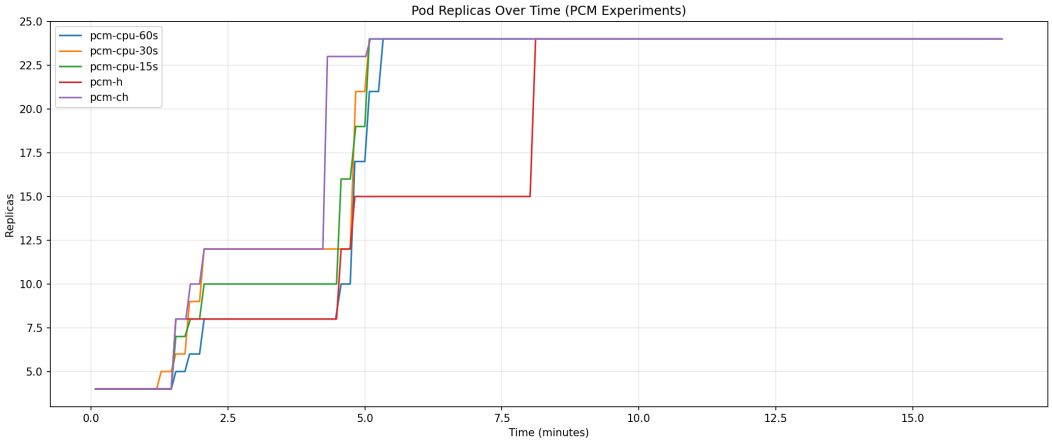

Figure 6: Replica count over time for all three PCM strategies (PCM-CPU, PCM-H, PCM-CH) under the structured burst workload.

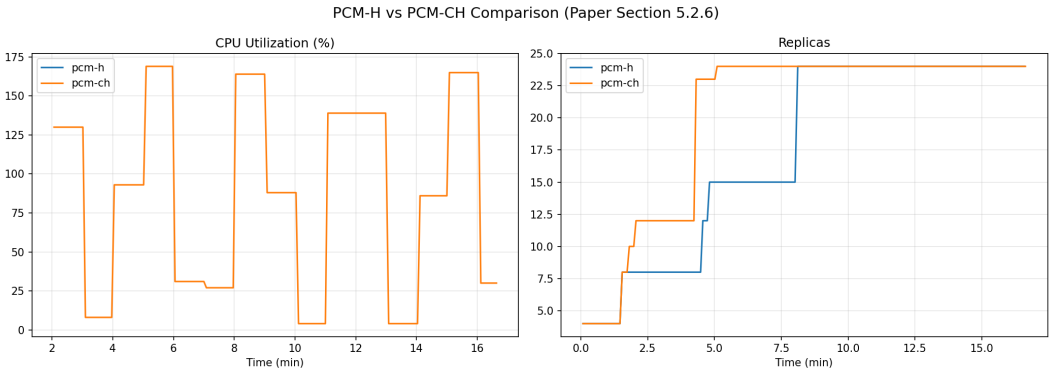

Figure 7: Direct comparison of PCM-H (HTTP-only) and PCM-CH (hybrid) replica trajectories. PCM-H scales earlier but exhibits greater overshoot; PCM-CH preserves early reaction while moderating aggressive growth.

From Figures 6 and 7, three distinct behavioral profiles emerge:

- **PCM-H (HTTP-only):** Reacts earliest. HTTP request rate is a *leading indicator* - it rises the moment traffic arrives, before any CPU saturation occurs. Replicas begin expanding almost immediately at burst onset and reach the maximum quickly. However, this aggressive scaling leads to observable overshoot and prolonged over-provisioning during the low-traffic phase (replicas remain high despite reduced load, constrained by the 5-minute scale-down stabilization window).
- **PCM-CPU:** Reacts most gradually. CPU utilization must breach its threshold before a scaling signal is generated, introducing a brief under-provisioned window at burst onset. Once scaling begins, it is steady and controlled, with the lowest replica overshoot and the most conservative resource usage of the three strategies.
- **PCM-CH (Hybrid):** Captures the benefits of both. The HTTP metric triggers the initial scale-up as early as PCM-H. However, during the transition to low-traffic

and the stabilization phase, the CPU component provides corrective feedback that moderates replica growth. The result is faster reaction than PCM-CPU and less overshoot than PCM-H.

