Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

DriGitScale: Drift-Aware GitOps for Consistency-Aware Predictive Autoscaling in Kubernetes

Ahmed Fathi AboElazm

ahmmedfathy_pgs@fci.capu.edu.eg

Helwan University

Abdalla Sayed Yassin

Helwan University

Hanan Fahmy

Helwan University

Research Article

Keywords: Autoscaling, Kubernetes, GitOps, Drift Detection, Time-Series Forecasting, Self-Adaptive Systems.

Posted Date: October 1st, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11019483/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

DriGitScale: Drift-Aware GitOps for Consistency-Aware Predictive Autoscaling in Kubernetes

Ahmed Fathi AboElazm
Department of Information Systems
Faculty of Computers and Artificial Intelligence
Capital University (Formerly Helwan University)
Helwan 11795, Egypt
ahmmedfathy_pgs@fci.capu.edu.eg or ahmed.fathi.g@gmail.com

Hanan Fahmy
Department of Information Systems
Faculty of Computers and Artificial Intelligence
Capital University (Formerly Helwan University)
Helwan 11795, Egypt
hanan.fahmy@fci.capu.edu.eg

Abdalla Sayed Yassin
Department of Information Systems
Faculty of Computers and Artificial Intelligence
Capital University (Formerly Helwan University)
Helwan 11795, Egypt
abdallasayed@fci.capu.edu.eg or abdallasayedyassin@gmail.com

***Abstract*—Autoscaling has become a fundamental capability for cloud-native applications operating under dynamic and evolving workload conditions. Although recent predictive autoscaling approaches have improved workload forecasting accuracy through machine learning and deep learning techniques, most existing solutions primarily focus on prediction performance while providing limited support for maintaining system-level consistency during scaling operations. Consequently, cloud-native environments may experience oscillatory scaling behavior, deployment inconsistencies, and reduced operational stability despite achieving high forecasting accuracy. To address these limitations, DriGitScale is proposed as a drift-aware GitOps-enabled self-adaptive autoscaling model for Kubernetes-based cloud-native systems that explicitly considers runtime consistency in addition to workload prediction. Unlike existing prediction-centered autoscaling approaches that primarily optimize forecasting accuracy, DriGitScale formulates autoscaling as a consistency-management problem by jointly evaluating prediction drift, resource drift, and configuration drift within a closed-loop decision-making process. This enables scaling decisions to be validated against runtime consistency conditions rather than relying solely on workload prediction accuracy, thereby reducing unnecessary scaling operations and improving deployment stability. The framework integrates multiple forecasting models, including Naive, Moving Average (MA), ARIMA, and LSTM, with GitOps-based deployment synchronization to maintain alignment between desired and actual system states. Experimental evaluation using NASA workload traces, bursty Wikipedia workloads, and synthetic datasets demonstrates that drift-aware decision validation improves scaling stability, reduces unnecessary scaling actions, and maintains competitive responsiveness with acceptable computational overhead. The results indicate that DriGitScale improves scaling stability, reduces unnecessary scaling actions, and preserves deployment consistency while maintaining competitive responsiveness across diverse workload conditions, demonstrating the practical benefits of consistency-aware autoscaling beyond prediction accuracy alone.**

***Index Terms*—Autoscaling, Kubernetes, GitOps, Drift Detection, Time-Series Forecasting, Self-Adaptive Systems.**

I. Introduction

Cloud-native applications have become the dominant paradigm for deploying modern distributed systems due to their scalability, flexibility, and support for continuous service delivery. These applications commonly operate on Kubernetes-based infrastructures and are often exposed to dynamic and evolving workload conditions. As workload demands fluctuate over time, efficient resource management becomes essential for maintaining application performance while minimizing resource waste and operational costs. Consequently, autoscaling has become a fundamental component of cloud-native environments, enabling applications to dynamically adjust resource allocation according to workload requirements. Among the most widely adopted autoscaling mechanisms is the Kubernetes Horizontal Pod Autoscaler (HPA), which automatically adjusts the number of running pods based on resource utilization metrics such as CPU, memory, custom metrics, and external metrics [11], [12].

Traditional autoscaling mechanisms primarily rely on reactive decision-making strategies in which scaling actions are triggered after workload changes have already occurred. Although reactive approaches are relatively straightforward to configure and deploy, they often suffer from delayed scaling responses, resource over-provisioning, under-provisioning, scaling oscillation, and potential service-level agreement (SLA) violations when operating under highly dynamic workloads [4], [5]. These limitations become increasingly significant in large-scale cloud-native environments where workload patterns may change rapidly and unpredictably. As a result,

maintaining stable and efficient autoscaling behavior remains a major challenge for modern Kubernetes deployments.

To overcome the limitations of reactive autoscaling, recent studies have explored predictive autoscaling techniques that forecast future workload behavior before scaling decisions are executed. Predictive autoscaling utilizes forecasting models such as ARIMA, LSTM, GRU, and hybrid machine learning approaches to proactively allocate resources according to anticipated workload demands [2], [3], [24], [25]. Other studies have explored graph-based learning techniques, attention-driven forecasting models, and optimization-oriented frameworks, including GraphOpticon, to enhance resource allocation efficiency and workload prediction in cloud-native environments [1], [20], [21], [26]. Collectively, these studies report improvements in prediction quality and proactive resource management under dynamic workloads. Nevertheless, their effectiveness is still assessed mainly in terms of forecasting accuracy and resource optimization, while limited attention is given to maintaining runtime consistency among workload predictions, resource allocation, and deployment configurations.

Despite the progress achieved by predictive autoscaling approaches, prediction accuracy alone does not necessarily guarantee stable autoscaling behavior. In practical cloud-native environments, scaling decisions are influenced not only by workload forecasts but also by runtime resource behavior and deployment configuration management. Consequently, inconsistencies may arise between predicted workload requirements, actual resource utilization, and deployed system configurations. Such inconsistencies may lead to unstable scaling decisions, oscillatory scaling behavior, redundant scaling actions, and reduced operational efficiency even when forecasting models achieve high prediction accuracy. Furthermore, although GitOps has emerged as an effective paradigm for maintaining consistency between desired and actual deployment states through declarative configuration management [15], [16], current predictive autoscaling solutions rarely incorporate GitOps principles directly into the autoscaling decision process.

These observations indicate that the fundamental challenge extends beyond workload prediction itself. Instead, autoscaling should be viewed as a system-level consistency problem in which prediction outcomes, runtime resource conditions, and deployment states must remain continuously aligned. To address this challenge, this paper proposes DriGitScale (Drift-Aware GitOps Self-Adaptive Autoscaling Framework), a novel framework for Kubernetes-based cloud-native environments. DriGitScale integrates prediction drift, resource drift, and configuration drift within a closed-loop decision-making architecture to continuously evaluate system consistency during autoscaling operations. In addition, the framework combines multiple forecasting models, including Naive forecasting, Moving Average (MA), ARIMA, and LSTM, with GitOps-based deployment synchronization to maintain alignment between desired and actual system states.

The main contributions of this paper can be summarized as follows:

- The proposal of DriGitScale, a drift-aware GitOps-enabled self-adaptive autoscaling model that enables consistency-aware scaling decisions by jointly evaluating prediction drift, resource drift, and configuration drift before executing scaling actions.
- The introduction of a unified drift formulation that combines prediction drift, resource drift, and configuration drift into a single consistency score for adaptive scaling decisions.
- The development of a consistency-aware scaling mechanism that validates scaling actions based on system-wide consistency rather than prediction accuracy alone.
- The integration of GitOps-based deployment synchronization to reduce discrepancies between desired and deployed system states.
- A comprehensive experimental evaluation using NASA workload traces, bursty Wikipedia workloads, and synthetic datasets to assess stability, responsiveness, drift behavior, and computational efficiency.

