Characterizing Horizontal Pod Autoscaler Usage in Open-Source Kubernetes Projects: A Large-Scale Empirical Study

Thiago Serafim Rodrigues
University of Fortaleza
Fortaleza, CE, Brazil
thiago.callyxz@gmail.com

Humberto F. Ribeiro Júnior
University of Fortaleza
Fortaleza, CE, Brazil
humbertofraga@gmail.com

Nabor C. Mendonça
University of Fortaleza
Fortaleza, CE, Brazil
nabor@unifor.br

Abstract

The Horizontal Pod Autoscaler (HPA) is Kubernetes' primary mechanism for workload-based horizontal scaling. Although the HPA API has evolved from a CPU-oriented interface into a richer control surface for multi-metric and behavior-aware scaling, there is still limited empirical evidence on how this API is used in practice. This paper presents a large-scale empirical study of HPA usage in Kubernetes-based open-source projects. We mined 12,741 GitHub repositories and analyzed 9,764 HPA files comprising 16,506 metric definitions. The study addresses four research questions covering API adoption and capability evolution, configuration richness, HPA-file maintenance and co-evolution, and statically detectable HPA configuration smells that may deserve inspection. To support the analysis, we introduce the HPA Feature Richness Score (HFRS), a three-dimensional structural metric capturing Signal Richness, Control Richness, and Adjustment Richness. Our results show a persistent gap between API adoption and capability utilization: `autoscaling/v2` is now the dominant API version, but the median HFRS is only 4 out of 13, and the most prevalent archetype is *Basic v2 adoption*, in which manifests declare a v2-family API while retaining a single CPU metric and no explicit behavior configuration. We also find that configuration richness is associated with HPA-file maintenance: HPA-active repositories contain richer configurations, while older and frozen HPA files are concentrated in simpler archetypes. Finally, we identify recurring HPA configuration smells, including legacy autoscaling/v1 usage in HPA-active repositories, structurally minimal v2-family configurations, fixed-replica HPA declarations, unsupported spec.behavior fields, and unusually low CPU utilization targets. The results provide a basis for auditing HPA manifests and for investigating why projects adopt newer API versions without using their richer scaling capabilities.

Keywords

Kubernetes, Horizontal Pod Autoscaler, Autoscaling, Infrastructure as Code, Mining Software Repositories, API Evolution

1 Introduction

Cloud-native computing has become a dominant paradigm for deploying scalable and manageable software systems, with Kubernetes serving as the de-facto orchestration platform [8]. Within this ecosystem, autoscaling is a central operational mechanism: it directly affects service reliability, resource efficiency, and the cost of running applications under variable workloads [16].

Kubernetes resources are commonly declared through YAML manifests that describe the desired state of the application and its supporting infrastructure. Among these resources, the Horizontal Pod Autoscaler (HPA) specifies how the number of replicas of a target workload, such as a `Deployment` or `StatefulSet`, should change in response to observed metrics [11]. The HPA periodically observes resource, custom, or external metrics and adjusts the number of replicas of its target workload. Although its basic control loop is simple, the HPA API has evolved from a CPU-oriented interface into a richer configuration surface that supports multi-metric scaling, custom and external metric sources, per-container metrics, stabilization windows, scaling policies, and direction-specific behavior configuration [11, 12]. As a result, the `apiVersion` declared in an HPA manifest may reveal only part of the story: a manifest may adopt a newer API version while using only a small subset of the capabilities that version provides.

This distinction between API adoption and capability utilization is important because Kubernetes configuration files are often copied, adapted incrementally, and left unchanged after the initial deployment setup. Public repositories may therefore contain a heterogeneous mixture of current operational configurations, tutorial and example manifests, including boilerplate adapted from documentation, prototype manifests, abandoned files, and superficially updated manifests that declare a newer API while retaining minimal CPU-based behavior. In such cases, adopting a newer API version may have limited practical value if configurations continue to use only the capabilities of the older interface [14]. At the project level, this may indicate missed opportunities to use workload-relevant metrics or more advanced scaling controls; at the ecosystem level, it may reveal API documentation gaps, migration friction, or capabilities whose intended use is not widely understood.

Existing empirical studies on Kubernetes and infrastructure-as-code have characterized manifest smells, security misconfigurations, configuration defects, Helm chart evolution, and other container-orchestration practices [4, 19, 20, 26, 27]. In parallel, autoscaling research has proposed and evaluated improved HPA variants, predictive controllers, and tuning strategies under controlled workloads [1, 3, 17, 24, 25]. However, these studies either examine Kubernetes configuration artifacts broadly or evaluate autoscaling mechanisms in controlled settings; they do not characterize field-level HPA usage at scale or analyze how the evolution of this specific API is reflected in open-source repositories.

This paper addresses this gap through a large-scale empirical study of HPA usage in open-source Kubernetes projects. We mined 12,741 GitHub repositories and analyzed 9,764 HPA files comprising 16,506 metric definitions. The study investigates four aspects of HPA practice: the adoption of API versions and capabilities over time; the richness of HPA configurations; the relationship between HPA configuration and repository maintenance; and recurring HPA configuration smells that may deserve inspection by developers, operators, and tool builders. The goal is not to classify individual manifests

as correct or incorrect, since the adequacy of an autoscaling policy depends on workload-specific and cluster-specific context. Instead, we characterize how the HPA API is used in public open-source repositories and identify systematic configuration-smell indicators that can inform documentation, linting tools, policy engines, and future API evolution.

To support this analysis, we introduce the HPA Feature Richness Score (HFRS), a three-dimensional additive metric that captures Signal Richness (SR), Control Richness (CR), and Adjustment Richness (AR). HFRS makes it possible to compare field-level HPA usage across API versions and to distinguish manifests that merely declare a newer API from those that use the richer scaling signals and behavior controls introduced by that API. We also derive a rule-based taxonomy of five configuration archetypes and catalogue six statically detectable HPA configuration smells, including legacy `autoscaling/v1` usage, structurally minimal v2-family configurations, stale HPA files in otherwise HPA-maintained repositories, fixed-replica HPA declarations, unsupported `spec.behavior` fields, and unusually low CPU utilization targets. Together, these analyses provide, to the best of our knowledge, the first comprehensive view of how Kubernetes HPA manifests are configured, maintained, and underutilized in open-source practice.

In summary, this paper makes three main contributions. First, it provides a large-scale empirical characterization of HPA usage across four API versions in open-source projects, based on 9,764 HPA files and 16,506 metric definitions mined from 12,741 GitHub repositories. Second, it introduces HFRS, a repeatable metric for quantifying HPA configuration richness across signal, control, and adjustment dimensions, together with a rule-based taxonomy of five configuration archetypes that captures how these dimensions combine in practice. Third, it provides a curated dataset and catalogue of recurring HPA configuration smells that can support future research, documentation improvements, and tool-based inspection of Kubernetes autoscaling manifests.

The remainder of this paper is organized as follows. Section 2 provides background on the Kubernetes HPA and its API evolution. Section 3 reviews related work. Section 4 describes the research methodology. Section 5 presents the results. Section 6 discusses implications and Section 7 examines threats to validity. Section 8 concludes the paper.

2 Background

2.1 Horizontal Pod Autoscaler

The Horizontal Pod Autoscaler (HPA) is a Kubernetes API resource and controller that automatically adjusts the number of pod replicas for a target workload, such as a `Deployment` or `StatefulSet`, based on observed resource, custom, or external metrics [11]. Its control loop runs periodically, every 15 seconds by default, and computes the desired replica count using the formula $\text{desiredReplicas} = \left\lceil \text{currentReplicas} \times \frac{\text{currentMetricValue}}{\text{targetMetricValue}} \right\rceil$.

