Feature Toggle Dynamics in Large-Scale Systems: Prevalence, Growth, Lifespan, and Benchmarking

Abstract. Feature toggles enable gradual rollouts and experimentation in software systems, yet often persist beyond their intended lifecycle, accumulating as technical debt. Prior research has examined feature toggle interactions and complexity, but no longitudinal study has quantified how toggles evolve over time across different organizational contexts. We analyse over 4,000 toggle events in Kubernetes (10 MLoC, 8.5 years) and GitLab (5 MLoC, 5 years). We find that feature toggle removals lags behind additions in both systems (by roughly 35% and 13%, respectively), leading to growing toggle inventories. Their lifespan patterns also differ notably, with Kubernetes toggles lasting a median of 734 days versus 185 in GitLab. Then, some feature toggles (1.33% and 0.73%, respectively) exceed all previously observed removal durations, becoming de facto permanent. Building on these findings, we propose a benchmarking framework with five key metrics and their empirically derived threshold zones, enabling practitioners to assess and compare toggle management practices across projects. All scripts and data are publicly available.

Keywords: Feature toggles lifecycle · Benchmarking framework · Software maintainability · Software engineering · Empirical study

1 Introduction

Feature toggles have become an essential part of modern software development [9]. They are Boolean switches [11] that allow teams to deploy unfinished features, run experiments on live traffic, and roll back problematic changes, all without touching the deployment pipeline. In theory, toggles are meant to be temporary [17]. Once a toggled feature is completed, the toggle should be removed and the code cleaned up. In practice, things rarely work out that way.

The problem is that feature toggles are easy to add and easy to forget [27]. A common scenario unfolds when a developer adds a toggle to guard a new feature in the system, during the time the feature stabilizes, and the toggle quietly remains in the codebase. Months later, nobody remembers why it exists or who owns it. Thus, the toggle becomes permanent not by design, but by neglect. If we multiply this across a large codebase, we may end up with dozens, sometimes hundreds [26], of toggles accumulating like sediment in the codebase. Each one adds conditional logic and makes the system harder to test and understand.

Despite the widespread use of feature toggles [10,16,22,24,29], we know surprisingly little about their actual lifecycle in production systems. Existing studies and gray literature address feature toggle interactions [30,29], added complexity [25], usage patterns [24], addition/removal practices [23,14,19], underlying

rationales [17,11], management [21,16,3], and standardization [20]. Yet key questions remain unexplored: *How many feature toggles exist in modern codebases? How fast are they added compared to how fast they are removed? Do active feature toggles accumulate over time, or do teams keep up with cleanup?* And importantly, *how long do individual feature toggles survive before someone finally deletes them?* These questions matter because the answers determine whether feature toggle management is sustainable at scale or whether teams are slowly drowning in technical debt [26,19] they cannot see.

This paper provides empirical answers to such questions. We conducted a longitudinal study of feature toggles in two large-scale open source systems, namely Kubernetes [1], a container orchestration platform with roughly 10 million lines of Go, and GitLab [2], a DevOps application with about 5 million lines of Ruby. By mining over 8.5 and 5 years, respectively, of their version control history, we tracked when toggles were added, when they were removed, and which ones are still alive despite exceeding all historical precedents. We find that feature toggle lifecycles vary greatly across projects. Basically, what counts as "overdue" in one codebase may be perfectly normal in another. We also find that cleanup of toggles rarely keeps pace with their addition, leading to a gradual accumulation of active toggles over time (about 35% and 13%, respectively).

Moreover, building on these findings, we propose the first benchmarking framework for feature toggle management. The framework introduces five metrics that quantify toggle events, accumulation rate, cleanup effectiveness, codebase density, and relative lifespan, enabling systematic cross-project comparison. We then derive their threshold zones empirically using Kubernetes and GitLab as baselines. The resulting framework offers practitioners concrete means to assess toggle health and benchmark their practices against real-world baselines.

In summary, this paper makes the following contributions:

- A longitudinal study of feature toggle dynamics in two large-scale systems, covering over 4,000 toggle events across more than 8 years of development.
- Empirical evidence that toggle lifecycles are project-dependent, with *e.g.,* median lifespans differing by nearly four times between software systems.
- A benchmarking framework with five complementary metrics and threshold zones, enabling practitioners to assess toggle health across projects. An interactive dashboard is available at `https://anonconfs.github.io/fthub/`.
- A replication package containing all scripts and data is publicly available at `https://anonymous.4open.science/r/V2026/` to enable reproducibility.

The rest of this paper is structured as follows. Section 2 provides background on feature toggles and the need for empirical lifecycle studies. Section 3 outlines our research questions, subject systems, and data processing methodology. Section 4 presents findings on toggle dynamics. Section 5 introduces our benchmarking framework, followed by threats to validity (Section 6), related work (Section 7), and conclusion with implications and future directions (Section 8).

[1] Kubernetes codebase, `https://github.com/kubernetes/kubernetes`.
[2] GitLab codebase, `https://gitlab.com/gitlab-org/gitlab`.

Listing 1.1. Feature gate definition in Kubernetes, /pkg/features/kube_features.go

```
1  const (
2  // owner: @bswartz
3  // Enables usage of any object for volume data source in PVCs
4  AnyVolumeDataSource featuregate.Feature="AnyVolumeDataSource"
5  //... the rest of feature gate definitions
6  )
7  var defaultVersionedKubernetesFeatureGates = map[featuregate.
       Feature]featuregate.VersionedSpecs {
8      AnyVolumeDataSource: { // ...
9      {Version: version.MustParse("1.33"), Default: true,
           PreRelease: featuregate.GA, LockToDefault: true},
           // GA in 1.33 -> remove in 1.36
10     }, // ...
11 }
```

Listing 1.2. Feature gate usage, /pkg/api/persistentvolumeclaim/util.go

```
1  // Drop the contents of the dataSourceRef field if the
       AnyVolumeDataSource feature gate is disabled.
2  if !utilfeature.DefaultFeatureGate.Enabled(features.
       AnyVolumeDataSource) {
3      if !dataSourceRefInUse(oldPVCSpec) {
4          pvcSpec.DataSourceRef = nil
5      }
6  }
```