The remainder of this paper is organized as follows. Section II presents the background and fundamental concepts related to predictive autoscaling, drift-aware systems, and GitOps-based cloud-native environments. Section III reviews existing related work and discusses the limitations of current predictive autoscaling approaches. Section IV formulates the problem addressed in this study and defines the key system components and optimization objectives. Section V presents the DriGitScale framework and overall system architecture. Section VI describes the methodology and algorithms adopted in the proposed solution. Section VII explains the experimental setup and evaluation datasets. Section VIII reports the experimental results and performance analysis. Section IX provides an in-depth discussion of the obtained findings. Section X outlines threats to validity. Finally, Section XI concludes the paper and presents future research directions.

II. Background

This section reviews the fundamental concepts and technologies that underpin predictive autoscaling in cloud-native environments. It introduces cloud-native architectures and Kubernetes orchestration, reviews Kubernetes autoscaling mechanisms and workload forecasting approaches, discusses drift in adaptive systems, and outlines GitOps-based deployment consistency. Together, these concepts establish the foundation upon which DriGitScale is developed.

A. Cloud-Native Systems and Kubernetes Architecture

Cloud-native applications have become the dominant deployment model for modern distributed systems due to their scalability, flexibility, and support for continuous delivery. Such applications are commonly designed using a microservices architecture, where application functionality is decomposed into multiple independently deployable services. Microservices are typically packaged as containers, enabling portability, isolation, and consistent execution across different computing environments.

Kubernetes has emerged as the de facto orchestration platform for managing containerized applications at scale. Within Kubernetes, applications are organized into pods, which represent the smallest deployable units in the cluster. Pods are managed through deployments that specify the desired application state and the required number of replicas. In addition, services provide stable communication endpoints that enable interaction among distributed application components. Kubernetes continuously monitors cluster state and automatically maintains the desired number of replicas to ensure application availability and reliability [11], [12].

Despite these benefits, cloud-native systems often operate under highly dynamic workload conditions characterized by traffic fluctuations, workload bursts, and changing resource demands. Consequently, resource management becomes a challenging task because insufficient resources may degrade application performance, whereas excessive resource allocation increases operational costs. These challenges highlight the need for intelligent autoscaling mechanisms that can dynamically adapt resource allocation to changing workload demands.

B. Kubernetes Autoscaling Mechanisms

Autoscaling has become a fundamental component of Kubernetes-based cloud-native infrastructures because it enables applications to automatically adjust resource allocation according to workload variations. Kubernetes provides several built-in autoscaling mechanisms that operate at different levels of the infrastructure stack.

The Horizontal Pod Autoscaler (HPA) automatically adjusts the number of pod replicas based on observed metrics such as CPU utilization, memory consumption, custom application metrics, or external metrics [11], [12]. The Vertical Pod Autoscaler (VPA) dynamically modifies CPU and memory allocations assigned to individual containers, allowing applications to adapt their resource requirements without changing the number of replicas. In addition, the Cluster Autoscaler adjusts the number of worker nodes within the cluster according to overall infrastructure demand.

These autoscaling mechanisms rely on monitoring services such as the Kubernetes Metrics Server, which continuously collects resource utilization information. Metrics obtained from the Metrics Server serve as the basis for scaling decisions and may include CPU usage, memory consumption, custom application metrics, and external workload indicators. From a scalability perspective, HPA and VPA primarily operate at the pod level by managing application containers and replica counts, whereas the Cluster Autoscaler operates at the node level by adding or removing worker nodes. The combination of pod-level and node-level scaling enables Kubernetes to maintain resource efficiency across both application and infrastructure layers.

Reactive autoscaling remains the traditional approach adopted by many Kubernetes deployments. In reactive autoscaling, scaling actions are triggered only after workload changes have already occurred. Although reactive mechanisms are relatively straightforward to configure and deploy, they may suffer from delayed scaling decisions, resource over-provisioning, under-provisioning, and unstable scaling behavior when operating under rapidly changing workloads. These limitations have motivated the development of predictive autoscaling techniques capable of anticipating workload changes before resource saturation occurs.

C. Predictive Autoscaling and Workload Forecasting

Predictive autoscaling extends traditional reactive autoscaling by incorporating workload forecasting models into the scaling decision process. Instead of relying solely on current resource utilization measurements, predictive autoscaling estimates future workload demands and proactively allocates resources before performance degradation occurs [3], [24], [51].

Several forecasting approaches have been widely adopted in cloud autoscaling research. Statistical time-series models such as ARIMA are frequently used because of their ability to capture temporal workload patterns and trends [8], [35]. Machine learning and deep learning techniques, particularly Long Short-Term Memory (LSTM) networks, have demonstrated strong performance in modeling nonlinear and sequential workload behavior [6], [7], [24]. Other studies have investigated Gated Recurrent Units (GRU), Graph Neural Networks (GNN), and hybrid forecasting approaches to improve proactive scaling performance in distributed cloud environments [21], [26], [25].

Although predictive autoscaling significantly improves forecasting capability and resource utilization efficiency, many existing approaches primarily focus on prediction accuracy. Consequently, issues related to scaling stability, deployment consistency, and adaptive decision reliability often receive less attention than forecasting performance.

D. Drift in Adaptive Cloud Systems

Modern cloud-native environments continuously evolve due to changing workload patterns, user behavior, and application requirements. As a result, the statistical characteristics of operational data may change over time, a phenomenon commonly referred to as concept drift in machine learning [13], [14]. Concept drift can reduce forecasting accuracy and negatively affect adaptive decision-making processes.

In cloud autoscaling scenarios, multiple forms of drift may emerge. Prediction drift occurs when forecasting models no longer accurately represent current workload characteristics. Resource drift refers to deviations between expected and observed resource utilization during runtime. Configuration drift represents inconsistencies between desired deployment configurations and actual system states. Collectively, these forms of drift may contribute to unstable or oscillatory scaling decisions, inefficient resource allocation, and reduced operational reliability. Consequently, drift-aware adaptation has become an increasingly important research direction for improving the robustness and long-term effectiveness of autoscaling systems operating in dynamic cloud environments.

E. GitOps and Deployment Consistency

GitOps has emerged as a widely adopted operational paradigm for managing cloud-native infrastructures and Kubernetes applications through declarative configuration management [15], [16]. In GitOps environments, the desired system configuration is stored in version-controlled repositories, while automated synchronization tools continuously reconcile the actual deployment state with the declared configuration.

One of the primary advantages of GitOps is its ability to maintain consistency between desired and actual system states. This approach improves reproducibility, traceability, and operational reliability while simplifying infrastructure management and continuous deployment processes. As a result, GitOps has become increasingly popular in Kubernetes-based environments and large-scale cloud-native deployments.

Despite the advantages of GitOps, most predictive autoscaling approaches do not directly integrate GitOps principles into the scaling decision process. Consequently, inconsistencies may arise between workload forecasts, scaling actions, and deployed configurations. Integrating GitOps with drift-aware autoscaling mechanisms provides an opportunity to improve deployment consistency while enhancing the reliability of adaptive scaling decisions. These concepts collectively motivate the design of the proposed DriGitScale framework.

III. Related Work

Autoscaling has become a fundamental component of cloud-native systems due to the dynamic and unpredictable nature of modern application workloads. Traditional reactive autoscaling mechanisms, such as threshold-based Horizontal Pod Autoscaler (HPA) strategies in Kubernetes, trigger scaling actions according to current resource utilization metrics including CPU and memory usage [11], [12]. Although these approaches are relatively straightforward to configure and deploy, they often suffer from delayed scaling responses, resource overprovisioning, under-provisioning, and unstable scaling behavior when workload conditions change rapidly [4], [5].

Over the past few years, autoscaling research has evolved from traditional reactive mechanisms to predictive, machine learning-based, graph-based, and other adaptive approaches. Despite significant advances in workload forecasting and resource optimization, maintaining long-term scaling stability and deployment consistency remains a major challenge in dynamic cloud-native environments.

A large body of recent research has focused on predictive autoscaling by exploiting workload forecasting models. Popular forecasting techniques include statistical time-series methods such as ARIMA [8], [35], recurrent neural networks such as LSTM [6], [24], and GRU-based architectures [25]. These methods generally improve forecasting accuracy and enable more proactive resource allocation than purely reactive mechanisms. However, most evaluations emphasize prediction accuracy and resource utilization, while giving comparatively less attention to long-term scaling stability and decision reliability.