When multiple metrics are configured simultaneously, the controller evaluates each metric independently and uses the maximum of the resulting desired replica counts, so that the workload is not scaled below the demand implied by any configured signal [12]. The computed value is then constrained by the replica bounds defined in the HPA manifest: `spec.minReplicas`, which defaults to 1 when omitted, and the mandatory `spec.maxReplicas` field [11].

The HPA depends on Kubernetes' metrics pipeline. CPU and memory metrics are provided through the Resource Metrics API, typically by the Metrics Server; custom object and pod metrics are exposed through the Custom Metrics API; and metrics produced outside the cluster, such as queue depth or request rate in an external system, are exposed through the External Metrics API [11, 12]. The availability of these APIs in a cluster determines which HPA metric types can be evaluated at runtime.

An important operational subtlety is that the `apiVersion` declared in a manifest determines which fields are part of the schema, but it does not guarantee that all configured metrics can actually be retrieved in the target cluster. For example, an HPA using an `External` metric requires an external metrics adapter; without the corresponding metrics pipeline, the manifest may be syntactically valid while the autoscaler is unable to compute the intended scaling decision.

2.2 Evolution of the HPA API

The HPA API has evolved from a CPU-oriented autoscaler into a richer control surface for multi-signal and behavior-aware scaling. The feature model shown in Figure 1 summarizes the cumulative configuration surface of `autoscaling/v2`, the current General Availability (GA) version. In the diagram, solid edges denote mandatory fields, circle endpoints denote optional fields, triangles denote alternatives, and colors indicate when fields entered the API surface: white for v1, orange for v2beta1, green for v2beta2, and blue for fields unified or stabilized in v2 GA.

autoscaling/v1. Introduced in Kubernetes 1.2, `autoscaling/v1` exposes the original HPA interface: `scaleTargetRef`, `minReplicas`, `maxReplicas`, and the CPU-oriented field `targetCPUUtilizationPercentage`. It has no `spec.metrics` array, no support for custom or external metric specifications, and no `spec.behavior`. We therefore treat `autoscaling/v1` as a stable but capability-limited API surface rather than as a removed API. The Kubernetes deprecation guide instead identifies the removal of the older beta HPA APIs: `autoscaling/v2beta1` stopped being served in Kubernetes 1.25, and `autoscaling/v2beta2` stopped being served in Kubernetes 1.26, with migration to `autoscaling/v2` recommended [10].

autoscaling/v2beta1. Introduced after `autoscaling/v1`, `autoscaling/v2beta1` replaces the single CPU threshold with a metrics array. This enables HPAs to use multiple metric specifications, including resource metrics, object metrics, pod metrics, external metrics, and container-level resource metrics. The main change is therefore the transition from a fixed CPU-specific field to an extensible metric model. However, `v2beta1` still lacks `spec.behavior`, so it does not provide explicit control over scale-up and scale-down stabilization or rate policies.

autoscaling/v2beta2 and v2 GA. The `autoscaling/v2beta2` API adds the `spec.behavior` block, allowing independent scale-up and scale-down configuration through stabilization windows, selection policies, and scaling-rate policies [12]. It also introduces a more consistent `MetricTarget` structure with explicit target

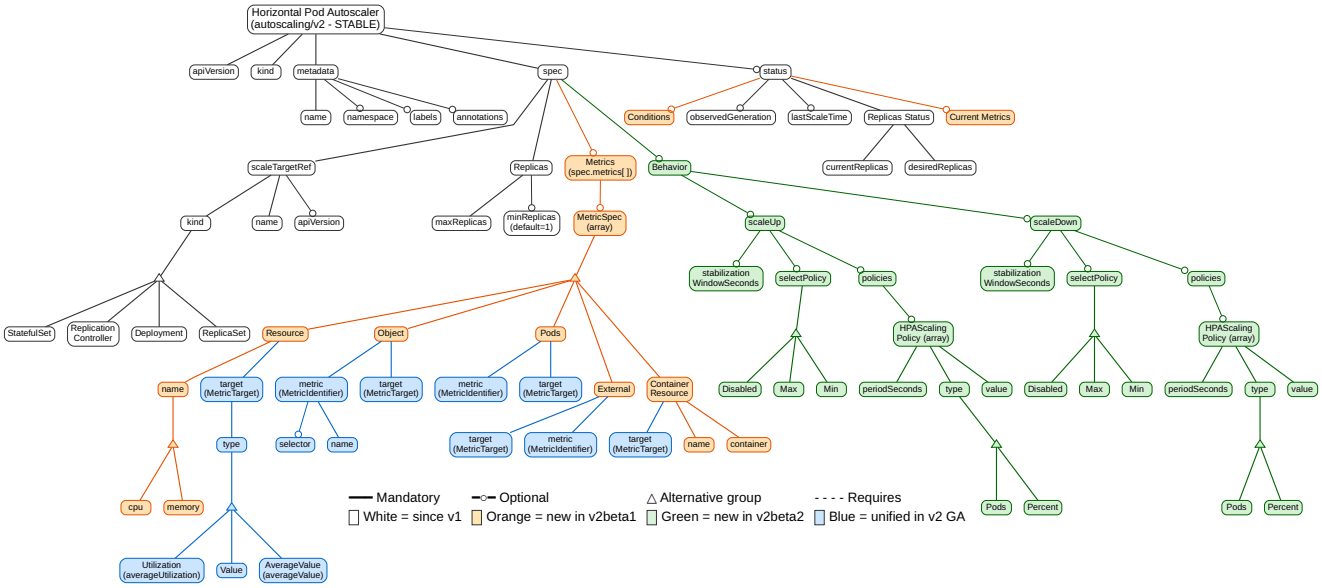

Figure 1: Feature model of the `autoscaling/v2` GA configuration surface.

types: `Utilization`, `AverageValue`, and `Value`. Kubernetes 1.23 promoted `autoscaling/v2` to GA, consolidating the v2beta2 capabilities into a stable API and unifying metric identifier semantics. No major optional capability was added at GA; the relevant change was the stabilization of the API surface and the subsequent deprecation and removal of the older beta versions [10].

3 Related Work

Research on cloud and Kubernetes autoscaling has largely focused on designing or evaluating scaling mechanisms. Prior studies have proposed predictive models, trend-detection strategies, adaptive controllers, and HPA tuning approaches to improve scaling decisions under changing workloads [1, 3, 15–17, 22, 24, 25]. These works provide important evidence about how autoscaling policies behave under controlled experimental conditions and how Kubernetes' default mechanisms can be extended or tuned. However, their emphasis is primarily algorithmic or systems-oriented: they evaluate proposed controllers, workload predictors, or parameter settings in specific testbeds. In contrast, our study examines how the built-in Kubernetes HPA is configured in public open-source repositories, focusing not on proposing a new autoscaling strategy but on characterizing the field-level use of the existing HPA API at scale.

A second line of work treats container and infrastructure configuration artifacts as empirical software engineering objects. Cito et al. [5] characterized the Docker ecosystem on GitHub, while studies of infrastructure and configuration code have investigated security and configuration smells and proposed metrics for assessing configuration quality [7, 18, 21]. More recent work has focused specifically on Kubernetes artifacts, examining security defects [4] and community-prescribed security practices in manifests [20], smells in manifests generated by large language models [27], and the evolution and security risks of Helm charts [26]. Together, these studies show that declarative infrastructure artifacts encode operational decisions and may accumulate smells, defects, outdated dependencies, and risky configurations. Our study follows this empirical orientation but focuses on a different unit of analysis: rather than assessing container or Kubernetes artifacts broadly, it examines the field-level use, evolution, maintenance, and recurring configuration smells of a single Kubernetes API, the HPA, across open-source repositories.