2 Background and Motivation

Feature toggles are simple Boolean switches embedded in codebase that enable or disable functionality at runtime, allowing teams to deploy incomplete features, conduct A/B experiments, and control rollouts without redeploying [10,11]. They have become particularly valuable in modern Continuous Integration and Continuous Delivery (CI/CD) pipelines [12], where they decouple deployment from release and enable safer, more frequent deliveries. Listing 1.1 illustrates how feature toggles (also known as *feature gates* in Kubernetes) are defined within a codebase, using `AnyVolumeDataSource` as an example (line 4), while Listing 1.2 shows their conditional usage within that codebase. Notably, Kubernetes embeds explicit removal timelines in feature gate definitions (comment in line 9 in Listing 1.1), indicating when a gate should be removed after the guarded feature is stabilized. This practice supports proactive lifecycle management. Hence, originally designed as temporary scaffolding for gradual delivery, feature toggles are supposed to be removed (*i.e.,* lines 2-10 in Listing 1.1 and 1-2, 6 in Listing 1.2) once a toggled feature is completed (*cf.* lines 3-5 in Listing 1.2) and the code cleaned up [17]. In practice, however, they often linger. Each forgotten toggle

adds conditional logic that increases cyclomatic complexity [26], expands the state space requiring testing [25], and makes the codebase harder to understand and reason about. Prior research has examined toggle interactions [29,30] and complexity [25], yet, to the best of our knowledge, there remains no systematic empirical evidence on *toggle prevalence in modern codebases, their temporal accumulation patterns, and their actual survival duration in production.*

Without empirical data on these dynamics, teams cannot tell if their toggle inventory is growing out of control, whether a six-month-old toggle is normal or overdue, or whether cleanup keeps pace with addition. To the best of our knowledge, they lack benchmarks to distinguish healthy additions / removals of toggles from silent accumulation, making evidence-based removal policies impossible. In fact, practitioners have long recognized this feature toggles accumulation problem, some have even proposed a "one-in-one-out" rule, requiring the removal of an existing toggle before adding a new one [11]. Others advocate embedding explicit expiration dates directly in toggle definitions (*cf.* Listing 1.1, line 9) to enable timely and potentially automated cleanup. Yet, without quantitative data, teams cannot assess for instance when, at what age of their project, to apply such rule. Then, this challenge of toggle accumulation will likely intensify in the next few months as AI-assisted development accelerates. Recently, a GitHub's 2025 Octoverse report shows that over 986 million code pushes happened just last year, with AI tools speeding up feature delivery [8]. As experimentation becomes easier and iteration cycles shorten, toggle volumes is expected to grow accordingly [7], further making difficult manual management of feature toggles. This gap motivates both empirical investigation and the establishment of quantitative benchmarks that help teams assess their toggle health, compare practices across organizational contexts, and determine when intervention is needed.

3 Study Design

We describe our research questions, subject systems, and data collection below.

3.1 Research Questions

This study investigates the lifecycle dynamics of feature toggles in large-scale software systems. Specifically, we track how many feature toggles are added (*i.e.,* introduced) and removed (*i.e.,* cleaned up) over time, how many remain active, and how long they persist before removal. The goal is to provide empirical evidence that supports better toggle management practices. We address three research questions using data from Kubernetes and GitLab.

RQ_1**: How do the counts of feature toggles added and removed evolve over the lifecycle of software projects?** To answer this question, we tracked addition and removal counts in Kubernetes and GitLab. This analysis reveals the churn dynamics of feature toggles, specifically whether removal keeps pace with addition rates, and identifies bulk events (refactoring,

merging) that produce significant spikes. Understanding these patterns helps assess whether toggles accumulate silently or are actively managed.

RQ_2: **How does the number of active feature toggles change over the lifecycle of software projects?** To this end, we computed the cumulative sum of additions minus removals to track the operational feature toggle inventory over time. This reveals whether active feature toggles grow unboundedly, stabilize, or decline within a project, each reflecting different management practices. We also normalized counts by lines of code to compare toggle density across projects independently of codebase size.

RQ_3: **How do the lifespans of feature toggles differ between software projects?** To this end, we measured the time from toggle addition to removal using survival analysis (Kaplan-Meier estimators). This determines whether toggles are short-lived or persistent, and whether lifespan patterns are universal or project-specific. We then classified toggles into temporary, intermediate, and long-lived based on project-specific quartiles, and identified de facto permanent toggles exceeding all historical removal precedents.

3.2 Analysed Projects

To address our research questions, we analysed the feature toggles in two real-world software systems, namely Kubernetes [3] and GitLab [4]. These projects are suitable subjects because they are large, actively maintained, widely adopted, and differ significantly in architecture. Kubernetes is a Go-based container orchestration platform and GitLab is a Ruby on Rails DevOps application, with approximately 10 million (9.98M LoC) and 5 million (4.86M LoC) lines of code, respectively, in the last analysed version of them [5]. Their release cycles also differ. Kubernetes releases approximately quarterly while GitLab releases monthly.

The way feature toggles are defined and used in these systems also differs. Kubernetes implements toggles as *feature gates*, which are defined in the pkg/features/kube_features.go file, whereas GitLab implements them as *feature flags*, which are defined in dedicated YAML files under the `config/feature_flags/` directory. Throughout this paper, we usually use the terms *gates* and *flags* when referring specifically to Kubernetes and GitLab feature toggles, respectively. Moreover, in this study, we analysed the commit histories of Kubernetes from June 06, 2014 (2c4b3a5) to August 19, 2025 (4e8b192), covering approximately 11 years and 2 months, and of GitLab from October 09, 2011 (93efff94) to August 22, 2025 (8127b1c5), covering about 13 years and 10 months.

The historical trajectories of feature toggles also differ. Kubernetes was started on June 06, 2014, but on January 20, 2017, it introduced for the first time 5 feature gates in the main branch (which we analysed), about 2.5 years after its start. However, the GitLab repository was created on October 09, 2011, but its first feature flag appeared in the main branch only on July 08, 2020, almost 9

[3] Kubernetes project, `https://github.com/kubernetes/kubernetes`.
[4] GitLab project, `https://gitlab.com/gitlab-org/gitlab`.
[5] Measured with `cloc`, `https://github.com/AlDanial/cloc`.

years later. This means we analysed in this study approximately 8 years and 7 months of feature gate history in Kubernetes and 5 years and 1 month of feature flag history in GitLab, up to August 2025. Currently, Kubernetes has 155 active feature gates, and GitLab has 403 active feature flags, all of which are Boolean.

3.3 Mining Feature Toggles

Typically, feature toggles are defined in a central location within a software system, either at the statement level or the file level. For instance, in Kubernetes, feature gates are declared as constants within specific files, such as the `AnyVolumeDataSource` gate in the `pkg/features/kube_features.go` file shown in Listing 1.1 (line 4). As for GitLab, feature flags are defined in their own YAML files organized within dedicated directories, such as for example the `access_rest_chat` flag in the `config/feature_flags/ops/` directory.