Several studies have investigated machine learning-assisted autoscaling strategies for Kubernetes and cloud-native applications. Marie-Magdelaine and Ahmed [28] proposed a machine learning-based proactive autoscaling framework for cloud-native systems. Toka et al. [29] introduced a learning-based approach for autoscaling in Kubernetes edge environments. Imdoukh et al. [30] examined predictive scaling techniques for containerized applications using machine learning models, while Abdullah et al. [33] proposed a burst-aware predictive autoscaling framework designed to improve responsiveness during sudden workload spikes. Although these methods have shown promising results, these approaches rely heavily on prediction outputs and provide limited mechanisms for validating scaling decisions under continuously changing runtime conditions.

Another active line of research investigates graph-based learning and deep learning techniques for predictive autoscaling. Graph-PHPA [21] employed a hybrid LSTM-GNN architecture to model workload dependencies and service relationships for proactive horizontal pod autoscaling. Theodoropoulos et al. [1], [22], [26] investigated graph neural networks and demand forecasting techniques to improve proactive resource allocation and fault tolerance in distributed cloud systems. Other studies explored attention-based sequence learning and adaptive forecasting techniques to enhance autoscaling performance [20], [24], [25]. Although graph-based approaches improve workload representation and service dependency modeling, they generally provide limited support for drift-aware adaptation and consistency validation during runtime operation.

Another research stream focuses on Kubernetes-specific autoscaling mechanisms and infrastructure optimization. KOSMOS [36] proposed a unified framework combining horizontal and vertical autoscaling techniques in Kubernetes environments. Tamiru et al. [37] evaluated the performance of the Kubernetes Cluster Autoscaler under different cloud deployment scenarios. Additional studies investigated the impact of relative and absolute metrics on container autoscaling [41] and proposed non-disruptive vertical autoscaling mechanisms [47]. These contributions improve infrastructure-level resource utilization and scaling efficiency; however, they primarily focus on resource optimization rather than consistency-aware autoscaling decisions.

Several researchers have also investigated workload dynamics, burstiness, and forecasting behavior in cloud systems. Roy et al. [51] proposed an early predictive autoscaling approach based on workload awareness, while Masdari and Khoshnevis [48] presented a comprehensive survey of workload forecasting techniques in cloud computing environments. Other studies investigated workload burstiness models [42], sequential learning architectures [43], and gated recurrent neural networks for time-series prediction [44]. While these approaches improve workload characterization and forecasting quality, they do not explicitly address the impact of prediction uncertainty and drift on autoscaling stability.

Another important research direction involves GitOps and

TABLE I
COMPARISON OF EXISTING AUTOSCALING APPROACHES WITH RESPECT TO PREDICTION CAPABILITY, DRIFT-AWARENESS, GITOPS INTEGRATION, SCALING STABILITY, AND OVERALL SYSTEM SCOPE.

Approach	Prediction-Based	Drift-Aware	GitOps Integration	Scaling Stability	System Scope
Reactive HPA [11], [12]	×	×	×	Low	Local Reactive
ARIMA/LSTM-based Scaling [3], [24], [25]	✓	×	×	Medium	Predictive
RL/Federated Approaches [20], [28], [33]	✓	Partial	×	Medium	Adaptive
Graph-Based Approaches [1], [21], [26]	✓	×	×	Medium–High	Global Optimization
DriGitScale (Proposed Framework)	✓	✓	✓	High	Closed-Loop Self-Adaptive

deployment consistency within cloud-native architectures. GitOps provides a declarative approach for maintaining synchronization between desired and actual deployment states through version-controlled repositories and automated reconciliation mechanisms [15]. Although GitOps has become widely adopted in cloud-native environments, current predictive autoscaling solutions rarely integrate GitOps principles directly into autoscaling decision-making processes. Consequently, inconsistencies may arise between workload forecasts, scaling actions, and deployed system configurations.

As summarized in Table I, existing autoscaling approaches differ considerably in terms of prediction capability, adaptation mechanisms, and system scope. Nevertheless, support for drift-aware adaptation, deployment-state consistency, and GitOps-integrated autoscaling remains limited across current solutions.

The comparative analysis presented in Table I reveals that existing studies have made substantial progress in workload forecasting, machine learning-driven scaling, graph-based optimization, and Kubernetes resource management. However, most current approaches evaluate autoscaling effectiveness primarily through prediction accuracy and resource efficiency metrics. As a result, limited attention has been given to drift-aware adaptation, deployment-state consistency, and synchronization between desired and actual system states.

To address these limitations, the proposed DriGitScale model introduces a consistency-aware autoscaling approach for Kubernetes-based cloud-native environments. Unlike existing approaches that primarily optimize prediction accuracy, DriGitScale jointly considers prediction drift, resource drift, and configuration drift within a closed-loop decision-making process. Furthermore, GitOps-based synchronization maintains alignment between desired and actual deployment states throughout autoscaling operations.

IV. Problem Formulation

Cloud-native applications operate under dynamic and continuously evolving workload conditions, making resource management and autoscaling challenging tasks in Kubernetes-based environments. Traditional predictive autoscaling approaches primarily focus on improving workload forecasting accuracy using statistical and machine learning models. However, high prediction accuracy alone does not necessarily guarantee stable and consistent autoscaling behavior during runtime operation. In practice, inconsistencies may emerge between predicted workload demand, actual resource allocation, and deployment configurations, leading to unnecessary scaling actions, resource inefficiencies, and unstable system behavior.

Although existing predictive autoscaling solutions have significantly improved workload forecasting, they still evaluate scaling effectiveness primarily from a prediction perspective, with limited consideration for runtime consistency. As cloud-native systems evolve, scaling decisions are influenced not only by workload predictions but also by the behavior of allocated resources and the consistency of deployed configurations. As a result, accurate workload predictions alone may still produce unstable autoscaling behavior when inconsistencies arise among different system components.

Let $W(t)$ denote the observed workload at time t, while $P(t)$ represents the predicted workload generated by forecasting models such as Naive Forecasting, Moving Average (MA), ARIMA, and LSTM. Based on workload predictions, the required computational resources are represented by $R_{required}(t)$, whereas the actual allocated runtime resources are represented by $R_{actual}(t)$.

In GitOps-enabled cloud-native environments, the desired deployment configuration maintained within the version-controlled repository is represented by $R_{desired}(t)$. Ideally, the desired state and the actual deployed state should remain synchronized. However, dynamic workload changes and runtime adaptations may introduce deviations that affect autoscaling consistency and system stability.

To formally capture these inconsistencies, DriGitScale considers three complementary drift dimensions:

- **Prediction Drift ($D_{pred}(t)$):** represents discrepancies between workload forecasts and observed workload behavior.
- **Resource Drift ($D_{res}(t)$):** represents inconsistencies between required resources and actual allocated resources during execution.
- **Configuration Drift ($D_{cfg}(t)$):** represents deviations between desired deployment configurations and actual deployed states.

The overall system inconsistency is represented through aggregated drift as follows:

$$D_{total}(t) = D_{pred}(t) + D_{res}(t) + D_{cfg}(t) \quad (1)$$

where $D_{total}(t)$ represents the cumulative inconsistency observed across workload prediction, resource allocation, and deployment configuration dimensions.

A threshold parameter ϑ is introduced to determine whether the observed inconsistency requires adaptive intervention. When the aggregated drift exceeds the predefined threshold, corrective autoscaling actions may be triggered to restore consistency and improve system stability.

Accordingly, the optimization objectives of DriGitScale can be summarized as follows:

- Minimize scaling instability.
- Minimize redundant scaling operations.
- Minimize configuration inconsistency.

subject to the consistency constraint:

$$\text{Consistency}(R_{desired}, R_{actual}) \tag{2}$$

and the adaptive scaling constraint:

$$\text{AdaptiveScaling}(D_{total}, \vartheta) \tag{3}$$

where autoscaling decisions are continuously adjusted according to observed drift behavior and deployment consistency requirements.