Finally, mining software repositories (MSR) research provides methodological guidance for constructing and interpreting large-scale corpora from public repositories [9, 23]. We build on these methodological recommendations by combining sharded repository discovery, structured YAML extraction, repository metadata, per-file commit history, and sensitivity analyses over corpus subsets. The main distinction of this study is that it connects MSR methods with the evolution of a concrete cloud-native API, using HPA manifests to study the gap between API availability, declared API adoption, and actual capability utilization in practice.

4 Methodology

4.1 Research Questions

Our study is guided by the following research questions:

- **RQ1: Evolution of APIs and Capabilities.** How has the adoption of different HPA API versions and their specific capabilities evolved over time in open-source projects?
- **RQ2: Feature Richness.** What levels and patterns of feature richness characterize HPA configurations found in practice?
- **RQ3: Maintenance and Co-evolution.** How frequently are HPA configurations maintained, and how does feature richness relate to HPA maintenance and repository activity?

RQ4: Configuration Smells. What outdated, inconsistent, underutilized, or potentially risky HPA configuration smells can be observed at scale?

RQ1 examines the temporal uptake of HPA API versions and field-level capabilities. We measure *adoption lag* as the elapsed time between each version's official Kubernetes release and the first observed commit using that version in a repository. This measure captures when an API version appears in the mined repositories; it does not imply that we observe explicit file-by-file migration unless such transitions are visible in the commit history. We also use repository creation cohorts and per-file commit years to analyze whether newer projects start with newer API versions and whether capabilities such as multi-metric scaling, non-CPU metrics, and `spec.behavior` become more common over time.

RQ2 measures how much of the HPA configuration surface is used by each manifest. We address this question through the HPA Feature Richness Score (HFRS), defined in Section 4.4, and through a deterministic archetype taxonomy derived from the 13 boolean HFRS components. The archetypes are assigned by explicit rules over the component profile rather than inferred by unsupervised clustering.

RQ3 distinguishes HPA-file maintenance from general repository activity. A repository is considered HPA-active when at least one HPA file was touched within two years of the pipeline date, HPA-outdated when no HPA file was touched for more than three years, and HPA-intermediate otherwise. We also identify stale HPA files inside otherwise HPA-active repositories, capturing cases in which some autoscaling manifests remain unchanged while other HPA files in the same repository continue to evolve. Moreover, we use a stratified commit-history sample to examine whether HPA changes tend to occur together with broader repository changes or as isolated tuning events.

Finally, **RQ4** catalogs six statically detectable HPA configuration smells derived from the HPA API specification, deprecation history, and HFRS component structure. We use the term configuration smell analogously to code and infrastructure smells: a recurring configuration structure that may indicate legacy usage, underutilization, inconsistency, or operational risk, but that is not necessarily a defect. These smells are descriptive rather than definitive defects: their presence signals that a manifest may deserve inspection, but workload-specific or deployment-specific context may justify a flagged configuration.

4.2 Data Collection and Processing

The data collection and processing pipeline comprises eight phases.

Phase 1: Candidate discovery. We used GitHub Code Search to discover candidate HPA files. Because GitHub Code Search limits each query to 1,000 results, we used 46 sharded queries that partitioned the search space along seven dimensions: explicit `kind: HorizontalPodAutoscaler` declarations, API-version strings, v2-specific field names such as `behavior`, `scaleDown`, and `stabilizationWindowSeconds`, common HPA file names, deployment directory patterns, repository-creation ranges, and project-quality signals. Results were deduplicated by repository full name and path, yielding 10,972 unique candidate files.

Phase 2: Repository metadata collection. For the repositories discovered in the previous phase, we collected GitHub metadata through batched GraphQL requests, including creation date, last push date, number of stars, fork status, and description. All intermediate results were persisted as CSV artifacts, allowing the pipeline to resume from any phase without repeating previous GitHub API calls. The full collection required approximately 27,624 GitHub API calls, mostly for code search, raw manifest retrieval, per-file commit dates, and the sampled commit-history analysis.

Phase 3: Manifest retrieval and validation. Candidate files were downloaded through the GitHub Contents API and parsed as YAML. A file was retained only if it contained a top-level `HorizontalPodAutoscaler` kind declaration and an `apiVersion` beginning with `autoscaling/`. Helm templates were excluded using both path-based and content-based rules: files under chart/template paths and files containing Go-template delimiters such as `{{...}}` were removed because templated values such as `minReplicas: {{ .Values.hpa.min }}` cannot be reliably parsed as numeric fields. This step removed 1,208 files, resulting in a main corpus of 9,764 valid non-Helm HPA manifests.

Phase 4: Maintenance metadata collection. For each retained HPA file, we retrieved the most recent commit touching that path using GitHub's path-filtered commit endpoint with `per_page=1`. This setting is sufficient because this phase requires only the latest commit touching each HPA file; complete histories are collected separately for the co-evolution analysis. We use this per-file commit date as the main temporal signal for HPA maintenance because repository-level timestamps such as `pushed_at` may reflect unrelated changes and can therefore misrepresent the age of configuration files [6, 9].

Phase 5: Manifest normalization and feature extraction. The parsed manifests were normalized into a metric-level dataset containing one row per HPA metric entry. Extracted fields include API version, replica bounds, metric type and name, target kind and value, `spec.behavior` configuration, and the kind of workload referenced by `scaleTargetRef`. Because a single HPA file may declare multiple metrics, the metric-level dataset contains 16,506 metric definitions, which are later aggregated back to the file level for HFRS computation.

Phase 6: Analytical dataset construction. The main analysis uses four levels of granularity. At the broadest level, repository metadata covers 12,741 GitHub repositories discovered during the search and metadata-collection process. The repository-level analytical subset contains 7,013 repositories with at least one retained HPA file. The file-level corpus contains 9,764 HPA manifests and is the primary unit for API-version, HFRS, archetype, and risk-indicator analyses. For behavior-aware files, we separately extract `scaleUp` and `scaleDown` settings, including `stabilizationWindowSeconds`, `selectPolicy`, and `policies`, because these fields determine the Control Richness components of HFRS. The metric-level dataset contains 16,506 metric definitions and supports analyses of signal types and target values. A separate commit-level dataset supports the co-evolution analysis in **RQ3**.

Phase 7: Co-evolution sample selection. To avoid collecting complete commit histories for all repositories, the co-evolution analysis uses a proportional stratified sample of 169 repositories by HPA-maintenance status, selected with a fixed random seed. The realized sample comprises 70 HPA-active, 50 HPA-outdated, and 49

HPA-intermediate repositories and is used to inspect co-evolution patterns, not to estimate corpus-level prevalence.

Phase 8: Commit-history and co-evolution analysis. For each sampled repository, we collected repository-level commits within a bounded lookback window and HPA-file-specific commits using path filters. This produced 261 HPA-file commit events and 8,665 repository-level events. For each HPA-file commit event e_{HPA} in repository r, we compute the minimum absolute time difference to any repository-level commit event e_{repo} in the same repository, as $d(e_{\text{HPA}}) = \min_{e_{\text{repo}} \in r} \left| t(e_{\text{HPA}}) - t(e_{\text{repo}}) \right|$. A distance of zero means that the HPA file was changed in the same commit as other repository files; larger values indicate more temporally isolated HPA modifications.