To analyse the evolution of feature toggles, specifically the addition of new toggles or removal of existing ones, we extracted and processed the version control history of the corresponding files and directories. For Kubernetes, we examined the commit history of the `pkg/features/kube_features.go` file, where most of the gates are defined. For GitLab, we analysed the commit history of all eight subdirectories under the `config/feature_flags/` directory. For each file and directory, we parsed the commit logs and identified toggle additions and removals through pattern matching on the unified diff output. Lines prefixed with `+` indicated additions, and lines prefixed with `-` indicated removals. For Kubernetes, we matched feature gate declarations of the form `MyFeature featuregate.Feature = "MyFeature"`, whereas for GitLab, we tracked the creation and deletion of `*.yml` files. Specifically, we recorded the name of the changed gate or flag, whether the change represented an addition or a removal, the commit SHA, and the associated timestamp. These data served as inputs for our subsequent analysis, including determining the active status and lifespan of individual feature toggles. To validate the accuracy of this automated extraction, we cross-checked the results by manually inspecting the commit logs of a few randomly selected toggles and comparing the obtained active toggles against the listed toggles in the project documentation and command-line outputs (for this, we used `kube-apiserver --help` for Kubernetes, and `find config/feature_flags -name '*.yml' | wc -l` for GitLab).

Moreover, to ensure full reproducibility, all build scripts, processed data, and results are publicly available at `https://anonymous.4open.science/r/V2026/`.

4 Results

This section presents the results obtained regarding the research questions.

4.1 Added and Removed Feature Toggles

As a first step, we examined the number of feature toggles added and removed over time in both subject systems. Figure 1 presents the monthly totals of

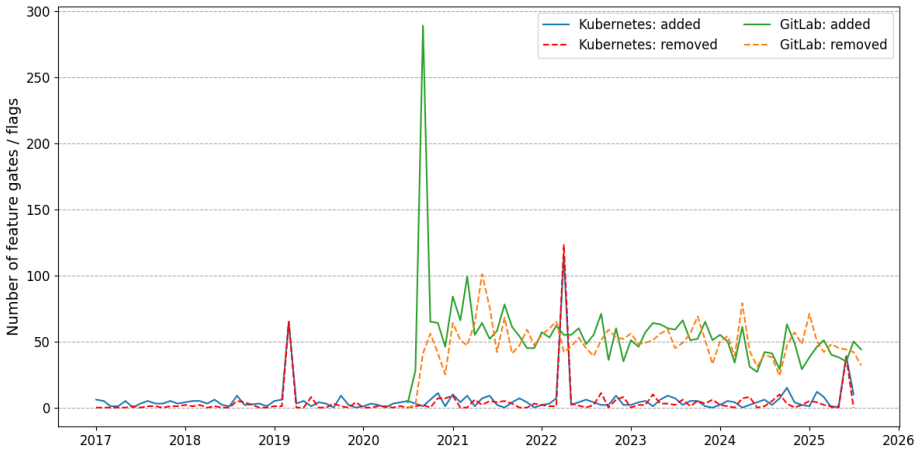

Fig. 1. Added vs. removed feature gates in Kubernetes and flags in GitLab

changes for Kubernetes and GitLab. The analysis revealed that 603 feature gates were added over 8 years and 7 months in Kubernetes, of which 448 were subsequently removed within the same period. Similarly, over 5 years and 1 month in GitLab, 3,442 feature flags were added, and 3,038 were later removed.

When comparing the line charts of added and removed toggles in Figure 1, it can be observed that, for both systems, the numbers of added and removed gates or flags generally evolve in parallel. This suggests that most toggles added are eventually removed, maintaining a relatively consistent balance between additions and removals over time. There are, however, a few notable exceptions. Specifically, there are three major bulk additions and removals of feature gates in Kubernetes, and one major bulk addition of feature flags in GitLab. The three bulk changes in Kubernetes were triggered by refactoring. The first occurred in early 2019, when the definition syntax of feature gates was updated. Namely, the old format `MyFeature utilfeature.Feature = "MyFeature"` was replaced by `MyFeature featuregate.Feature = "MyFeature"`, resulting in the refactoring of 65 gates. The second took place in April 2022, when all feature gates were reordered alphabetically to reduce merge conflicts. On the same day, several mature gates were removed, leading to 112 additions and 123 removals. The most recent bulk change, in June 2025, also involved alphabetizing, during which 34 gates that were not properly sorted were adjusted. In GitLab, a significant number of feature flags were added toward the end of 2020, when 222 development feature flags were merged into the master branch as part of a migration process, effectively incorporating nearly all development flags into the main codebase.

On average, 5.85 feature gates are added per month in Kubernetes and 55.52 feature flags in GitLab, while 4.35 gates and 49.00 flags are removed, respectively. These averages indicate that, slightly more feature toggles are added than removed (by 34.48% and 13.31% each month, respectively). The differences in

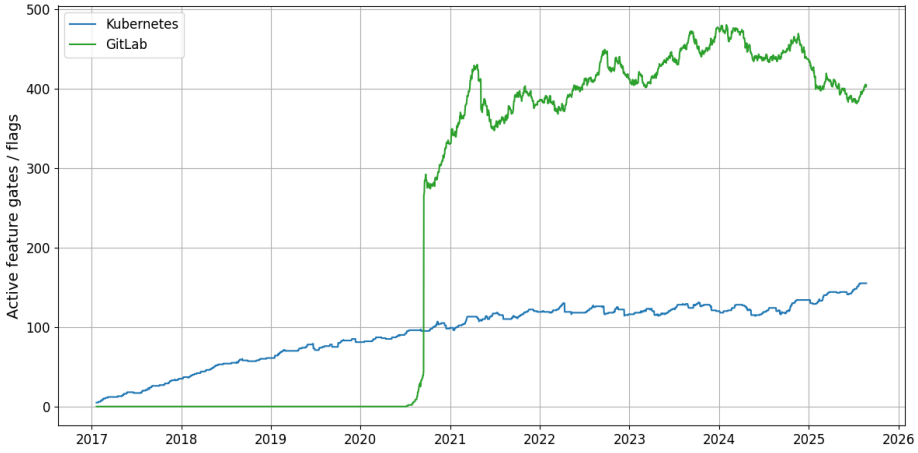

Fig. 2. Daily active toggles in Kubernetes and GitLab, cumulative values

these averages suggest that the rate of change in amount and frequency varies across projects. A closer investigation showed that they are influenced by factors such as refactoring, release phases, and feature toggle management practices.

> RQ_1 **insights:** Feature toggles are regularly added and removed in software projects. However, the frequency varies significantly across projects (GitLab has 10 times more toggle events per month than Kubernetes). Despite these differences, the roughly parallel evolution of additions and removals within each project indicates that teams actively manage feature toggle lifecycles, balancing experimentation with maintainability even as a small net accumulation of toggles persists (34% surplus in Kubernetes, 13% in GitLab).

4.2 Active Feature Toggles

To address our second research question, we analysed the number of feature toggles that were active in the codebases on each day. We did this by calculating the cumulative sum of added minus removed toggles in both subject systems, which allowed us to track whether their numbers increased, stabilized, or decreased over the project's history. We first noted that in the last analysed commits, 155 feature gates in Kubernetes and 403 feature flags in GitLab are active.