Accordingly, the problem addressed in this study is formulated as a consistency-aware autoscaling optimization problem rather than solely as a workload prediction problem. Unlike existing predictive autoscaling approaches that focus primarily on forecasting accuracy, DriGitScale jointly minimizes prediction drift, resource drift, and configuration drift while maintaining alignment between desired and actual deployment states. This formulation establishes the theoretical foundation for DriGitScale presented in the following section.

V. Proposed Framework

Based on the consistency-aware problem formulation presented in Section IV, this section introduces DriGitScale, the proposed consistency-aware autoscaling model for Kubernetes-based cloud-native environments. The primary objective of DriGitScale is to improve autoscaling stability by validating prediction-driven scaling decisions against runtime resource conditions and deployment-state consistency before scaling actions are executed.

Unlike traditional reactive and predictive autoscaling approaches, which primarily rely on resource utilization metrics or workload forecasts, DriGitScale treats autoscaling as a system-consistency problem. The framework continuously evaluates prediction drift, resource drift, and configuration drift to determine whether scaling actions are necessary. In addition, GitOps-based synchronization is incorporated to maintain alignment between desired and actual deployment states throughout the autoscaling lifecycle.

A. Overview

DriGitScale is designed as a closed-loop self-adaptive autoscaling model that integrates workload monitoring, predictive modeling, drift-aware decision making, and GitOps-based deployment synchronization.

Unlike traditional autoscaling approaches that rely solely on reactive metrics or workload forecasts, DriGitScale evaluates scaling decisions using a multi-dimensional consistency model that jointly considers workload prediction, runtime resource allocation, and deployment-state alignment. This architecture enables DriGitScale to maintain consistency across three complementary system states:

- Predicted workload
- Actual runtime behavior
- Desired configuration state

By explicitly modeling inconsistencies between these states, the framework addresses limitations in existing predictive and optimization-based approaches that overlook system-level drift [1], [4].

The overall architecture consists of five main operational stages, as illustrated in Fig. 1.

B. Data Ingestion and Preprocessing

Within DriGitScale, this stage is responsible for collecting real-time system metrics from the Kubernetes cluster using monitoring tools such as Prometheus.

The collected data includes:

- CPU utilization
- Workload intensity

Specifically, raw CPU utilization metrics are continuously collected from Prometheus in real-time and subsequently processed before being used for workload prediction. The collected data is normalized to a bounded range and transformed into fixed-length time-series sequences using a sliding-window mechanism. This preprocessing step ensures compatibility with time-series forecasting models and improves prediction stability.

C. Predictive Modeling

In this stage, multiple time-series models are used to forecast future CPU utilization.

DriGitScale supports:

- Naive
- Moving Average (MA)
- ARIMA
- LSTM

Each model generates a predicted CPU value at time t, denoted as:

$$C_{pred}(t) \tag{4}$$

Supporting multiple forecasting models allows DriGitScale to remain model-agnostic, allowing flexible deployment depending on system constraints such as computational cost and required prediction accuracy. This design aligns with prior predictive autoscaling approaches while extending them beyond pure forecasting [2], [3].

D. Drift-Aware Decision Engine

The Drift-Aware Decision Engine represents the primary contribution of DriGitScale and serves as the central component responsible for consistency-aware autoscaling decisions. Rather than triggering scaling actions solely based on predicted workload values, the engine evaluates multiple sources

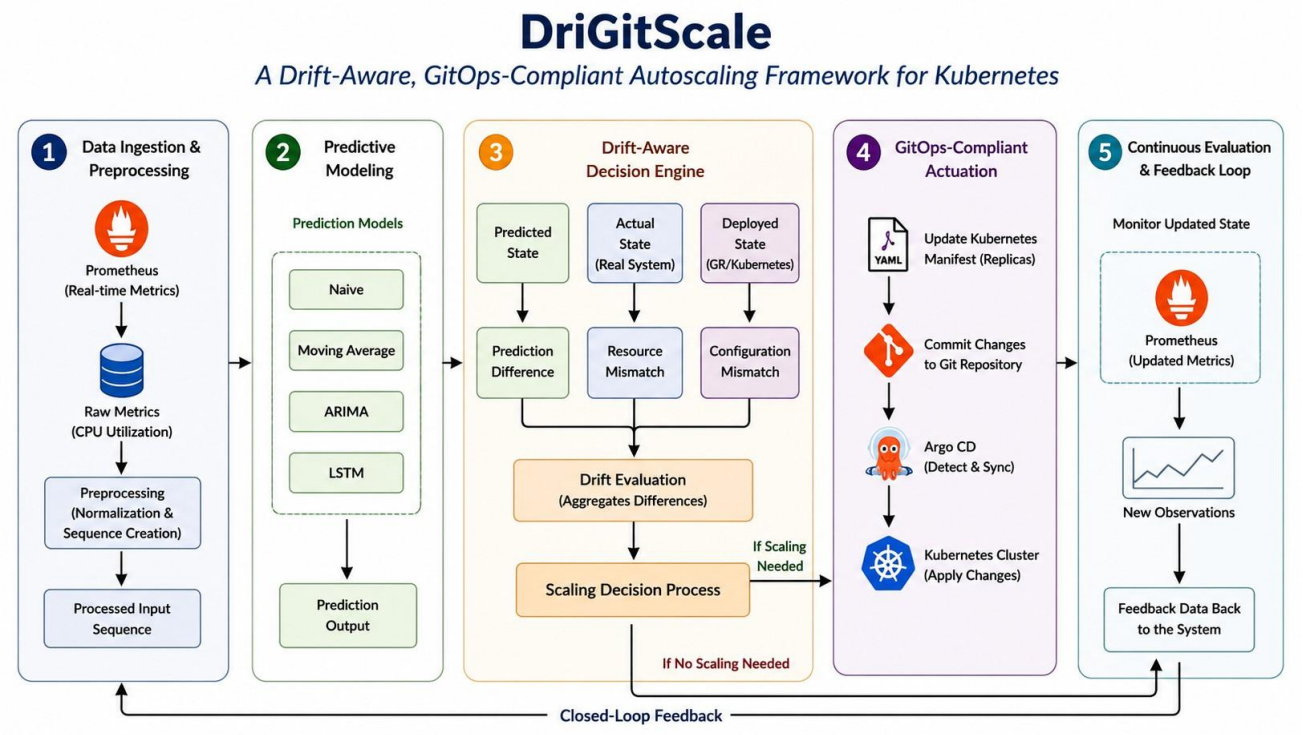


Fig. 1. Architecture of DriGitScale, showing the end-to-end closed-loop workflow that integrates data ingestion, predictive modeling, drift-aware decision making, GitOps-based actuation, and continuous feedback for self-adaptive autoscaling.

of inconsistency across the system before determining whether corrective scaling actions are required.

Consistent with the problem formulation introduced in Section IV, the system evaluates discrepancies between different system states through three complementary drift dimensions:

- **Prediction Drift:** $D_{pred} = |C_{pred} - C_{actual}|$
- **Resource Drift:** $D_{res} = |R_{required} - R_{actual}|$
- **Configuration Drift:** $D_{cfg} = |R_{git} - R_{actual}|$

These components are aggregated into a unified drift score:

$$D_{total} = \alpha D_{pred} + \beta D_{res} + \gamma D_{cfg} \quad (5)$$

This formulation operationalizes the aggregated drift concept introduced in Equation (1) by assigning weighted contributions to prediction drift, resource drift, and configuration drift. The weighting factors allow DriGitScale to adjust the relative influence of each drift dimension according to system requirements. where α, β, γ are weighting parameters controlling system sensitivity.

A scaling decision is triggered only when:

$$D_{total} > \vartheta \quad (6)$$

Equation (6) operationalizes the adaptive scaling constraint defined in Equation (3), ensuring that scaling actions are executed only when the observed system inconsistency exceeds the predefined drift threshold. where ϑ denotes the predefined drift threshold.