Most repositories contribute a small number of HPA files: among the 7,013 repositories with HPA data, 85.5% contribute exactly one HPA manifest, with a median of one and a maximum of 17 files in a single monorepo. At the metric level, 59.6% of files contain a single metric entry, 31.0% contain two entries, and 9.4% contain three or more; the mean is 1.71 metric entries per file.

4.3 Corpus Filtering and Sensitivity Variables

The main corpus contains all 9,764 valid non-Helm HPA files. We do not apply a recency filter to construct the main corpus because excluding older repositories would remove precisely the kind of stale or weakly maintained configuration artifacts that are relevant to the study. Instead, recency is treated as a sensitivity variable: analyses can be repeated on subsets defined by recent repository activity or recent HPA-file maintenance to assess whether the main findings are driven by outdated repositories.

We also retain tutorial, example, and practice-oriented repositories rather than discarding them. A repository is flagged as a toy project when its `owner/project` name contains keywords associated with educational, example, test, or practice contexts, such as `tutorial`, `course`, `workshop`, `demo`, `sample`, `example`, `template`, `playground`, `sandbox`, `test`, `practice`, or `exercise`. Of the 7,013 repositories with at least one retained HPA file, 966 (13.77%) are flagged by this heuristic. The flag is only a contextual indicator and may produce both false positives and false negatives; it is not treated as a definitive project classification. We keep these repositories in the main corpus because tutorial and prototype configurations are part of the public configuration ecosystem and may influence reuse, but we preserve the `is_toy` flag for sensitivity analysis [2].

Following prior MSR work on repository quality and development practices [28], we also compute a repository-level confidence score from eight binary signals: recent repository push activity, repository lifespan greater than six months, presence of at least one valid HPA file, presence of at least one multi-metric HPA, presence of at least one HPA with `spec.behavior`, use of `autoscaling/v2` or later, absence of toy-project keywords, and at least one GitHub star. The score is not used to exclude repositories from the main corpus; it is used only to support robustness checks over higher-confidence subsets.

4.4 HPA Feature Richness Score

To compare HPA configurations across API versions, we define the HPA Feature Richness Score (HFRS). HFRS is a structural measure of configuration richness, not a correctness or quality metric. A low HFRS may be adequate for a simple workload, and a high HFRS may still be poorly calibrated for its deployment context. The score captures how much of the HPA configuration surface is explicitly used by a manifest:

$$\text{HFRS}(f) = \text{SR}(f) + \text{CR}(f) + \text{AR}(f), \qquad \text{HFRS}(f) \in [0, 13]. \quad (1)$$

Signal Richness (SR, 0–5) captures the diversity and sophistication of scaling signals. SR1 is awarded when the file uses a v2-family API and therefore has access to `spec.metrics[]`; SR2 when two or more metrics are configured; SR3 when at least one non-CPU metric is used; SR4 when at least one `External`, `Pods`, or `Object` metric is present; and SR5 when two or more distinct metric types are used. *Control Richness* (CR, 0–4) captures explicit scaling-behavior configuration. CR1 is awarded when `spec.behavior` is present and non-empty; CR2 when `scaleUp` is explicitly configured; CR3 when `scaleDown` is explicitly configured; and CR4 when both directions are configured. *Adjustment Richness* (AR, 0–4) captures explicit bounds and target-parameter choices. AR1 is awarded when `minReplicas` is explicitly set; AR2 when `maxReplicas` is present; AR3 when at least one numeric target value is configured; and AR4 when the manifest uses heterogeneous target values or target kinds across metrics.

HFRS intentionally assigns one point to each component. Equal weighting avoids unsupported assumptions about the relative operational importance of different fields: the score measures breadth of explicit API use, not correctness, quality, or runtime effectiveness. Some components are intentionally cumulative—for example, a file with multiple metrics may also receive points for non-CPU metrics and metric-type diversity—because each point records a distinct structural property of the manifest. Validation against expert judgment or runtime outcomes is outside the scope of this study.

Files are classified into five HFRS classes: Minimal (0–2), Basic (3–5), Moderate (6–8), Rich (9–11), and Highly Rich (12–13). These boundaries divide the 0–13 range into approximately equal-width descriptive intervals; they are interpretive categories, not empirically validated quality thresholds. The `maxReplicas` component has limited discriminatory power because the field is mandatory in valid HPA manifests; we retain it to keep malformed or partially extracted manifests explicit, but the interpretation of HFRS relies primarily on the signal and control dimensions and on the remaining adjustment components. This classification allows us to distinguish manifests that merely declare a newer `apiVersion` from those that actually use the richer scaling signals and behavior controls introduced by newer HPA API versions.

4.5 Archetypes and Configuration Smells

In addition to the continuous HFRS score, we classify each HPA file into one of five mutually exclusive archetypes using deterministic rules over the HFRS component vector. The rules prioritize features that qualitatively change the configuration profile: legacy API usage, use of non-resource or external signals, explicit behavior configuration, and dual-resource scaling. The resulting archetypes are: *Legacy minimal*, for `autoscaling/v1` CPU-oriented configurations; *Basic migration*, for v2-family files that retain a single CPU metric and no

Table 1: HPA file distribution by API version

apiVersion	n	%	HFRS
autoscaling/v2 GA	6,478	66.35%	7.05
autoscaling/v1	1,702	17.43%	3.01
autoscaling/v2beta2	961	9.84%	5.16
autoscaling/v2beta1	623	6.38%	3.66
Total	9,764	100.00%	5.94

behavior configuration; *Dual-resource*, for files using CPU and memory without advanced behavior or external signals; *Behavior-aware*, for files configuring `spec.behavior`; and *Signal-rich*, for files using `External`, `Pods`, or `Object` metrics. These archetypes are intended as descriptive profiles, not as quality labels. They represent the dominant combinations of API generation, signal diversity, and behavior configuration observed in the corpus. The rules are applied in a fixed priority order, so each file receives exactly one label even when it exhibits properties associated with multiple profiles; the taxonomy is descriptive rather than exhaustive.

For **RQ4**, we define six HPA configuration smells that can be detected statically from the manifest and metadata: legacy `autoscaling/v1` usage in HPA-active repositories; structurally minimal v2-family configurations; stale HPA files inside otherwise HPA-active repositories; fixed-replica HPA declarations where `minReplicas` equals `maxReplicas`; unsupported `spec.behavior` fields under API versions that do not define them; and unusually low CPU utilization targets. These smells are deliberately conservative: they identify patterns that may deserve review, while acknowledging that some configurations may be intentional or appropriate in context. The smells are not mutually exclusive, and a file may satisfy multiple rules; per-file assignments in the replication package support intersection analyses. The $\mathrm{HFRS} \leq 4$ threshold used for structurally minimal v2-family files captures configurations that generally retain a single CPU metric, no explicit behavior, and only basic bounds and target fields. The 10% CPU cutoff is a conservative inspection threshold for unusually low targets, not a universal recommendation.

5 Results

5.1 RQ1: Evolution of APIs and Capabilities

Table 1 summarizes the distribution of HPA files by API version and their mean HFRS. The current stable `autoscaling/v2` API dominates the corpus, accounting for 66.35% of the 9,764 HPA files. In contrast, `autoscaling/v1` still appears in 17.43% of files, despite exposing only the original CPU-oriented configuration surface and lacking the metric and behavior capabilities introduced in the v2-family APIs. Of the 1,702 v1 files, 1,199 belong to repositories created in 2022 or later, after `autoscaling/v2` became GA. These files should not be interpreted as evidence of projects migrating from v2 back to v1; rather, they indicate that many newer repositories still started from the older CPU-oriented API surface. This pattern is consistent with reuse of older examples, tutorials, or templates, although repository intent cannot be inferred from manifests alone.