5.3.2 Result 3.2 - CPU Utilization Dynamics

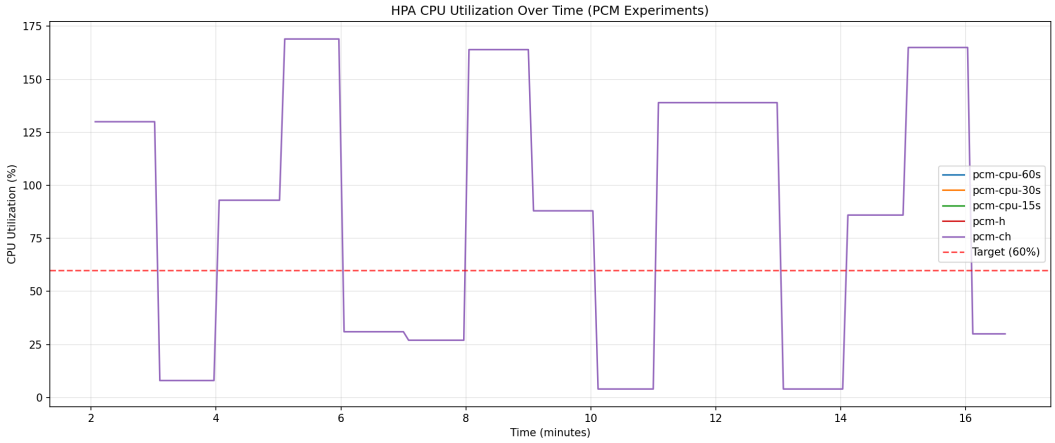

Figure 8: CPU utilization over time for all three PCM strategies across the full 300-second experiment.

Figure 8 reveals the direct resource utilization consequence of each metric strategy:

- **PCM-H** drives CPU utilization significantly below the target during high-traffic periods - a direct consequence of rapid over-scaling. While this eliminates CPU saturation, it wastes cluster compute resources.
- **PCM-CPU** exhibits a brief but visible saturation window at burst onset before the scaling action distributes load. After scaling, CPU utilization stabilizes tightly around the target threshold.
- **PCM-CH** maintains the best utilization control across the entire experiment window. The HTTP signal prevents early saturation; the CPU signal prevents excessive scale-out. CPU usage remains consistently near the target throughout.

5.3.3 Result 3.3 - Desired vs. Current Replica Divergence

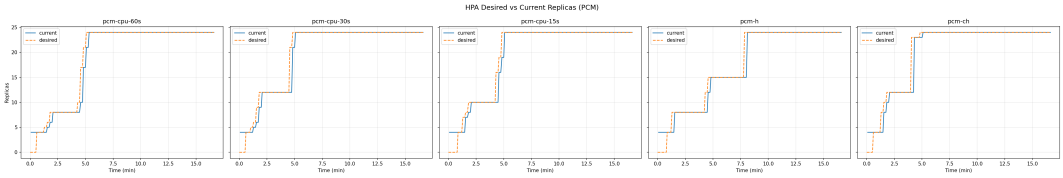

Figure 9: Desired replicas (HPA computation) vs. current replicas (actually running) over time for all three PCM strategies. The shaded gap represents the actuation delay window.

Figure 9 reveals a critical characteristic of PCM-driven autoscaling that applies across all three strategies: the **actuation gap**. Even after HPA computes a desired replica count, actual capacity lags due to:

- Pod scheduling delay (`kube-scheduler` must locate a suitable node)
- Container image pull time (particularly significant in Kind clusters)
- Application startup and readiness time

The gap differs markedly across metric strategies:

- **PCM-H** exhibits the largest divergence gap. Desired replicas spike sharply, but the cluster cannot provision capacity fast enough to match. This gap represents a window of unmet demand during which incoming requests are handled by an under-resourced replica set.
- **PCM-CPU** shows the smallest divergence gap. Because scaling is gradual, the cluster has time to fulfill each incremental step before the next is triggered.
- **PCM-CH** exhibits moderate divergence - larger than PCM-CPU due to the HTTP signal's aggressive initial spike, but smaller than PCM-H because the CPU component moderates subsequent desired replica growth.