This mechanism filters noisy fluctuations and reduces unnecessary scaling actions (thrashing), which is a known limitation in traditional and prediction-only autoscaling approaches [4], [5].

E. GitOps-Based Actuation Layer

Once a scaling decision is validated by the Drift-Aware Decision Engine, DriGitScale initiates the execution phase through a GitOps-based actuation mechanism. Instead of directly modifying runtime resources through the Kubernetes API, scaling actions are translated into declarative configuration updates maintained within a version-controlled Git repository.

Within DriGitScale, the execution process consists of three main steps. First, Kubernetes deployment manifests are updated according to the scaling decision. Second, the modified configuration files are committed to the Git repository, which acts as the single source of truth for the desired system state. Finally, GitOps synchronization tools such as ArgoCD continuously monitor repository changes and reconcile the actual cluster state with the desired configuration.

This approach improves deployment consistency, traceability, and reproducibility while reducing the likelihood of configuration drift. Moreover, it ensures continuous alignment between desired and actual deployment states, which is a fundamental requirement in modern GitOps-enabled cloud-native environments [15], [16].

F. Continuous Feedback Loop

DriGitScale operates as a closed-loop self-adaptive system in which monitoring, prediction, drift evaluation, deci-

sion making, and deployment execution continuously interact through a feedback mechanism.

Following each scaling action, workload observations, resource utilization metrics, and deployment-state information are continuously collected from the running Kubernetes environment. These observations are subsequently used to evaluate prediction quality, identify emerging drift conditions, and assess the effectiveness of previous scaling decisions.

The resulting feedback allows DriGitScale to continuously adapt its behavior as workload conditions evolve. Consequently, the framework can respond to evolving workload patterns while minimizing unnecessary scaling actions and maintaining long-term scaling stability and deployment consistency.

G. Key Design Principle

The fundamental design principle of DriGitScale is that autoscaling should not be treated solely as a workload forecasting problem. While accurate workload prediction remains important, stable autoscaling behavior requires maintaining consistency across workload forecasts, runtime resource allocation, and deployment-state management.

Accordingly, DriGitScale evaluates scaling decisions using a multi-dimensional consistency perspective that combines prediction drift, resource drift, and configuration drift within a unified decision-making process. By integrating drift-aware validation with GitOps-based deployment synchronization, the framework aims to reduce scaling instability, avoid redundant scaling operations, and improve operational reliability in cloud-native environments.

This consistency-aware perspective differentiates DriGitScale from existing prediction-centered autoscaling approaches that primarily evaluate performance based on forecasting accuracy alone.

VI. Methodology and Algorithms

This section presents the algorithmic implementation of DriGitScale. The methodology formalizes workload monitoring, predictive modeling, drift computation, decision making, and GitOps-based execution to ensure reproducibility and facilitate experimental evaluation.

A. Multi-Dimensional Drift Formulation

The mathematical formulations introduced in Section IV are operationalized in this section through explicit drift-computation procedures. Equations (7)–(11) provide the implementation-level realization of the consistency-aware autoscaling model defined by Equations (1)–(3). Let t represent the discrete time step of the adaptive loop.

Prediction Drift ($D_{pred}(t)$) represents the absolute difference between the predicted and actual workload utilization values:

$$D_{pred}(t) = |C_{pred}(t) - C_{actual}(t)| \tag{7}$$

Where $C_{pred}(t)$ denotes the predicted CPU utilization output by the model, and $C_{actual}(t)$ is the actual runtime CPU utilization collected from cluster monitoring telemetry.

Based on workload predictions, the number of required resource instances is estimated. Let CPU_{per_pod} be the target capacity threshold allocated to an individual pod instance. The required replicas $R_{required}(t)$ is formulated using the ceiling function to ensure adequate cluster resource allocation:

$$R_{required}(t) = \frac{C_{pred}(t)}{CPU_{per_pod}} \tag{8}$$

Resource Drift ($D_{res}(t)$) captures the discrete configuration deviations between required capacity estimations and active infrastructure allocations:

$$D_{res}(t) = |R_{required}(t) - R_{actual}(t)| \tag{9}$$

Where $R_{actual}(t)$ represents the number of active running replicas inside the cluster topology.

Configuration Drift ($D_{cfg}(t)$) represents the deviations between desired system topologies declared code-side and the active cluster runtime states:

$$D_{cfg}(t) = |R_{git}(t) - R_{actual}(t)| \tag{10}$$

Where $R_{git}(t)$ denotes the current desired replica counts parsed directly from the declarative Git manifest repository files.

Since resource and configuration drift parameters operate on integer replica scales while prediction drift is bounded between 0 and 1, a scaling normalization step factor of 10 is applied to the integer metrics to ensure uniform sensitivity weight contributions during overall evaluation. The cumulative total system inconsistency is formalized as the aggregated total drift score $D_{total}(t)$:

$$D_{total}(t) = \alpha D_{pred}(t) + \beta \frac{D_{res}(t)}{10} + \gamma \frac{D_{cfg}(t)}{10} \tag{11}$$

Equation (11) represents the implementation-level realization of the aggregated drift model introduced in Equation (1). The computed drift score is subsequently used to evaluate the adaptive scaling condition defined in Equation (3). Where α, β, and γ represent structural weighting parameters controlling system sensitivity toward each respective dimension.

B. DriGitScale Algorithms

The operational procedures of DriGitScale are executed sequentially through the following decoupled operational steps.

Algorithm 1 Data Ingestion and Preprocessing

Require: Raw CPU metrics from monitoring system
Ensure: Processed time-series dataset
1: Collect CPU utilization data from Prometheus
2: Normalize values to range $[0, 1]$
3: Apply sliding window of size W
4: Generate input-output sequences
5: **return** Processed dataset

Algorithm 1 performs the initial data preparation stage of DriGitScale. It collects raw CPU utilization metrics from the monitoring infrastructure, normalizes the collected values, and

transforms them into fixed-length time-series sequences using a sliding-window mechanism. The generated sequences are subsequently used as inputs for the predictive models.

Algorithm 2 Predictive Modeling

Require: Time-series sequence
Ensure: Predicted CPU utilization $C_{pred}(t)$
1: Select prediction model: Naive / Moving Average / ARIMA / LSTM
2: Feed input sequence into the selected model
3: Perform inference
4: Generate prediction $C_{pred}(t)$
5: **return** Predicted value

Algorithm 2 describes the predictive modeling stage. Depending on the selected forecasting technique, the processed time-series sequences are provided to the prediction model to estimate future CPU utilization. The predicted workload is then forwarded to the drift computation stage for consistency evaluation.

Algorithm 3 Drift Computation Engine

Require: $C_{pred}, C_{actual}, R_{actual}, R_{git}$
Ensure: Total drift score D_{total}
1: Compute prediction drift: $D_{pred} = |C_{pred} - C_{actual}|$
2: Estimate required replicas: $R_{required} = \lceil C_{pred}/CPU_{per_pod} \rceil$
3: Compute resource drift: $D_{res} = |R_{required} - R_{actual}|$
4: Compute configuration drift: $D_{cfg} = |R_{git} - R_{actual}|$
5: Normalize drift components: $D_{res} = D_{res}/10$, $D_{cfg} = D_{cfg}/10$
6: Aggregate drift score: $D_{total} = \alpha \cdot D_{pred} + \beta \cdot D_{res} + \gamma \cdot D_{cfg}$
7: **return** D_{total}

The normalization factor was introduced to ensure that resource drift and configuration drift remain comparable to prediction drift during score aggregation. This prevents individual drift dimensions from dominating the overall inconsistency score. Algorithm 3 directly implements Equations (7)–(11) and produces the aggregated drift score required for consistency-aware scaling decisions.