To characterize the temporal uptake of newer API versions, we computed adoption lag as the elapsed time between each version's official Kubernetes release and the first observed HPA-file commit using that version in a repository. Table 2 shows that median adoption lag ranges from 3.8 to 4.1 years across `v2beta1`, `v2beta2`, and `v2` GA. Fewer than 1.4% of files using any of these versions appear within the first year after the corresponding release. Because the analysis observes the first appearance of each version in the mined repositories, these values should be interpreted as adoption lag rather than direct evidence of file-level migration. The consistency of the lag across versions suggests that HPA API uptake is slow and largely driven by when projects are created or first add HPA manifests, rather than by rapid upgrades of existing configurations.

Table 2: Adoption lag by API version

Version	n	Median lag	≤1 yr
autoscaling/v2beta1	623	1,501 d (4.1 yr)	0.96%
autoscaling/v2beta2	961	1,478 d (4.0 yr)	1.35%
autoscaling/v2 GA	6,478	1,386 d (3.8 yr)	1.27%

Table 3: Distribution of v2 GA and v1 files by creation year

Cohort	v2 GA %	v1 %
2018	12.43%	45.56%
2019	8.63%	44.31%
2020	6.81%	32.97%
2021	13.83%	23.25%
2022	25.38%	32.32%
2023	49.18%	31.46%
2024	52.34%	28.48%
2025	93.15%	5.12%

The cohort analysis in Table 3 reinforces this interpretation. The table focuses on the two dominant stable versions; the beta versions are omitted because their cohort counts are comparatively small and their complete distribution is reported in Table 1. Repositories created in 2025 use `autoscaling/v2` GA in 93.15% of their HPA files, indicating that new projects increasingly start from the stable v2 API. The transition is particularly sharp between the 2024 and 2025 cohorts, where v2 GA usage rises from 52.34% to 93.15%. However, legacy v1 remains visible even in recent cohorts: 5.12% of HPA files in repositories created in 2025 still use `autoscaling/v1`. The persistence of v1 in repositories created after v2 GA became available suggests that API stabilization alone does not immediately eliminate older configuration patterns from public repositories.

Capability adoption shows a more nuanced picture than API-version adoption alone. Figure 2 shows that resource and memory signals become increasingly common over time, whereas `External`, `Pods`, and `Object` metrics remain comparatively rare. Similarly, `spec.behavior` adoption grows mainly in the v2 GA period, indicating that explicit control over scale-up and scale-down behavior is still far from universal. Figure 3 complements this view by showing the replacement of older API versions over time and the gradual increase in behavior-aware configurations.

Answer to RQ1: The HPA ecosystem has largely converged toward `autoscaling/v2` at the API level, but this convergence does not imply full use of the v2 capability surface. The stable v2 API

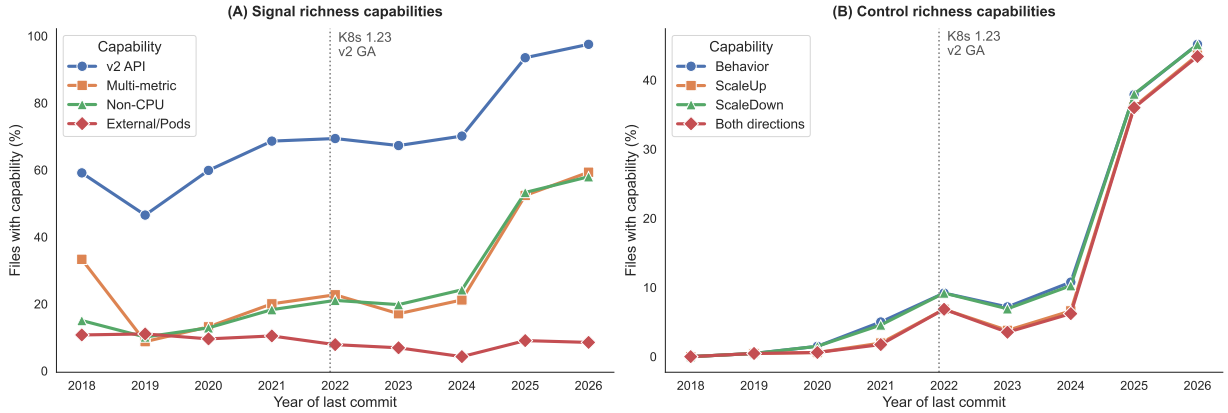

Figure 2: Capability adoption over time by year of last HPA-file commit. Panel A shows signal-richness capabilities, including non-CPU and external/Pods signals. Panel B shows control-richness capabilities associated with `spec.behavior`.

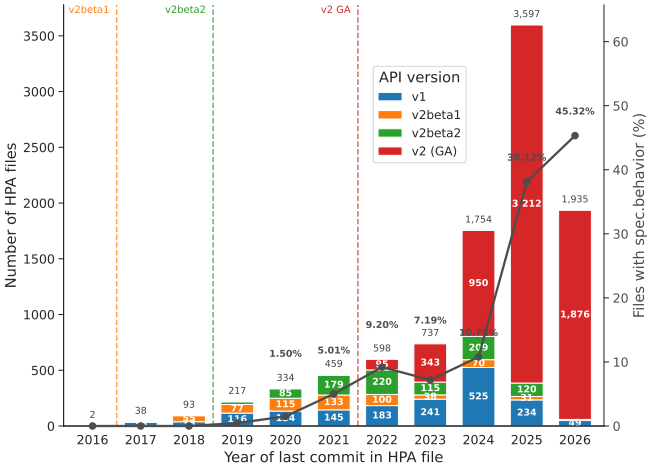

Figure 3: Temporal evolution of HPA API versions by year of last file commit. The black line shows the percentage of files with `spec.behavior` configured.

accounts for 66.35% of the corpus and reaches 93.15% among repositories created in 2025, yet adoption lag remains long across API generations and legacy `autoscaling/v1` persists even in recent cohorts. Capability adoption is also uneven: memory and resource-based signals grow over time, whereas external/application-level signals and explicit behavior configuration remain comparatively less common.

5.2 RQ2: Feature Richness

Table 4 presents the HFRS distribution across the 9,764 HPA files. The distribution is concentrated in the lower half of the scale: 55.65% of files fall in the *Basic* class, while only 20.96% reach the *Rich* or *Highly Rich* classes. The median HFRS is 4 out of 13. The subdimension means help explain this concentration: mean SR is 1.75/5, mean CR is 1.00/4, and mean AR is 3.19/4. Adjustment Richness is high because most valid HPA files define replica bounds and at least

Table 4: HFRS distribution by Richness Class

Class	n	%	HFRS range
Minimal	142	1.45%	0–2
Basic	5,434	55.65%	3–5
Moderate	2,142	21.94%	6–8
Rich	1,678	17.19%	9–11
Highly Rich	368	3.77%	12–13
Total	9,764	100.00%	mean=5.94, median=4

Table 5: Distribution of Configuration Archetypes

Archetype	n	%	HFRS mean
1. Legacy minimal	1,668	17.08%	2.96
2. Basic v2 adoption	3,592	36.79%	3.88
3. Dual-resource	1,603	16.42%	6.46
4. Behavior-aware	2,121	21.72%	9.97
5. Signal-rich	780	7.99%	9.77
Total	9,764	100.00%	

one target value. Control Richness is much lower because most files, including many v2 GA files, do not configure `spec.behavior`. Thus, a common pattern is not the absence of HPA configuration altogether, but the presence of structurally simple manifests that rely on default scaling behavior.