Figure 2 shows the evolution of active feature toggles in both Kubernetes and GitLab. We observed that Kubernetes started with 5 active gates on January 20, 2017. Over the course of that year, the number grew almost linearly, reaching a maximum of 35 active gates on December 22, 2017. This almost linear growth continued for about two years. Then, beginning in March 2019, the number of active gates in Kubernetes rose steadily from 70 to 155 by July 29, 2025, reflecting a net growth rate of approximately 0.036 gates per day (about 13.3

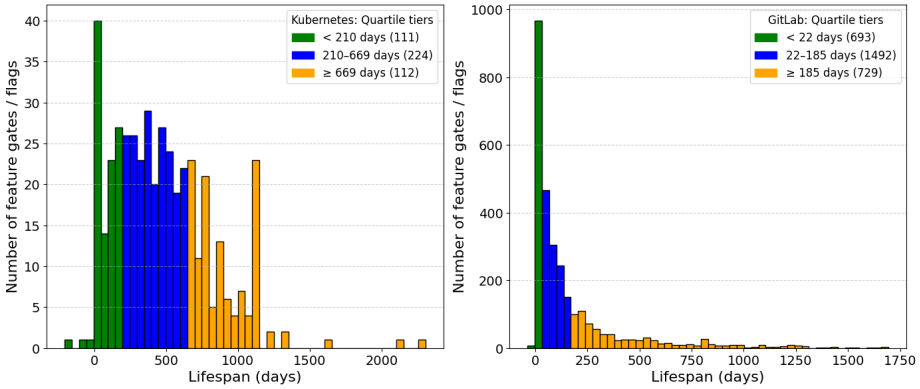

Fig. 3. Distribution of feature toggles longevity in Kubernetes and GitLab

per year) over 2,339 days. We observed that this growth pattern was shaped by multiple additions between release cycles and subsequent removals prior to releases, producing visible stepwise changes in the cumulative totals. In contrast, Figure 2 shows that GitLab's active feature flags increased sharply from 1 on July 8, 2020, to 335 within the same year. Growth persisted until reaching 403 active flags on August 23, 2025, before showing signs of stabilization and a slight recent decline. This translates to an average net growth rate of about 0.215 flags per day (approximately 78.6 per year) across 1,872 days in GitLab.

Despite their differing growth trajectories, both systems maintain considerable operational feature toggle surfaces. On average, Kubernetes has 102.6 active toggles per day, compared to 394.2 in GitLab. When normalized by lines of code (LoC), the disparity is more pronounced. Based on the last analysed commits, Kubernetes has one gate per ≈64 kLoC, while GitLab exhibits a much denser toggle surface with about one flag per ≈12 kLoC. This five fold higher toggle density in GitLab highlights a significant imbalance and underscores the considerable ongoing effort required to manage long-lived toggles in both codebases.

> RQ_2 **insights:** Active feature toggles generally accumulate over a software project's lifecycle, though growth rates vary significantly (from 13 to 79 feature toggles per year in our study). When normalized by codebase size, feature toggle density can differ fivefold across projects, reflecting different release cycles, organizational practices, and maintenance overhead.

4.3 The Lifespan of Feature Toggles

To address our third research question, we analysed the lifespan of feature gates in Kubernetes and feature flags in GitLab across their entire project histories. Specifically, we measured the time elapsed between the addition and removal of each feature toggle. Then, we classified toggles into three categories: *temporary*,

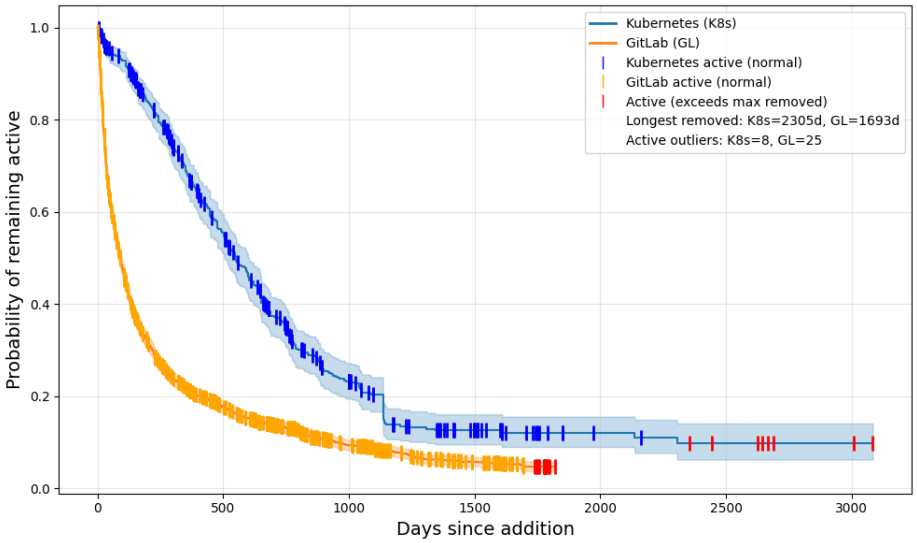

Fig. 4. Kaplan-Meier survival curves showing feature toggle lifespans in Kubernetes and GitLab. Active toggles are marked with vertical ticks, whereas red marks indicate those exceeding the maximum observed lifespan of removed toggles

■ (removed within the first 25% of the overall lifespan), *intermediate*, ■ (removed between 25% and 75% of the overall lifespan), *long-lived*, ■ (removed at or beyond 75% of the overall lifespan), and *permanent*, ▌(never removed, or with a lifespan exceeding that of the longest-lived feature gate or flag).

The results for Kubernetes, shown in Figure 3 (left), indicate that 111 feature gates were *temporary*, 224 had an *intermediate* lifespan, and 112 were *long-lived*. For GitLab (Figure 3, right), 693 feature flags were *temporary*, 1,492 *intermediate*, and 729 *long-lived*. While the proportions across tiers are similar (approximately 25%, 50%, and 25%) due to quartile-based classification, the key insight lies in the *absolute day thresholds* that define each tier. In Kubernetes, *temporary* means fewer than 210 days, whereas in GitLab it means fewer than 22 days, that is, nearly a tenfold difference. This implies that a toggle considered "temporary" in Kubernetes would already qualify as "long-lived" in GitLab, underscoring how strongly feature toggle lifecycle expectations differ across projects.

The maximum observed toggle lifespan also differs considerably between the two projects, which is 2,305 days (approximately 6.3 years) in Kubernetes versus 1,696 days (approximately 4.6 years) in GitLab. Moreover, the average lifespan is 471.6 days in Kubernetes compared to 165.7 days in GitLab, indicating that toggles in Kubernetes persist nearly three times longer on average.