Algorithm 4 Drift-Aware Scaling Decision

Require: $D_{total}, R_{required}, R_{git}$
Ensure: Final scaling decision R_{final}
1: Define drift threshold ϑ
2: **if** $D_{total} > \vartheta$ **then**
3: $R_{final} = R_{required}$
4: **else**
5: $R_{final} = R_{git}$
6: **end if**
7: **return** R_{final}

Note: The threshold parameter ϑ controls the sensitivity of the system to drift. Its impact on stability and responsiveness is analyzed experimentally in the Results section. This threshold-based mechanism enables DriGitScale to distinguish between minor fluctuations and significant consistency deviations, thereby reducing unnecessary scaling actions while maintaining system responsiveness. Algorithm 4 operationalizes the adaptive scaling constraint introduced in Equation (3) by evaluating the aggregated drift score against the predefined threshold before executing any scaling action.

Algorithm 5 GitOps-Based Actuation

Require: R_{final}
Ensure: Updated system state
1: Update replica count in YAML configuration file
2: Commit changes to Git repository
3: Push changes to remote repository
4: Trigger synchronization using ArgoCD
5: Apply updated configuration to the cluster

Algorithm 5 implements the execution phase of DriGitScale. Once a scaling decision has been validated, the desired replica configuration is updated in the Git repository. GitOps synchronization tools then reconcile the running Kubernetes cluster with the updated declarative configuration, ensuring deployment consistency.

VII. Experimental Setup

This section presents the experimental environment, datasets, prediction models, evaluation metrics, and validation procedure adopted to evaluate DriGitScale. The experiments were designed to assess not only prediction performance but also the effectiveness of the proposed consistency-aware autoscaling mechanism under different workload conditions.

A. Datasets

To provide a comprehensive evaluation, DriGitScale was assessed using both real-world and synthetic datasets. This combination allows the experiments to cover stable, bursty, and highly dynamic workload patterns while improving the generalizability of the evaluation.

NASA Workload Dataset: A popular benchmarking dataset that offers practical CPU usage records. It reflects steady workload behavior and is often employed to measure prediction-based auto-scaling algorithms [17].

Wikipedia-like Dataset: A dataset with bursts of activity featuring rapid traffic increases. It reflects real-life performance from web applications under extremely variable conditions [18].

Synthetic Dataset: An artificially created dataset intended to mimic highly volatile and noisier environments. Fluctuations are artificially introduced into the dataset to mimic situations involving extreme drift.

The selected datasets represent different workload characteristics, including relatively stable workloads (NASA), bursty and highly dynamic workloads (Wikipedia), and artificially generated extreme fluctuation scenarios (Synthetic). Together,

these datasets provide complementary workload characteristics, enabling DriGitScale to be evaluated under a broad range of operating conditions and drift scenarios.

B. System Configuration

The experiments are conducted under the following configuration:

- Sliding window size: $W = 3$
- CPU per pod: 0.3
- Maximum replicas: 10

The drift aggregation parameters are defined as: $\alpha = 0.5, \beta = 0.3, \gamma = 0.2$. The drift decision threshold is set to: $\vartheta = 0.05$.

Note: The selection of ϑ is based on a sensitivity analysis presented in the Results section, where different threshold values are evaluated to balance stability and responsiveness.

The weighting parameters were selected to assign higher importance to prediction drift while still considering resource and configuration inconsistencies in the overall drift evaluation process.

C. Prediction Models

The framework supports multiple time-series models with varying levels of complexity:

- Naive (baseline model)
- Moving Average (MA)
- ARIMA
- LSTM

Using prediction models with different levels of complexity demonstrates that DriGitScale is not tied to a specific forecasting technique and can operate under different computational and accuracy requirements.

These models were selected to represent different forecasting complexities, ranging from simple statistical baselines to advanced deep learning approaches.

D. Evaluation Metrics

The system is evaluated using the following metrics:

- **Average Drift Score** Measures the overall inconsistency between system states.
- **Inference Time** Measures the computational cost of prediction.
- **Responsiveness** Measures the time required for the system to react to workload changes. Lower values indicate faster system response.
- **Decision Stability** Measures the frequency of scaling decisions (control-level stability).
- **Actual Stability** Measures the frequency of replica changes in the deployed system (system-level stability).

Since responsiveness is measured as the time required for the system to react to workload changes, lower values indicate better performance.

E. Ablation Study Setup

To evaluate the impact of the proposed drift-aware decision mechanism, an ablation study is conducted:

Drift OFF (Baseline): Scaling decisions are based solely on predicted values without drift filtering.

Drift ON (Proposed): Scaling decisions are triggered only when the drift score exceeds the threshold ϑ.

This comparison isolates the contribution of the drift-aware decision engine and enables its impact on system stability and responsiveness to be evaluated independently of the prediction models.

F. Experimental Procedure

The overall experimental workflow is summarized as follows:

1) Load and preprocess CPU time-series data.
2) Train or initialize prediction models.
3) Generate predictions at each timestep.
4) Compute drift components and aggregate drift score.
5) Apply scaling decisions based on the drift condition.
6) Record system behavior and evaluation metrics.
7) Repeat experiments across all datasets, models, and drift configurations.

VIII. Results

This section presents the experimental results through a research-question-driven analysis. The evaluation examines the behavior of different prediction models, the contribution of drift-aware decision making, the generalizability of DriGitScale across heterogeneous workloads, and the effect of threshold selection on system behavior. Together, these experiments assess whether consistency-aware decision making improves autoscaling stability beyond prediction accuracy alone.

A. RQ1: How accurately and efficiently do different prediction models support autoscaling?

To validate the flexibility of DriGitScale, multiple prediction models were evaluated, including Naive, Moving Average (MA), ARIMA, and LSTM.

The objective of this experiment is to determine whether DriGitScale maintains stable autoscaling behavior when combined with prediction models of different computational complexity. Rather than focusing exclusively on forecasting accuracy, the experiment evaluates whether consistency-aware scaling remains effective regardless of the underlying predictor.

TABLE II
MODEL PERFORMANCE BENCHMARKING.

Model	Inference Time (s) ↓	Responsiveness ↓	Avg Drift ↓
LSTM	0.0722	0.254–0.429	0.0106–0.0154
ARIMA	0.00170	0.091–0.216	0.0121–0.0129
MA	0.000055	0.268–0.313	0.0154–0.0221
Naive	0.000004	0.097–0.194	0.0168–0.0265

As summarized in Table II, computational efficiency and scaling responsiveness vary considerably across the evaluated forecasting models. Lower inference time indicates reduced prediction overhead, whereas lower responsiveness values indicate faster adaptation to workload changes. Average drift represents consistency deviations observed during scaling decisions.

As illustrated in Fig. 2, lightweight forecasting models remain highly responsive despite their lower computational complexity. These observations confirm that prediction complexity alone does not determine autoscaling effectiveness. Although LSTM provides strong forecasting capability, lightweight approaches such as Moving Average and ARIMA maintain competitive responsiveness with substantially lower computational overhead. These findings support the proposed consistency-aware scaling philosophy, where prediction quality is integrated with drift validation rather than being treated as the sole optimization objective.

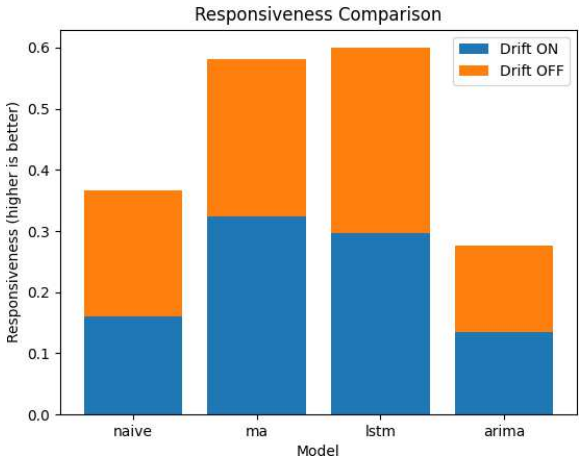


Fig. 2. Responsiveness comparison across prediction models.

B. RQ2: Does drift-awareness reduce scaling instability?

To evaluate the contribution of the drift-aware decision engine, an ablation study was conducted by comparing system behavior with and without drift-awareness.