Table 5 reports the five rule-based archetypes derived from the HFRS component profile. The largest group is *Basic v2 adoption*, representing 36.79% of files. These manifests declare a v2-family API but retain a single CPU metric and no explicit behavior configuration, making their effective scaling logic close to the legacy v1 profile. The *Legacy minimal* archetype accounts for 17.08% of files and corresponds to CPU-oriented `autoscaling/v1` manifests. Together, these two archetypes show that more than half of the corpus is dominated by minimal CPU-based scaling patterns.

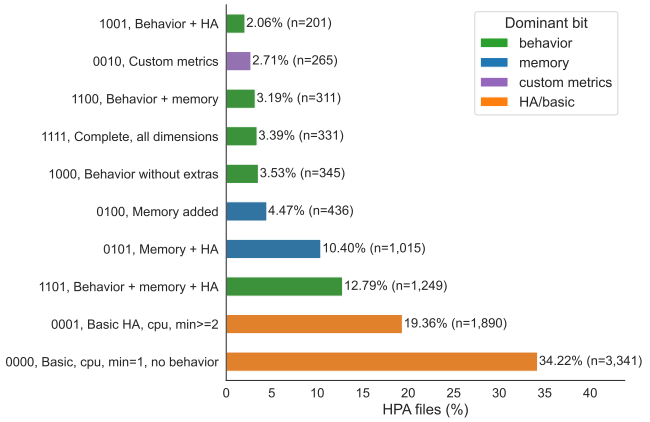

Figure 4: Most frequent configuration profiles in the corpus.

The remaining archetypes represent progressively richer forms of HPA usage. *Dual-resource* files configure CPU and memory metrics but do not use external signals or behavior policies. *Behavior-aware* files configure `spec.behavior` and therefore use the control surface introduced in the v2-family APIs to shape scale-up or scale-down behavior. *Signal-rich* files use at least one `External`, `Pods`, or `Object` metric and therefore depend on a richer metrics pipeline. Although *Behavior-aware* and *Signal-rich* files have similar mean HFRS values, they represent different forms of sophistication: the former emphasizes control over scaling dynamics, while the latter emphasizes integration with non-resource or application-specific signals.

Figure 4 provides a finer-grained view of these archetypes and leads to the same conclusion: the most frequent profile, corresponding to CPU-only scaling with `minReplicas=1` and no `spec.behavior`, accounts for 34.22% of all HPA files. The next most frequent profile, CPU-only scaling with `minReplicas`≥2 and no behavior configuration, accounts for another 19.36%. Thus, more than half of the corpus is concentrated in two CPU-only profiles with no explicit behavior control.

Answer to RQ2: HPA adoption in public repositories is widespread but often shallow. The median HFRS is only 4 out of 13, and 55.65% of files fall in the *Basic* class. The dominant archetype, *Basic v2 adoption*, shows that many manifests declare a v2-family API while retaining a single CPU metric and no explicit behavior configuration. Thus, the main pattern is not absence of autoscaling, but limited use of the richer signal and control capabilities available in newer HPA APIs.

5.3 RQ3: Maintenance and Co-evolution

We classify the 7,013 repositories with at least one valid HPA file according to HPA-file maintenance. A repository is HPA-active when at least one HPA file was touched within two years of the pipeline date, HPA-intermediate when the latest HPA-file commit is between two and three years old, and HPA-outdated when no HPA file was touched for more than three years. Under this definition, 4,955 repositories (70.65%) are HPA-active, 702 (10.01%) are HPA-intermediate, and 1,356 (19.34%) are HPA-outdated. This

Table 6: HFRS by HPA maintenance Status

Status	n files	HFRS mean	HFRS median
Recent HPA	7,021	6.66	6
Intermediate	910	4.39	4
Stale HPA	1,833	3.96	3

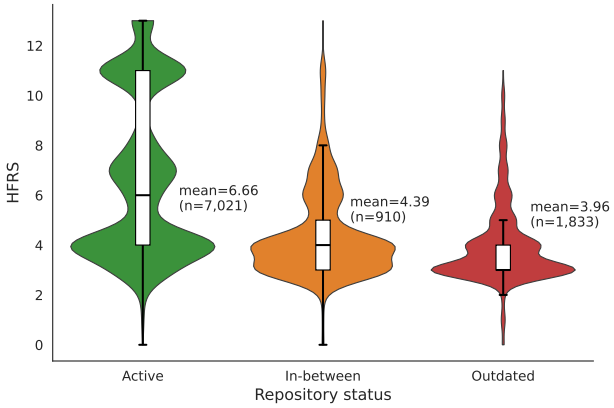

Figure 5: HFRS distribution by HPA maintenance status.

status captures maintenance of HPA material specifically, not general repository activity. The *Behavior-aware* archetype appears in 38.26% of HPA-active repositories but in only 4.28% of HPA-outdated repositories, suggesting a strong association between recent HPA maintenance and use of the richer behavior-control surface.

The maintenance-status counts above are repository-level; Table 6 and Figure 5 report the HPA files contained in repositories assigned to each category. Table 6 and Figure 5 show that HPA-active repositories have substantially richer HPA configurations than HPA-outdated repositories. The mean HFRS is 6.66 for HPA-active repositories and 3.96 for HPA-outdated repositories. The difference is statistically significant under a Mann-Whitney test ($U = 9{,}941{,}384$, $p < 0.001$, $r = -0.545$). Within HPA-active repositories, file age is also negatively associated with richness: the Spearman correlation between `hpa_file_age_years` and HFRS is $\rho = -0.381$ ($p < 0.001$). Among v2 GA files in HPA-active repositories, the association remains visible ($\rho = -0.329$, $p < 0.001$), indicating that the effect is not only a consequence of older files being written before newer API capabilities existed.

We also identify 56 frozen HPA files, corresponding to 0.57% of the corpus. Under our definition, a frozen HPA is a file older than two years that belongs to a repository where at least one other HPA file was touched within the two-year HPA-activity window. This is intentionally narrower than a generic stale-file definition based on repository `pushed_at`; it captures stale HPA material inside repositories that still maintain other HPA manifests. Frozen files have a mean HFRS of 3.52, compared with 6.69 for non-frozen files in the same HPA-active repositories. They are concentrated in the *Legacy minimal* and *Basic v2 adoption* archetypes, and none falls in the *Rich* or *Highly Rich* classes.

Table 7: HPA configuration-smell catalogue (S1–S6)

ID	Pattern	n	%
S1	legacy v1 in HPA-active repository	728	7.46%
S2	v2 structurally minimal (HFRS $\leq$ 4)	3,480	35.64%
S3	Frozen HPA in HPA-active repo	56	0.57%
S4	minReplicas = maxReplicas (inoperative)	445	4.56%
S5	`spec.behavior` in invalid apiVersion	14	0.14%
S6	cpu.averageUtilization < 10%	140	1.43%

The commit co-evolution sample provides complementary evidence about how HPA files change. Of the 241 HPA–repository event pairs analyzed, 157 (65.15%) have temporal distance zero, meaning the HPA file was modified in the same commit as broader repository changes. The remaining 84 pairs have distance greater than zero; among these, 16 cases are isolated HPA modifications with distance greater than one day and a median distance of 269 days. This distribution suggests two distinct maintenance patterns: HPA configuration is often adjusted together with broader infrastructure or application changes, while standalone HPA tuning appears comparatively infrequent.