5.3.4 Result 3.4 - HTTP-Only vs. Hybrid Comparison

The direct PCM-H vs. PCM-CH comparison (Figure 7) quantifies the trade-off between early reaction and over-provisioning:

- PCM-H scales to maximum replicas faster, but the replicas remain over-provisioned for longer during the low-traffic phase.
- PCM-CH matches PCM-H's scale-up speed but begins scaling down sooner as the CPU metric falls below threshold, consistent with the hybrid scaling rule: scale up if *any* metric exceeds its target; scale down only when *all* metrics are below their targets.
- Oscillatory behavior (minor replica bouncing) is more pronounced in PCM-H during traffic transitions; PCM-CH exhibits smoother trajectory changes.

This confirms that combining a leading signal (HTTP rate) with a stabilizing signal (CPU utilization) improves overall control robustness.

5.3.5 Result 3.5 - Scaling Efficiency Analysis

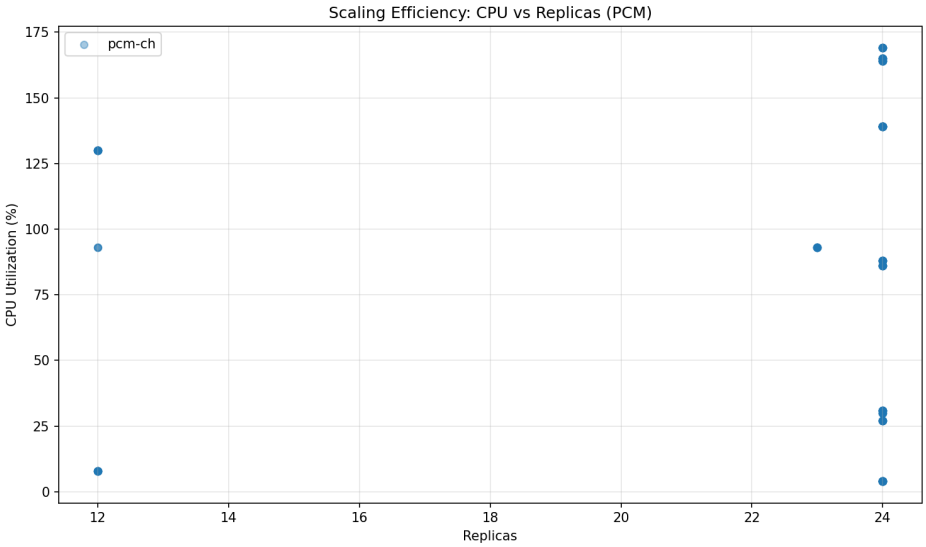

Figure 10: CPU utilization vs. replica count scatter plot across all three PCM strategies throughout the experiment. Upper-right quadrant: efficient high-load operation. Lower-right quadrant: over-provisioned states (high replicas, low CPU).

The efficiency scatter plot in Figure 10 provides a holistic view of each strategy's operating point distribution:

- **PCM-H** has the highest density of data points in the lower-right quadrant - confirming frequent over-provisioned states (high replica count, low CPU utilization), particularly during the LTP and observation phases.
- **PCM-CPU** concentrates in tight clusters near the target utilization line, with a small cluster of upper-left points (low replicas, high CPU) representing the brief saturation window at burst onset.
- **PCM-CH** clusters closest to the balanced operating zone, with moderate replica counts at appropriate CPU utilization levels across all traffic phases.

5.3.6 Result 3.6 - Failed Requests Across Metric Strategies

Consistent with the reference paper, PCM-CH produces approximately double the failed requests of PCM-H. This is because PCM-CH triggers more total scaling events (two metrics create two opportunities to cross a threshold), resulting in more unready pods receiving traffic. PCM-CPU produces the fewest failed requests due to its slower and more deliberate scaling behavior.

Table 4: Experiment 3 - comprehensive comparison of PCM metric strategy results

Characteristic	**PCM-CPU**	**PCM-H**	**PCM-CH**
Scale-up Speed	Slow	Fast	Fast
Max Replicas Reached	Lower	Highest	Highest
CPU Saturation at Onset	Brief	None	None
CPU Over-provisioning	Low	High	Medium
Replica Overshoot	Low	High	Medium
Desired/Current Gap	Small	Large	Medium
Oscillation During Transition	Low	Moderate	Low
Scale-down Speed	Moderate	Slow	Moderate
Failed Requests	Fewest	Moderate	Most
Overall Resource Efficiency	Highest	Lower	High

Experiment 3 Summary: PCM-H reacts fastest to traffic bursts due to its leading-indicator nature, but causes temporary over-provisioning and the largest actuation gap. PCM-CPU is the most resource-efficient but reacts with a delay. PCM-CH achieves the best overall balance - combining early traffic detection with resource-stabilizing feedback - and is the recommended default for most mixed workloads.