TABLE III
DRIFT-AWARE ABLATION STUDY

Dataset	Metric	With Drift	Without Drift
NASA	Actual Stability	0.0097	0.1315
	Responsiveness	0.2394	0.2347
Wikipedia	Actual Stability	0.0081	0.0545
	Responsiveness	0.2333	0.2149
Synthetic	Actual Stability	0.0229	0.0918
	Responsiveness	0.2132	0.2315

As summarized in Table III and illustrated in Figs. 3 and 4, drift-aware decision making consistently reduces scaling instability across evaluated workloads. For example, under NASA traces, actual instability decreased from 0.1315 to 0.0097 after enabling drift-aware filtering. Similar improvements are observed for both the Wikipedia and Synthetic datasets, indicating that consistency-aware validation enhances scaling stability independently of the forecasting model.

These findings directly support the central hypothesis of this work: stable autoscaling depends not only on workload prediction quality but also on consistency-aware validation before scaling actions are executed. The observed reduction in instability across all evaluated datasets demonstrates the effectiveness of integrating prediction drift, resource drift, and configuration drift within the autoscaling decision process.

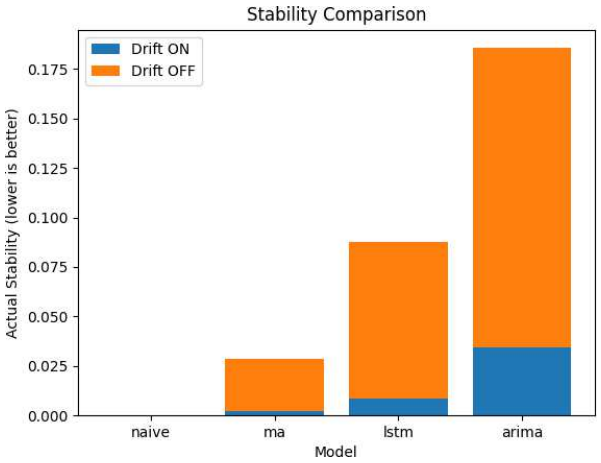


Fig. 3. Stability comparison between drift-aware and baseline configurations.

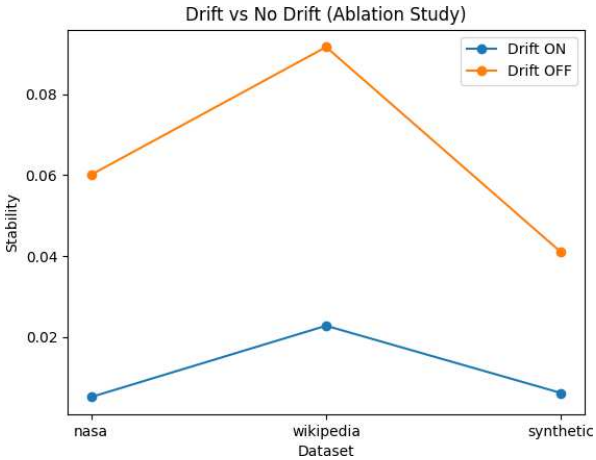


Fig. 4. Effect of drift-aware decision-making on autoscaling behavior across datasets.

C. RQ3: Does the framework generalize across heterogeneous workload patterns?

To assess the robustness of the proposed framework, experiments were conducted on multiple datasets with different workload characteristics.

The objective of this experiment is to evaluate robustness under workloads exhibiting different behavioral characteristics including smooth trends, bursty demand variation, and noisy patterns.

TABLE IV
CROSS-DATASET BEHAVIOR ANALYSIS.

Dataset	Workload Type	System Behavior	Stability
NASA	Smooth	Stable scaling	High
Wikipedia	Bursty	Controlled scaling	Medium
Synthetic	Noisy / Chaotic	Filtered decisions	Medium

As summarized in Table IV and illustrated in Fig. 5, DriGitScale remains effective across heterogeneous workload conditions. For relatively stable workloads, DriGitScale maintains highly consistent scaling behavior with minimal instability. Under bursty and noisy workloads, the drift-aware mechanism effectively suppresses unnecessary scaling reactions while preserving adaptation to significant workload changes.

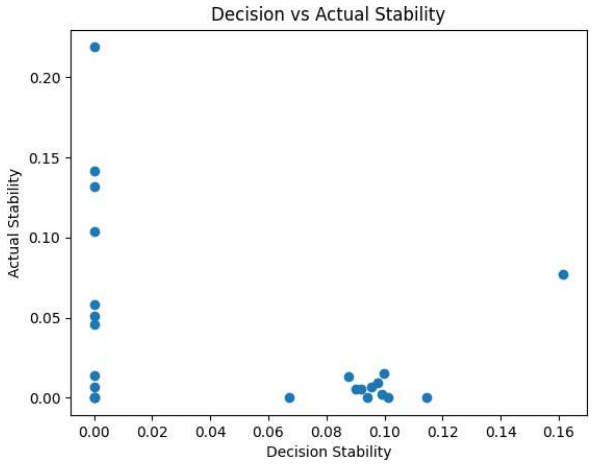


Fig. 5. Comparison between decision- level changes and actual scaling actions.

D. RQ4: How sensitive is framework behavior to threshold selection?

The effect of threshold sensitivity (ϑ) was assessed to find the best point of balance between system stability and responsiveness.

Threshold sensitivity directly affects the balance between responsiveness and scaling stability. Therefore, threshold selection must prevent excessive infrastructure reactions while preserving sufficient responsiveness to workload changes.

Several different thresholds such as 0.01, 0.02, 0.05, and 0.1 were used in the experiment. As shown from the results, using low thresholds leads to too many scaling actions, thus making the system behave unstably with thrashing. On the other hand, high thresholds provide few scaling actions, although it takes time for the system to respond to changes in workload.

As illustrated in Fig. 6, experimental observations indicate that $\vartheta = 0.05$ provides the most balanced operating point between responsiveness and stability under evaluated workload scenarios. These observations justify the threshold selected for DriGitScale and confirms that moderate sensitivity levels provide a practical balance between responsiveness and scaling stability. This value minimizes unnecessary scaling actions while maintaining acceptable responsiveness, demonstrating the importance of threshold tuning in controlling the trade-off between stability and reaction speed.

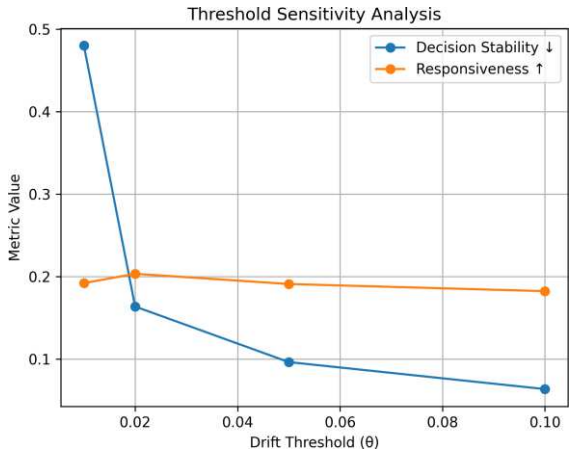


Fig. 6. Threshold sensitivity analysis showing the trade-off between stability and responsiveness.

IX. DISCUSSION

A. Impact of Prediction Models

The experimental results demonstrate that prediction accuracy alone is insufficient to guarantee effective autoscaling behavior. Although LSTM achieved low drift values across several experiments (Table II), it did not consistently provide the highest scaling stability. Similarly, lightweight approaches such as Moving Average (MA) exhibited competitive responsiveness while requiring substantially lower computational overhead.

These observations suggest that effective autoscaling depends not only on the quality of workload forecasting but also on how prediction outputs are translated into scaling decisions. Consequently, prediction performance should be evaluated alongside decision consistency and overall system stability rather than in isolation.

B. Role of Drift-Aware Decision Making

The incorporation of drift awareness resulted in consistent stability improvements across all evaluated datasets. As evident from Table III and Fig. 3, the adoption of drift-aware

decision-making results in fewer scaling operations (thrashing) than the baseline system. The drift-aware mechanism successfully filters transient fluctuations while remaining responsive to significant workload changes.