To complement the lifespan distribution of removed toggles above, we applied survival analysis to model how long feature toggles remain active before removal. The Kaplan-Meier estimator is well-suited for this purpose because it accounts for *censored* data, that is, toggles still active at the end of our study

whose eventual removal date remains unknown [2]. Unlike the histogram analysis in Figure 3, which only includes toggles that have been removed, the survival analysis incorporates all toggles, both removed and currently active. As a result, the survival curves in Figure 4 reveal notable differences in toggle lifecycle management. In Kubernetes, half of all feature gates remain active for at least 734 days (roughly 2 years), whereas in GitLab, the median survival time is only 185 days (about 6 months), nearly four times shorter. This gap is also visible in the shape of the curves, showing that Kubernetes maintains a survival probability above 50% until around day 700, whereas GitLab's curve drops below this threshold much earlier, suggesting more aggressive or frequent cleanup practices.

Of particular concern are also the toggles represented by red vertical markers in Figure 4, which denote currently active toggles that have already exceeded the maximum observed lifespan among all historically removed toggles in their respective systems (*i.e.,* 2,305 days in Kubernetes and 1,696 days in GitLab). Specifically, we identified eight Kubernetes feature gates and 25 GitLab feature flags that have surpassed these thresholds. These outliers represent de facto *permanent* toggles, which have persisted well beyond typical cleanup windows and now risks becoming deeply entangled with the codebase.

The existence of such long-lived active toggles raises a critical question: *when does a "temporary" experimental feature effectively become permanent?* Our data suggest that once a toggle survives beyond the historical maximum removal threshold, its probability of eventual removal drops considerably. The survival curve analysis thus provides not merely a descriptive mean but a predictive way for those toggles that may exceed the historical norms of their longevity. Moreover, this finding highlights the importance of proactive toggle management practices in software projects, including the use of explicit expiration dates, automated cleanup mechanisms, and regular toggle reviews within development workflows. Such measures can help prevent the accumulation of stale toggles, reduce code complexity, and enhance overall system maintainability.

> RQ_3 **insights:** Feature toggle lifespans are highly system dependent, with Kubernetes toggles persisting nearly three times longer on average (472 vs. 166 days) and half of them surviving at least four times longer (734 vs. 185 days). A small but important subset of active toggles (8 in Kubernetes, 25 in GitLab) has exceeded all historical removal thresholds, effectively becoming permanent and highlighting the need for proactive cleanup mechanisms.

5 Feature Toggle Benchmarking Framework

Our analysis of Kubernetes and GitLab reveals that toggle management practices vary significantly across projects, with differences in churn rates (10 times higher in GitLab), toggle density (5 times higher per kLoC), and median lifespans (4 times longer in Kubernetes). These findings raise a natural question: *how can practitioners determine whether their own toggle inventory is healthy or problematic?* For example, *Is 20 active toggles too many for a 200 kLoC*

project? or *Is adding 10 toggles per month sustainable?* To address these questions, we propose five *benchmark metrics* for cross-project comparison of toggle management practices, together with an *interpretation framework* consisting of empirically derived threshold zones using Kubernetes and GitLab as baselines.

5.1 Benchmark Metrics

We propose five benchmark metrics to enable comparison of feature toggle management practices across software projects, namely **toggle density** (prevalence), **churn rate** and **net accumulation** (growth dynamics), and **cleanup ratio** and **normalized lifespan** (lifecycle management).

For a project $p \in \mathcal{P}$, where $\mathcal{P} = \{\text{Kubernetes}, \text{GitLab}, \ldots\}$, let $|\mathcal{A}^{(p)}|$ denote the number of active feature toggles, $|\text{additions}^{(p)}|$ the total toggles added, $|\text{removals}^{(p)}|$ the total toggles removed, $L^{(p)}$ the lines of code, $T^{(p)}$ the analysis period in months, and $\tau^{(p)}$ the average release cycle duration (in days).

***C**hurn Rate.* The first benchmark metric is the churn rate ($C^{(p)}$). It quantifies the total monthly toggle activity by summing additions and removals. This captures how actively a project manages its toggle inventory, with higher values indicating more frequent experimentation and cleanup cycles. It is computed as:

$$C^{(p)} = \frac{|\text{additions}^{(p)}| + |\text{removals}^{(p)}|}{T^{(p)}} \tag{1}$$

***N**et Accumulation.* We then propose using net accumulation ($N^{(p)}$), which measures the difference between addition and removal rates, indicating whether toggles are accumulating in the codebase over time. A positive value signals that toggle cleanup is not keeping pace with new toggle additions, potentially leading to technical debt. It is computed as:

$$N^{(p)} = \frac{|\text{additions}^{(p)}| - |\text{removals}^{(p)}|}{T^{(p)}} \tag{2}$$

*Cleanup **R**atio.* The third metric is the cleanup ratio ($R^{(p)}$), which measures the proportion of added toggles that are eventually removed, indicating historical cleanup discipline in a software system. It is computed as:

$$R^{(p)} = \frac{|\text{removals}^{(p)}|}{|\text{additions}^{(p)}|} \tag{3}$$

*Toggle **D**ensity.* The fourth metric, toggle density ($D^{(p)}$), measures the number of active feature toggles per thousand lines of code (kLoC), enabling the comparison across codebases of different sizes. It is computed as:

$$D^{(p)} = \frac{|\mathcal{A}^{(p)}|}{L^{(p)}} \times 1000 \tag{4}$$

Table 1. Feature toggle interpretation framework, with proposed thresholds for toggle inventory health assessment. Its application to ♦ Kubernetes and ■ GitLab.

Metric	Threshold	Zone	Description	Projects
Churn Rate ($C^{(p)}$)	< 15/month	Low	Deliberate changes	♦ 10.2
	15–100/month	Moderate	Balanced activity	
	≥ 100/month	High	Rapid iteration	■ 104.5
Net Accumulation ($N^{(p)}$)	< 2/month	Sustainable	Cleanup keeps pace	♦ 1.5
	2–5/month	Warning	Gradual debt	
	≥ 5/month	Critical	*One-in-one-out* needed	■ 6.5
Cleanup Ratio ($R^{(p)}$)	≥ 0.85	Healthy	≥85% removed	■ 0.88
	0.70–0.85	Warning	Potential debt	♦ 0.74
	< 0.70	Critical	Significant debt	
Toggle Density ($D^{(p)}$)	< 0.02/kLoC	Conservative	Low toggle footprint	♦ 0.016
	0.02–0.10/kLoC	Moderate	Typical density	■ 0.081
	≥ 0.10/kLoC	Aggressive	Strict cleanup needed	
Norm. LifeSpan ($S^{(p)}_{\text{norm}}$)	< 3 cycles	Short-lived	Rapid cleanup	
	3–8 cycles	Moderate	Typical lifecycle	♦ 6.1 ■ 6.2
	≥ 8 cycles	Long-lived	Extended maintenance	

*Toggle Life**S**pan.* The last metric, normalized lifespan ($S^{(p)}_{\text{norm}}$), measures toggle duration in release cycles (rather than absolute days). This normalization reflects the fact that toggles are typically removed at release milestones, making release cycles a more meaningful unit for cross-project comparison than calendar days.