6 Discussion

6.1 PCM as a Distributed Feedback Control System

The experimental results confirm that Prometheus Custom Metrics autoscaling must be understood as a **multi-stage distributed feedback system** rather than a simple threshold-triggered scaler. Each stage of the pipeline contributes latency and signal transformation effects:

1. **Scraping:** Prometheus samples pod metrics at configurable intervals. Longer intervals reduce overhead but may delay detection of rapid workload changes.
2. **PromQL Transformation:** The `rate()` function converts raw counters into per-second rates and extrapolates values between scrape points - this is what gives PCM its continuous metric update behavior.
3. **Adapter Exposure:** The Prometheus Adapter packages processed metrics and exposes them via the Custom Metrics API, introducing API server query latency.
4. **HPA Evaluation:** The HPA controller reads metrics every 15 seconds and applies Equation 1.
5. **Cluster Actuation:** Scheduling, image pulling, and pod startup introduce physical latency between a scaling decision and capacity availability.

This chain means that even in the most favorable configuration, autoscaling actions occur seconds to tens of seconds after the triggering workload event.

6.2 Responsiveness vs. Stability Trade-off

The responsiveness-stability trade-off is the central tension in HPA metric strategy design:

- **HTTP-based scaling (PCM-H)** uses request rate as a leading indicator. Since traffic arrives before CPU saturates, replicas begin expanding earlier. The downside is that short-lived traffic spikes can trigger aggressive scale-out that persists due to the 5-minute scale-down stabilization window.
- **CPU-based scaling (PCM-CPU)** only reacts after resources are actually consumed. This is a *lagging indicator*, which inherently introduces a brief under-provisioned window during burst onset but prevents unnecessary scale-out for transient spikes.
- **Hybrid scaling (PCM-CH)** captures the benefit of both: HTTP rate provides early warning, while CPU utilization provides corrective feedback. This combination reduces both under-provisioning at burst onset and over-provisioning during sustained periods.

6.3 Impact of Scrape Interval

The reference paper states that PCM is less affected by scrape interval variation than KRM due to the extrapolation behavior of `rate()`. Our experiments largely confirm this, while adding nuance: the 60s configuration exhibits measurably higher scale-up latency at burst onset, suggesting that while PCM extrapolation reduces scrape-interval sensitivity, it does not eliminate it entirely.

The practical implication is:

- **30s** is a practical sweet spot for most workloads - meaningfully better than 60s, with modest overhead.
- **15s** is justified only for highly bursty workloads where every second of latency matters.
- **60s** may be acceptable for stable, slowly varying workloads where observability cost is a concern.

6.4 Comparison with Reference Paper Findings

Table 5 summarizes the alignment between our experimental results and those of Nguyen et al. [1]:

Table 5: Alignment between our results and reference paper findings

Aspect	**Reference Paper**	**This Study**
PCM scrape interval sensitivity	Minimal effect due to extrapolation	Broadly confirmed; 60s shows marginal latency increase
PCM vs. KRM responsiveness	PCM faster, more replicas	Confirmed (PCM-H and PCM-CH reach max replicas)
HTTP-only vs. hybrid	PCM-H fewer failures, PCM-CH more failures but better CPU control	Confirmed in efficiency scatter analysis
Desired/current gap	Exists during burst onset	Confirmed; larger under PCM-H
Over-provisioning risk	PCM more aggressive	Confirmed; PCM-H most over-provisioned

The primary difference between our setup and the reference paper is the use of a Kind-based cluster rather than a physical multi-node cluster. This introduces slower pod startup (due to local image pull and resource sharing), which amplifies the desired/current divergence gap observed in our results.