Answer to RQ3: HPA configuration richness is strongly associated with HPA-file maintenance. HPA-active repositories contain richer configurations than HPA-outdated repositories, and older HPA files tend to have lower HFRS even within HPA-active repositories. Frozen HPA files are concentrated in the simplest archetypes, while the co-evolution sample shows that HPA changes usually occur together with broader repository changes and that isolated HPA tuning is comparatively rare. These results indicate that richer HPA usage depends not only on API availability, but also on whether autoscaling manifests continue to evolve after their initial introduction.

5.4 RQ4: Configuration Smells

Table 7 summarizes the six statically detectable HPA configuration smells analyzed in **RQ4**. These smells should be interpreted as inspection signals rather than definitive defects. They identify configurations whose operational adequacy depends on context, but whose structure suggests possible legacy usage, underutilization, inconsistency, or ineffective scaling behavior.

S1 identifies 728 files (7.46%) using legacy `autoscaling/v1` in HPA-active repositories. We do not treat v1 as a deployment-breaking removed API; according to the Kubernetes deprecation guide, the removed HPA APIs are the older beta versions, while `autoscaling/v1` remains a stable but capability-limited surface [10]. S1 therefore marks files that may deserve modernization because they cannot express multi-metric scaling or `spec.behavior`. S2 is the most prevalent indicator, affecting 3,480 files (35.64%). It captures v2-family manifests with HFRS $\leq$ 4, i.e., files that declare a newer API version but remain structurally minimal: no non-CPU signal, no multiple metrics, no custom or external metric source, and no behavior configuration. Together, S1 and S2 affect 4,208 files, or approximately 43% of the corpus, making the API adoption/utilization gap the most visible pattern in the study.

S3 corresponds to the 56 frozen HPA files described in **RQ3**. S4 identifies 445 files (4.56%) where `minReplicas` equals `maxReplicas`. Among these 445 files, 66.97% set both bounds to 1, meaning that the declared HPA can never scale the workload beyond a single replica. In such cases, the HPA controller may still compute a desired replica count, but the result is always clamped to a single allowed value, making the autoscaler functionally inert. This pattern may be intentional in templates or staged deployments, but it provides no autoscaling benefit while retaining an HPA declaration. S5 identifies 14 files (0.14%) that include `spec.behavior` under API versions that do not define that field, namely `autoscaling/v1` or `autoscaling/v2beta1`. Depending on Kubernetes version and server-side field-validation mode, the unsupported field may be rejected, warned about, or ignored; in all cases, the intended behavior policy is not part of the declared API surface [13]. Finally, S6 identifies unusually low CPU utilization targets: 143 CPU metric entries, corresponding to 140 unique HPA files, set `averageUtilization` below 10%. The value 5% is the most common among these low thresholds. According to the formula presented in Subsection 2.1, very low CPU targets amplify the ratio between observed and target utilization, causing aggressive scale-out when pods exceed a small fraction of their requested CPU. The affected entries are spread across 128 unique repository owners, suggesting that this pattern is not confined to a single copied source or project family.

Answer to RQ4: The most prevalent HPA configuration smells are widespread forms of limited capability use rather than rare syntactic anomalies. Legacy `autoscaling/v1` usage and structurally minimal v2-family configurations together affect approximately 43% of the corpus, making the adoption/utilization gap the dominant smell category. Less frequent smells, such as fixed-replica HPAs, unsupported `spec.behavior` fields, frozen HPA files, and unusually low CPU thresholds, are more directly actionable because they can be detected statically and surfaced by linters, policy engines, or CI checks.

6 Discussion

Our results reveal a persistent gap between HPA API adoption and capability utilization. Although `autoscaling/v2` is now the dominant API version in the corpus, a large fraction of manifests remain structurally close to the original CPU-based `autoscaling/v1` model. This suggests that API stabilization alone is insufficient to ensure uptake of richer configuration capabilities. For practitioners, the main implication is that HPA modernization should not be limited to changing the `apiVersion` field. Migration to a v2-family API should also trigger a review of scaling signals, behavior policies, replica bounds, and target values. In particular, teams should check whether workloads would benefit from memory or application-level metrics, whether scale-up and scale-down behavior should be explicitly controlled, and whether existing HPA manifests have become stale relative to the current workload.

The configuration smells identified in **RQ4** are directly actionable for tool builders and platform teams. Several smells can be detected statically without cluster access, including legacy `autoscaling/v1` usage, structurally minimal v2-family manifests, fixed-replica HPAs, unsupported `spec.behavior` fields, and unusually low CPU utilization targets. These checks could be incorporated

into Kubernetes linters, policy engines, CI/CD pipelines, and repository-level audits. Importantly, these smells should not be treated as universal defects. For example, `minReplicas = maxReplicas` may be intentional in a staged deployment, and a low HFRS may be sufficient for a simple service. The practical value of the smell catalogue lies in surfacing configurations that deserve review, not in automatically rejecting them.

From an API-evolution perspective, the findings show that version uptake and capability uptake are distinct phenomena [14]. A repository may migrate syntactically to `autoscaling/v2` while preserving the field-level behavior of the older interface. This distinction can inform documentation and migration guidance by revealing which capabilities remain uncommon after becoming stable.

For researchers, the study highlights HPA manifests as a useful empirical lens for studying the evolution of cloud-native configuration APIs. Prior work has often focused either on improving autoscaling algorithms or on detecting generic Kubernetes configuration issues. The next empirical step is to test whether structural richness is associated with runtime outcomes, including SLO violations, scaling instability, and over-provisioning. Longitudinal studies can also determine whether HPA configurations become richer as projects mature or remain close to the form in which they were introduced.

7 Threats to Validity

The study has several threats to validity. Internal validity is affected by the syntactic nature of the extraction pipeline. YAML parsing errors, non-standard repository layouts, generated manifests, or templating constructs may affect field detection. We mitigate this threat by excluding Helm templates, using structured parsing rules, separating file-level and metric-level analyses, and preserving the extraction scripts and intermediate CSV files in the replication package. Nevertheless, manifests that require rendering, composition, or deployment-specific values may not be fully represented by static analysis.

Construct validity concerns the interpretation of HFRS and the configuration-smell indicators. Its equal weighting and class boundaries are design choices for a transparent structural index rather than empirically validated assessments of importance or quality. Likewise, the low-CPU threshold is an inspection rule whose operational meaning depends on workload context. HFRS measures configuration richness, not correctness, performance, or operational maturity. A low score may be appropriate for a simple workload, while a high score may still encode a poorly calibrated or ineffective autoscaling policy. Similarly, the **RQ4** configuration smells identify configurations that may deserve inspection, not definitive faults. This distinction is especially important for public repositories, where manifests may serve different purposes: production deployment, examples, tutorials, tests, or documentation. We therefore avoid interpreting HFRS as a quality score and treat the configuration-smell catalogue as a set of conservative inspection signals.

External validity is limited by the use of public GitHub repositories. The corpus may overrepresent tutorials, examples, prototypes, and educational repositories, while private production infrastructure repositories may contain richer or more carefully maintained HPA configurations. To reduce this bias, we retain toy-project and confidence indicators as sensitivity variables instead of using them to filter the main corpus. Sensitivity analyses over subsets such as HPA-active files, recently committed files, and starred repositories show that the central pattern remains stable: even in more selective subsets, many HPA files use only a limited portion of the available API surface. The results should therefore be interpreted as evidence about public open-source HPA practice, not as a direct estimate of private production Kubernetes usage.

Finally, the statistical analyses should be interpreted descriptively. The mined corpus is not a random sample of all Kubernetes deployments, and GitHub search itself introduces coverage and ranking biases. We use non-parametric tests and effect sizes to summarize differences observed within the collected corpus, not to claim population-level generalization to all Kubernetes users. The replication package includes the notebooks and generated datasets needed to inspect, reproduce, and adapt the analyses.