Let $\tilde{S}^{(p)} = \text{median}(\mathcal{S}^{(p)})$ denote the median lifespan (in days) of all removed toggles in project p, where $\mathcal{S}^{(p)} = \{S^{(p)}_1, S^{(p)}_2, \ldots, S^{(p)}_n\}$ is the set of individual toggle lifespans. Let $\tau^{(p)}$ denote the average release cycle duration (in days) for project p. The normalized lifespan is then:

$$S^{(p)}_{\text{norm}} = \frac{\tilde{S}^{(p)}}{\tau^{(p)}} \tag{5}$$

By computing these five metrics, practitioners gain a quantitative basis for assessing feature toggle health. The following subsection establishes threshold zones for each metric, providing interpretive context for the computed values.

5.2 Interpretation Framework

While the five proposed metrics provide a quantitative view of a project's toggle management practices, the raw numbers alone do not say much without something to compare them to. Practitioners, for instance, cannot easily judge whether a given toggle density or churn rate in their system reflects healthy practices. To make these results more meaningful, we introduce a framework for interpreting feature toggle metrics. We construct this framework by calculating

all five metrics for Kubernetes and GitLab using the data from Section 4, and then defining threshold zones that treat these two systems as reference baselines.

To illustrate how we established these thresholds, consider the *Net Accumulation* metric ($N^{(p)}$). We regard a project as maintaining *sustainable* toggle management if fewer than two toggles accumulate per month, a threshold that Kubernetes meets at 1.5 toggles per month. However, when the accumulation rate reaches five or more toggles per month, the project enters a *critical* zone where toggles risk becoming technical debt, and a stricter cleanup policy, such as a "one-in-one-out" approach [11], becomes necessary. We applied similar reasoning to the other four metrics, basically using Kubernetes as a reference for *conservative* toggle management and GitLab for more *aggressive* practices. Table 1 summarizes the resulting thresholds along with their interpretations.

Using the framework is straightforward. Basically, to evaluate a project's toggle health, practitioners can compute the five metrics from the version control history and compare the results with the thresholds in Table 1. If a project consistently falls within the conservative zone across all metrics, it likely follows a Kubernetes approach characterized by a low volume of feature toggles. In contrast, if it falls within the aggressive zone, it resembles GitLab's high-churn toggle management approach. Mixed classifications (for example, high churn but a low cleanup ratio) may reveal process imbalances, such as toggle accumulation, that call for specific intervention, such as for instance cleanup.

These thresholds are intended as starting points for system self-assessment rather than strict rules. We expect that as more data become available from additional systems, the thresholds for the first four metrics can be refined to better reflect general industrial practices. In contrast, normalized lifespan inherently adapts to each system's release cycle, making it self-calibrating by design.

5.3 Applying the Framework

The last column of Table 1 shows where Kubernetes (♦) and GitLab (■) fall within each threshold zone, illustrating how the framework differentiates their toggle management approaches. Below, we show how the framework applies to each project and how this application can inform its use in other projects.

As it can be noticed, Kubernetes exhibits relatively modest toggle activity, with about ten toggle related events (additions and removals) per month ($C^{(p)} = 10.2$), placing it firmly within the *low* churn zone. Its toggle inventory grows slowly, adding roughly one and a half more toggles than it removes each month ($N^{(p)} = 1.5$), which falls comfortably within the *sustainable* range. Then, the codebase also maintains a sparse toggle footprint, with approximately one toggle per 64 kLoC ($D^{(p)} = 0.016$/kLoC), well within the *conservative* zone. This overall pattern reflects a deliberate, release oriented strategy in which toggles are introduced sparingly but kept alive for extended periods.

GitLab, in contrast, operates at a much higher intensity. The project records over one hundred toggle events per month ($C^{(p)} = 104.5$), about ten times more than Kubernetes, placing it in the *high* churn zone. Its toggle inventory grows rapidly, with a net accumulation of around six and a half toggles per month

($N^{(p)} = 6.5$), crossing into the *critical* threshold. Despite this aggressive pace, GitLab demonstrates stronger cleanup discipline. Basically, 88% of all introduced toggles are eventually removed ($R^{(p)} = 0.88$), corresponding to a *healthy* status compared to Kubernetes's 74%, which places it in the *warning* zone. This apparent paradox, higher cleanup rate yet greater accumulation, can be explained by scale, namely GitLab adds nearly ten times as many toggles, so even with effective cleanup, the absolute number of remaining toggles remains higher.

Perhaps most revealing is the normalized lifespan metric. In absolute terms, Kubernetes toggles persist nearly four times longer than those in GitLab (734 vs. 185 days). However, when expressed relative to each project's release cycle, both maintain toggles for approximately six release cycles ($S^{(p)}_{\text{norm}} \approx 6.1$ for Kubernetes with quarterly releases, ≈ 6.2 for GitLab with monthly releases). This convergence suggests that despite vastly different behaviors, both projects share a common underlying practice, that is toggles typically span about six releases before removal, regardless of whether those releases occur monthly or quarterly.

These data reveal two distinct but equally valid approaches for feature toggle management. The first is a *low-volume, long-tail* approach, exemplified by Kubernetes, where toggles are introduced sparingly, maintained across multiple release cycles, and removed through longer deprecation windows. The second is a *high-volume, short-lived* approach, exemplified by GitLab, where toggles serve as lightweight, disposable switches that enable rapid feature iteration and A/B testing, but require aggressive cleanup discipline to prevent inventory accumulation. Our findings do not indicate that either approach is inherently superior. We believe that each reflects organizational practices and development rhythms tailored to the project's domain, release cycle, and acceptable failure cost.

5.4 Interactive Benchmarking Dashboard

To support adoption of the proposed framework, we created an interactive web-based dashboard, which is available at `https://anonconfs.github.io/fthub/`.

The dashboard supports three main use cases. For *visual comparison*, practitioners can inspect how Kubernetes and GitLab score across the five benchmark metrics on a radar chart, illustrating low-volume, long-tail versus high-volume, short-lived toggle management practices. For *self-assessment*, teams can enter their own project metrics using the provided web form and see where they fall within the threshold zones, highlighting potential areas of concern. Finally, for *community contribution*, users can submit and save new project data to a .csv file, enabling the benchmarking dataset to grow and the threshold boundaries to be refined in the future as more evidence is collected.