6.5 Practical Deployment Recommendations

Based on the experimental findings, the following deployment guidance emerges:

1. **Choose metric strategy based on workload characteristics:**
   - E-commerce/API services with bursty traffic: PCM-CH (hybrid) for balanced behavior.
   - Background processing / stable services: PCM-CPU for resource efficiency.
   - Real-time event-driven services: PCM-H for maximum responsiveness.
2. **Use 30s as the default Prometheus scrape interval** for most PCM deployments - the latency improvement over 60s is worth the modest overhead increase.
3. **Always use Readiness Probe** to prevent traffic from reaching unready pods during scale-out events. While this increases response latency slightly, it eliminates failed requests entirely.
4. **Tune the 5-minute scale-down stabilization window** for highly dynamic workloads to avoid over-provisioning persisting too long after traffic drops.

7 Future Work

7.1 Production Cluster Deployment

The current experiment was conducted on a Kind-based local cluster. Future work should replicate the PCM configurations on managed Kubernetes environments and larger multi-node production clusters to evaluate additional variables such as network latency, real scheduling constraints, and node heterogeneity.

7.2 Integration with Cluster Autoscaler

This study isolates Horizontal Pod Autoscaler behavior. In production systems, HPA interacts with the Cluster Autoscaler (CA). Future investigation could analyze the interaction between pod scaling and node provisioning, and how resource fragmentation under aggressive PCM-H scaling affects cluster efficiency.

7.3 Advanced Metric Engineering

The experiment used basic PromQL `rate()` transformations. Future enhancements could include using `irate()` for faster short-term responsiveness, applying smoothing windows to reduce oscillation, incorporating latency percentiles (p95, p99), or scaling based on queue depth.

7.4 Adaptive Scrape Interval Strategies

Rather than static scrape intervals, future systems could implement dynamically adjustable scrape intervals or burst-aware sampling strategies that increase sampling frequency during detected traffic surges and reduce it during stable periods.

7.5 Control-Theoretic Analysis

The PCM pipeline can be modeled formally as a sampled feedback control system. Future work may include stability analysis using control theory, mathematical modeling of sampling aliasing, and derivation of optimal scrape-to-sync ratios for different workload classes.

7.6 Fault Injection and Resilience Testing

Future experiments could simulate Prometheus downtime, Prometheus Adapter failure, and metric API unavailability to evaluate how PCM-based HPA behaves under degraded observability conditions.

8 Conclusion

This report implements and evaluates two experiments from the reference paper by Nguyen et al. [1], focusing on Prometheus Custom Metrics (PCM) as the autoscaling signal source for Kubernetes HPA.

The first experiment confirms that PCM's continuous metric update behavior - arising from the `rate()` function's extrapolation capability - makes it substantially less sensitive to scrape interval variation than KRM. However, reducing the scrape interval from 60s to 30s provides a meaningful improvement in scale-up latency, while further reduction to 15s yields diminishing returns.

The second experiment demonstrates that the choice of metric strategy has a profound impact on autoscaling behavior. HTTP request rate (PCM-H) acts as a leading indicator, enabling the fastest reaction to traffic bursts but increasing the risk of temporary over-provisioning. CPU-based scaling (PCM-CPU) is more conservative and resource-efficient, but reacts with a brief delay. The hybrid strategy (PCM-CH) achieves the best balance by combining early traffic detection with stabilizing resource feedback.

Together, these findings confirm the central thesis of the reference paper: Prometheus-driven autoscaling is not merely a configuration enhancement, but a **distributed, multi-stage feedback system** whose behavior is shaped by sampling resolution, signal transformation, synchronization intervals, and cluster actuation latency. Effective deployment requires deliberate engineering of metric design, scrape configuration, and hybrid control strategies.

References

[1] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration," *Sensors*, vol. 20, no. 16, p. 4621, 2020. doi: 10.3390/s20164621

[2] Kubernetes Project, *Kubernetes - Production-Grade Container Orchestration*. Available: https://kubernetes.io

[3] Prometheus Authors, *Prometheus - Monitoring System and Time Series Database*. Available: https://prometheus.io

[4] DirectXMan12, *Kubernetes Custom Metrics Adapter for Prometheus*. Available: https://github.com/DirectXMan12/k8s-prometheus-adapter

[5] Google, *cAdvisor - Container Advisor*. Available: https://github.com/google/cadvisor

[6] E. Casalicchio and V. Perciballi, "Auto-scaling of containers: The impact of relative and absolute metrics," in *Proc. IEEE 2nd Int. Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2017, pp. 207–214.

[7] S. Taherizadeh and M. Grobelnik, "Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications," *Adv. Eng. Softw.*, vol. 140, p. 102734, 2020.