8 Conclusion

This paper presented a large-scale empirical study of Horizontal Pod Autoscaler (HPA) usage in open-source Kubernetes projects, based on 9,764 HPA manifests from 12,741 repositories. We examined API evolution, configuration richness, maintenance, co-evolution, and statically detectable configuration smells. To support this analysis, we introduced the HPA Feature Richness Score (HFRS), which captures signal, control, and adjustment richness, and used it to characterize recurring configuration archetypes. The results reveal a persistent gap between API adoption and capability utilization: although `autoscaling/v2` is dominant, many manifests still use only a small portion of its configuration surface, and richer configurations are more common in actively maintained repositories. The smell analysis further shows that legacy API usage, structurally minimal v2 configurations, fixed-replica HPAs, unsupported behavior fields, frozen files, and unusually low CPU targets can be detected at scale and may warrant inspection.

HPA is only one example of a Kubernetes resource whose API has accumulated optional capabilities over time. Similar adoption-utilization gaps may exist in probes, resource requests and limits, ingress configuration, deployment strategies, and network policies. Studying these resources comparatively could reveal whether shallow capability adoption is specific to autoscaling or a more general property of infrastructure-as-code evolution. Such evidence would help researchers and tool builders design better migration guidance, documentation, and automated support for configuration modernization.

Artifact Availability

A replication package for this study is publicly available at https://doi.org/10.5281/zenodo.22134989.

Acknowledgements

Generative AI tools assisted in structuring and revising this manuscript, as well as in generating code for data analysis and visualization.

References

[1] Hussain Ahmad, Christoph Treude, Markus Wagner, and Claudia Szabo. 2024. Smart HPA: A Resource-Efficient Horizontal Pod Auto-Scaler for Microservice Architectures. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. IEEE, 46–57. doi:10.1109/ICSA59870.2024.00013

[2] Adem Ait, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2022. An Empirical Study on the Survival Rate of GitHub Projects. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 365–375. doi:10.1145/3524842.3527941

[3] Dariusz R. Augustyn, Łukasz Wyciślik, and Mateusz Sojka. 2024. Tuning a Kubernetes Horizontal Pod Autoscaler for Meeting Performance and Load Demands in Cloud Deployments. *Applied Sciences* 14, 2 (2024), 646. doi:10.3390/app14020646

[4] Dibyendu Brinto Bose, Akond Rahman, and Shazibul Islam Shamim. 2021. "Under-reported" Security Defects in Kubernetes Manifests. In *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*. 9–12. doi:10.1109/EnCyCriS52570.2021.00009

[5] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. 2017. An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 323–333. doi:10.1109/MSR.2017.67

[6] Jailton Coelho, Marco Tulio Valente, Luciano Milen, and Luciana L. Silva. 2020. Is This GitHub Project Maintained? Measuring the Level of Maintenance Activity of Open-Source Projects. *Information and Software Technology* 122 (2020), 106274. doi:10.1016/j.infsof.2020.106274

[7] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. 2020. Toward a Catalog of Software Quality Metrics for Infrastructure Code. *Journal of Systems and Software* 170 (2020), 110726. doi:10.1016/j.jss.2020.110726

[8] Kelsey Hightower, Brendan Burns, and Joe Beda. 2017. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media, Inc., Sebastopol, CA, USA.

[9] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2016. An In-depth Study of the Promises and Perils of Mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071. doi:10.1007/s10664-015-9393-5

[10] Kubernetes Authors. 2025. Deprecated API Migration Guide. https://kubernetes.io/docs/reference/using-api/deprecation-guide/. Accessed: 2026-05-02.

[11] Kubernetes Authors. 2026. Horizontal Pod Autoscaling. https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/. Accessed: 2026-05-02.

[12] Kubernetes Authors. 2026. HorizontalPodAutoscaler Walkthrough. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/. Accessed: 2026-02-24.

[13] Kubernetes Authors. 2026. Kubernetes API Concepts. https://kubernetes.io/docs/reference/using-api/api-concepts/. Accessed: 2026-05-02.

[14] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2021. A Systematic Review of API Evolution Literature. *Comput. Surveys* 54, 8, Article 171 (2021), 36 pages. doi:10.1145/3470133

[15] Benoît Lemoine. 2024. Kubernetes Horizontal Pod Autoscaler Enhanced Design for Protected Workloads. In *2024 IEEE Symposium on Computers and Communications (ISCC)*. 1–7. doi:10.1109/ISCC61673.2024.10733692

[16] Tania Lorido-Botrán, José Miguel-Alonso, and Jose A. Lozano. 2014. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing* 12, 4 (2014), 559–592. doi:10.1007/s10723-014-9314-7

[17] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. 2020. Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. *Sensors* 20, 16 (2020), 4621. doi:10.3390/s20164621

[18] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 164–175. doi:10.1109/ICSE.2019.00033

[19] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. 2023. Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* 32, 4, Article 99 (2023), 36 pages. doi:10.1145/3579639

[20] Shazibul Islam Shamim, Hanyang Hu, and Akond Rahman. 2025. On Prescription or Off Prescription? An Empirical Study of Community-Prescribed Security Configurations for Kubernetes. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2432–2444. doi:10.1109/ICSE55347.2025.00170

[21] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does Your Configuration Code Smell?. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 189–200. doi:10.1145/2901739.2901761

[22] Tien Van Do, Nam H. Do, Csaba Rotter, T. V. Lakshman, Csaba Biró, and Tamás Bérczes. 2025. Properties of Horizontal Pod Autoscaling Algorithms and Application for Scaling Cloud-Native Network Functions. *IEEE Transactions on Network and Service Management* 22, 2 (2025), 1889–1898. doi:10.1109/TNSM.2025.3532121

[23] Melina Vidoni. 2022. A Systematic Process for Mining Software Repositories: Results from a Systematic Literature Review. *Information and Software Technology* 144 (2022), 106791. doi:10.1016/j.infsof.2021.106791

[24] Rafael Xavier, Bruno B. P. Cafeo, Baldoino Fonseca, Davy de Medeiros Baia, and Elder Cirilo. 2025. LSTM-based Forecasting for Non-linear Workloads in On-premises Software Systems. In *Proceedings of the XXXIX Brazilian Symposium on Software Engineering*. Sociedade Brasileira de Computação, 260–270. doi:10.5753/sbes.2025.9917

[25] Junwei Xiao, Fan Yang, Wei Ai, Tao Meng, and Guoqi Xie. 2026. PAHPA: Revolutionizing Kubernetes Autoscaling With Integrated Predictive Analytics and Real-Time Monitoring. *IEEE Transactions on Services Computing* 19, 3 (2026), 2343–2355. doi:10.1109/TSC.2026.3685325

[26] Ahmed Zerouali, Ruben Opdebeeck, and Coen De Roover. 2023. Helm Charts for Kubernetes Applications: Evolution, Outdatedness and Security Risks. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 523–533. doi:10.1109/MSR59073.2023.00078

[27] Yue Zhang, Rachel Meredith, Wilson Reeves, Júlia Coriolano, Muhammad Ali Babar, and Akond Rahman. 2024. Does Generative AI Generate Smells Related to Container Orchestration? An Exploratory Study with Kubernetes Manifests. In *Proceedings of the 21st International Conference on Mining Software Repositories*. ACM, 192–196. doi:10.1145/3643991.3645079

[28] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The Impact of Continuous Integration on Other Software Development Practices: A Large-scale Empirical Study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 60–71. doi:10.1109/ASE.2017.8115619