Providing such a framework through an interactive dashboard lowers the barrier for practitioners looking to assess and improve their toggle management practices. Then, as more projects are added, it can become an evolving collection of feature toggle lifecycle benchmarks across diverse software ecosystems.

6 Threats to Validity

This section discusses the threats to the validity of this study's findings.

6.1 Internal Validity

One internal threat concerns *toggle identification* accuracy. Our mining approach relies on pattern matching within specific files (Kubernetes' `kube_features.go`) and directories (GitLab's `config/feature_flags/`). A small number of toggles defined elsewhere, in the case of Kubernetes, may be excluded from our analysis. However, manual inspection confirmed that these represent the main toggle definition locations in both projects and that any omissions are minimal.

Another threat involves *refactoring events*. Bulk additions and removals caused by code refactoring (*e.g.,* syntax changes, alphabetical reordering, merging) can distort our findings. We identified three such events in Kubernetes and one in GitLab. While these spikes affect monthly counts, they do not impact much lifespan calculations since refactored toggles retain their original creation dates.

A further threat relates to *negative lifespans*. A small number of toggles exhibit negative computed lifespans, likely due to git history anomalies such as rebasing, cherry-picking across branches, or timezone inconsistencies in commit metadata. Given their negligible count (3 in Kubernetes, 8 in GitLab), we retained these outliers, which do not significantly affect our lifespan analyses.

6.2 Construct Validity

The five metrics in our benchmarking framework capture quantitative aspects of toggle lifecycle management. However, they may not fully represent toggle *health*. Factors like toggle complexity, code coupling, test coverage of toggle paths, and documentation quality also affect maintainability but are not included. Future work could extend our framework to incorporate these qualitative dimensions.

6.3 External Validity

A first threat is related to *generalizability*. Our study is based on only two open source systems with different architectures, sizes, and toggle management practices. Although Kubernetes and GitLab are large, and well-documented projects, the results may not generalize to closed source software, smaller codebases, or projects with different release cycles and development workflows. Additionally, the increasing adoption of AI-assisted development tools may alter toggle dynamics and proposed thresholds in ways not captured by our historical data.

Then, *toggle type homogeneity* is another threat. In both systems, feature toggles are only Boolean switches. Projects that rely on multi-valued toggles, percentage-based rollouts, or toggles with more explicit expiration metadata may show different lifecycle patterns that we did not cover in this study.

The *time span* of our data may poses another threat. Our analysis covers 8.5 years of history for Kubernetes and 5 years for GitLab, and then toggle

management practices can change over time. As a result, the observed patterns may reflect more mature processes rather than the intrinsic properties of toggles themselves. Therefore, replicating this study on additional systems and time periods would help strengthen the generalizability of our findings.

7 Related Work

Feature toggle research has expanded rapidly in recent years as developer teams increasingly rely on toggles to manage runtime variability in their systems. Meinicke *et al.* [17] contrasted feature toggles with configuration options, highlighting key similarities and differences as well as clarifying when each mechanism is best suited for use. Rahman *et al.* [24] documented usage patterns in large codebase systems like Chromium. Meinicke *et al.* [16] illustrated through mining studies how feature toggles have become widespread in open-source projects, yet Neely and Stolt [19] and Ramanathan *et al.* [26] showed that these toggles can accumulate technical debt when left unmanaged. Additional research by Rahman *et al.* [25], Schröder *et al.* [29], and Tërnava *et al.* [30] explored the complexity and interdependencies that toggle code introduces, often associating toggles with greater maintainability challenges and added burdens for software management. In response, tools like Piranha by Ramanathan *et al.* [26] and AI-assisted Reflag [28] aim to automate toggle removal after their purpose has been served, helping teams keep codebases maintainable and more secure by reducing technical debt and minimizing attack surfaces. This study builds on similar earlier research and grey literature (*e.g.,* Davies [4], Unleash [31]) by taking a longitudinal approach to observe how feature toggles are added, removed, retained, and vary in lifespan. Moreover, we show how toggle ecosystems evolve over time rather than in isolated snapshots. This reveals the real, ongoing lifecycle pressures teams face with feature toggles, providing actionable insights to improve toggle governance and prioritize cleanup in evolving codebases.

A separate body of related work addresses *compile-time* variability in software product lines and configurable systems. On the evolution side, Ernst *et al.* [6] studied the usage of C/C++ preprocessors in practice, while Sundermann *et al.* [13] recently analysed the Linux kernel's feature model across two decades and over 3,000 versions to track feature and configuration evolution. Regarding collaborative benchmarks, for example, the S.P.L.O.T repository [18] and ESPLA catalog [15] offer collections of real-world feature models from product lines, with ESPLA also tracking extractive adoption cases plus metadata on system size and feature counts. The Universal Variability Language (UVL) [1] provides another dataset of real-world models to represent variability across tools. On the measurement side, El-Sharkawy *et al.* [5] introduced MetricHaven, a tool supporting variability-aware code metrics that combine information from code files and variability models, offering over 23,000 metric variations for assessing product line quality. These and similar efforts focus mainly on compile-time variability mechanisms such as preprocessor directives and feature models. Our work instead examines *runtime* variability through feature toggles in CI/CD en-

vironments. Still, we share their goal of helping practitioners measure variability prevalence and compare practices across projects. Moreover, our work builds on this path by delivering the first empirical study of feature toggle dynamics plus a set of grounded metrics for assessing toggle lifecycle health.

8 Conclusion

We presented the first longitudinal study of feature toggle dynamics (prevalence, growth, and lifespan) in large-scale software systems. By analysing over 4,000 toggle events in Kubernetes (8.5 years) and GitLab (5 years), we provided empirical evidence on how many toggles are added, removed, and persist over time.

Our findings reveal four key insights. First, feature toggles are prevalent in modern codebases, with active inventories reaching hundreds of toggles (155 in Kubernetes, 403 in GitLab). Secondly, toggle removals consistently lag behind additions, leading to gradual inventory growth in software systems (35% surplus in Kubernetes, 13% in GitLab). Thirdly, toggle lifecycles are highly project-dependent, with median lifespans differing by nearly four times across software systems (734 days in Kubernetes vs. 185 in GitLab), reflecting fundamentally different management approaches. Lastly, a small but concerning subset of toggles tend to become permanent in the codebase (8 in Kubernetes, 25 in GitLab).

Building on these findings, we proposed a benchmarking framework with five metrics and empirically derived thresholds. To facilitate adoption, we also developed an interactive dashboard for self-assessment and community contributions.

Future work could strengthen these findings in several ways. Extending the dataset to additional systems would help validate and refine the proposed thresholds. Investigating how AI-assisted development affects toggle dynamics is also increasingly relevant as such tools become widespread. On the practical side, building CI/CD tools that flag toggles approaching critical thresholds and automating their removal once they expire could help teams manage toggle debt proactively. Finally, incorporating qualitative factors like toggle complexity and documentation quality would provide a more comprehensive view of toggle health.

Ultimately, we hope this work helps teams make informed decisions about their toggle practices and encourages broader efforts to establish common standards for when to add, how long to keep, and when to remove feature toggles.

References

1. Benavides, D., Sundermann, C., Feichtinger, K., Galindo, J.A., Rabiser, R., Thüm, T.: Uvl: Feature modelling with the universal variability language. JSS **225**, 112326 (2025). https://doi.org/https://doi.org/10.1016/j.jss.2024.112326
2. Bland, J.M., Altman, D.G.: Survival probabilities (the Kaplan-Meier method). BMJ pp. 1572–1580 (1998). https://doi.org/10.1136/bmj.317.7172.1572
3. Bryan, A.: 11 open-source feature flag tools (2024), https://www.getunleash.io/blog/11-open-source-feature-flag-tools, accessed: 2025-12-11
4. Davies, A.: Feature toggles: The good, the bad, and the ugly (2018), https://www.youtube.com/watch?v=r7VI5x2XKXw, accessed: 2025-12-11

5. El-Sharkawy, S., Krafczyk, A., Schmid, K.: MetricHaven: More than 23,000 metrics for measuring quality attributes of software product lines. In: SPLC. p. 25–28. ACM (2019). https://doi.org/10.1145/3307630.3342384
6. Ernst, M., Badros, G., Notkin, D.: An empirical analysis of C preprocessor use. TSE **28**(12), 1146–1170 (2002). https://doi.org/10.1109/TSE.2002.1158288
7. GitHub: Octoverse: A new developer joins GitHub every second as AI leads TypeScrip to#1 (2025), https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/, accessed: 2025-12-11. September 2024–August 2025 data
8. GitHub: What 986 million code pushes say about the developer workflow in 2025 (2025), https://github.blog/news-insights/octoverse/what-986-million-code-pushes-say-about-the-developer-workflow-in-2025/, accessed: 2025-12-11
9. GitHub Engineering: How we ship code faster and safer with feature flags (2021), https://github.blog/engineering/ship-code-faster-safer-feature-flags/, accessed: 2025-11-29
10. Henderson, C.: Flipping out (December 2009), https://code.flickr.net/2009/12/02/flipping-out/, accessed: 2025-12-11
11. Hodgson, P.: Feature toggles (aka feature flags) (November 01, 2021), https://martinfowler.com/articles/feature-toggles.html, accessed: 2025-12-11
12. Humble, J., Farley, D.: Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education (2010)
13. Kuiter, E., Sundermann, C., Thüm, T., Heß, T., Krieter, S., Saake, G.: How configurable is the Linux kernel? Analyzing two decades of feature-model history – RCR report. TSE (August 2025). https://doi.org/10.1145/3764666
14. Mahdavi-Hezaveh, R., Dremann, J., Williams, L.: Software development with feature toggles: Practices used by practitioners. EMSE **26**(1) (January 2021). https://doi.org/10.1007/s10664-020-09901-z
15. Martinez, J., Assunção, W.K.G., Ziadi, T.: ESPLA: A catalog of extractive SPL adoption case studies. p. 38–41. ACM (2017). https://doi.org/10.1145/3109729.3109748, https://doi.org/10.1145/3109729.3109748
16. Meinicke, J., Hoyos, J., Vasilescu, B., Kästner, C.: Capture the feature flag: Detecting feature flags in open-source. In: MSR. p. 169–173. ACM (2020). https://doi.org/10.1145/3379597.3387463
17. Meinicke, J., Wong, C.P., Vasilescu, B., Kästner, C.: Exploring differences and commonalities between feature flags and configuration options. In: ICSE-SEP. p. 233–242. ACM (2020). https://doi.org/10.1145/3377813.3381366
18. Mendonca, M., Branco, M., Cowan, D.: S.P.L.O.T.: Software product lines online tools. p. 761–762. ACM (2009). https://doi.org/10.1145/1639950.1640002
19. Neely, S., Stolt, S.: Continuous delivery? Easy! Just change everything (well, maybe it is not that easy). In: 2013 Agile Conference. pp. 121–128. IEEE (2013). https://doi.org/10.1109/AGILE.2013.17
20. OpenFeature: OpenFeature: Standardizing feature flagging for everyone (2025), https://openfeature.dev/, accessed: 2025-12-11
21. Osherove, R.: Feature toggle framework list (2021), https://pipelinedriven.org/feature-toggle-frameworks-list/, accessed: 2025-12-11
22. Prutchi, E.S., de S. Campos Junior, H., Murta, L.G.P.: How the adoption of feature toggles correlates with branch merges and defects in open-source projects? SPE **52**(2), 506–536 (2022). https://doi.org/10.1002/spe.3034

23. Rahman, M.T., Querel, L.P., Rigby, P.C., Adams, B.: Feature toggles: Practitioner practices and a case study. In: MSR. p. 201–211. ACM (2016). https://doi.org/10.1145/2901739.2901745
24. Rahman, T.: Feature toggle usage patterns: A case study on Google Chromium. In: MSR. pp. 142–147. IEEE (2023). https://doi.org/10.1109/MSR59073.2023.00032
25. Rahman, T., Shalabi, I., Sharma, T.: Exploring influence of feature toggles on code complexity. In: EASE. p. 363–368. ACM (2024). https://doi.org/10.1145/3661167.3661190
26. Ramanathan, M.K., Clapp, L., Barik, R., Sridharan, M.: Piranha: Reducing feature flag debt at Uber. In: ICSE. p. 221–230 (2020). https://doi.org/10.1145/3377813.3381350
27. Reddit user: What is the ideal way to add implementation switch or feature flags in code? (2024), https://www.reddit.com/r/ExperiencedDevs/comments/1cb2mzm/what_is_the_ideal_way_to_add_implementation/, accessed: 2025-12-11
28. Reflag: Building AI flag cleanup (2025), https://reflag.com/blog/building-ai-flag-cleanup, accessed: 2025-12-11
29. Schröder, M., Kevic, K., Gopstein, D., Murphy, B., Beckmann, J.: Discovering feature flag interdependencies in Microsoft Office. In: ESEC/FSE. p. 1419–1429. ACM (2022). https://doi.org/10.1145/3540250.3558942
30. Tërnava, X., Lesoil, L., Randrianaina, G.A., Khelladi, D.E., Acher, M.: On the interaction of feature toggles. In: VaMoS. ACM (2022). https://doi.org/10.1145/3510466.3510485
31. Østhus, E.: Feature toggle life time best practices (June 25 2021), https://www.getunleash.io/blog/feature-toggle-life-time-best-practices, accessed: 2025-12-11