For example, under NASA workload traces, actual instability decreased from 0.1315 to 0.0097 after enabling drift-aware filtering. Similar improvements were observed across Wikipedia and synthetic workloads, demonstrating that consistency-aware decision making improves scaling stability beyond prediction quality alone. The observed reduction in instability demonstrates the effectiveness of drift-aware decision validation in filtering unnecessary scaling actions while preserving adaptation to significant workload changes.

These results indicate that incorporating prediction, resource, and configuration drift into the decision process leads to more stable autoscaling behavior. Unlike the existing solutions that use prediction information alone, the suggested solution considers consistency across the system, leading to improved stability.

C. Trade-offs Between Models

Different prediction models have different strengths and weaknesses:

- LSTM – strong ability to predict but slower inference time
- ARIMA – stable performance and predictions
- Moving Average (MA) – quick predictions and sensitive to noise
- Naive – easy implementation with limited adaptability

As illustrated in Table II, the results support the claim that our suggested approach is independent of model type and can be implemented depending on the need of the system. The key observation is that DriGitScale reduces dependence on the complexity of the underlying prediction model by validating scaling decisions through consistency-aware drift analysis.

D. Stability vs Responsiveness Trade-off

The experimental results reveal an important trade-off between responsiveness and stability. Drift-aware configurations significantly improved scaling stability by reducing unnecessary scaling actions; however, this improvement was accompanied by a modest increase in response latency.

The threshold sensitivity analysis further illustrates this relationship. Lower threshold values increase responsiveness but may trigger excessive scaling fluctuations, whereas higher threshold values improve stability at the cost of delayed reactions to workload changes. The selected threshold value ($\vartheta = 0.05$) provided the most balanced operating point across all evaluated datasets.

In practical cloud-native environments, a small increase in response latency is often preferable to frequent and unnecessary scaling oscillations, making stability-oriented decision strategies particularly valuable for production deployments.

E. Key Insight

Autoscaling must be considered a system consistency problem and not a mere prediction problem. The inclusion of drift awareness enables scaling decisions to remain consistent with the overall system state rather than relying exclusively on prediction outputs.

It changes the paradigm from:

Is the prediction accurate? ⟹ Is there system consistency?

F. Practical Implications

The experimental findings have several practical implications for deploying DriGitScale in cloud-native environments:

- The system minimizes the instability of the infrastructure due to scaling operations
- The system maximizes resource efficiency by minimizing scaling operations
- The system maintains state consistency using GitOps
- The system supports different types of models without being tied to any particular architecture

Overall, the experimental results demonstrate that combining predictive autoscaling with drift-aware consistency verification provides a practical and effective solution for improving autoscaling stability in cloud-native environments.

X. Threats to Validity

While the findings of the research indicate a promising direction, a number of threats to validity need to be considered in order to correctly interpret the findings and the limitations associated with the suggested framework.

A. Internal Validity

The experiment utilizes synthetic drift injection in order to simulate system dynamics. Nevertheless, despite using synthetic data, it cannot fully capture the complexity and variability that can be observed in actual production systems. Controlled workload fluctuations may introduce bias into the evaluation of responsiveness and stability metrics. Consequently, the observed results may not fully represent the behavior of the framework under all real-world operational conditions.

B. External Validity

CPU consumption is the principal scaling indicator considered in this study. On the other hand, autoscaling systems in practical applications are likely to consider many indicators besides CPU, such as memory consumption, network latency, and throughput.

Therefore, further validation using additional resource metrics and multi-dimensional workload characteristics is required to assess the generalizability of the proposed framework across diverse production environments.

C. Construct Validity

The assessment of system performance is conducted through parameters like drift score, stability, responsiveness, and inference time. While the above measures are appropriate in measuring system performance, they do not wholly depict how the application performs through factors such as latency, availability, and user experience.

Consequently, the evaluation does not directly capture end-user quality-of-service indicators such as application latency, service availability, and user-perceived performance.

D. Model Limitations

Models chosen for predictions (Naive, Moving Average, ARIMA, and LSTM) have different levels of complexity and aim at showing that the framework is agnostic to models used in predicting.

More sophisticated prediction frameworks, however, like Transformer-based models or hybrid models, have not been considered. These models may provide additional forecasting capabilities and could further improve autoscaling performance within the proposed framework.

E. Deployment Constraints

The execution approach that relied on GitOps was tested in an experimental setting. Although GitOps-based actuation improves deployment consistency, practical deployments may introduce additional delays due to repository synchronization, network latency, and reconciliation cycles of GitOps tools such as ArgoCD. These operational factors were not explicitly modeled in the current experimental environment and therefore represent an area for future investigation.

XI. Conclusion and Future Work

This work addressed a fundamental limitation of existing predictive autoscaling approaches for cloud-native environments, where high forecasting accuracy alone is insufficient to ensure stable and consistent runtime autoscaling behavior. Runtime inconsistencies among workload predictions, resource allocation, and deployment configurations may lead to unnecessary scaling actions, resource inefficiency, and reduced operational reliability.

To overcome these limitations, DriGitScale was proposed as a drift-aware and GitOps-enabled self-adaptive autoscaling model that reformulates autoscaling as a consistency-management problem rather than solely a workload prediction problem. By jointly evaluating prediction drift, resource drift, and configuration drift, DriGitScale validates scaling decisions against multiple dimensions of system behavior before execution, enabling more reliable and stable autoscaling.

Experimental results demonstrated that the proposed model improves scaling stability while maintaining acceptable responsiveness under different workload scenarios. In addition, GitOps-based synchronization preserves consistency between desired and actual deployment states, reducing configuration drift during runtime adaptation.

Overall, the presented approach demonstrates that reliable cloud-native autoscaling depends not only on accurate workload prediction but also on maintaining consistency across prediction outputs, runtime resource allocation, and deployment configurations. These findings provide a practical direction for designing more dependable self-adaptive autoscaling mechanisms for modern Kubernetes-based environments.

Future work will focus on extending DriGitScale by incorporating multi-metric autoscaling using additional runtime indicators such as memory utilization, network latency, and request rate, investigating more advanced prediction models including transformer-based architectures, evaluating the model on production-scale Kubernetes clusters, developing adaptive drift-threshold mechanisms, and exploring reinforcement learning techniques to further optimize scaling decisions through continuous runtime learning.

References

[1] T. Theodoropoulos, et al., "GraphOpticon: A global proactive horizontal autoscaler for improved service performance and resource consumption," *Future Generation Computer Systems*, vol. 174, 107926, 2026.

[2] B. Kumar, A. Verma, and P. Verma, "Optimizing resource allocation using proactive scaling with predictive models and custom resources," *Computers and Electrical Engineering*, vol. 118, 109419, 2024.

[3] D. Vu, M. Tran, and Y. Kim, "Predictive hybrid autoscaling for containerized applications," *IEEE Access*, vol. 10, pp. 104505–104519, 2022.

[4] G. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, pp. 559–592, 2014.

[5] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling techniques in cloud computing: A survey," *Future Generation Computer Systems*, vol. 33, pp. 1–15, 2014.

[6] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[8] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*, 5th ed. Hoboken, NJ, USA: Wiley, 2015.

[9] M. Armbrust, et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[10] A. Verma, et al., "Large-scale cluster management at Google with Borg," in *Proc. EuroSys*, 2015, pp. 1–17.

[11] B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[12] Kubernetes, Horizontal Pod Autoscaler, [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[13] J. Gama, I. Z˘liobaite˙, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Computing Surveys*, vol. 46, no. 4, pp. 1–37, 2014.

[14] A. Tsymbal, "The problem of concept drift: definitions and related work," Tech. Rep., Trinity College Dublin, 2004.

[15] F. Beetz and S. Harrer, "GitOps: The evolution of DevOps?," *IEEE Software*, vol. 39, no. 4, pp. 70–75, 2021.

[16] IEEE, "Standard for DevOps implementation within software development life cycles," IEEE Std 2675-2021, 2021.

[17] NASA Ames Research Center, HTTP Server Workload Dataset, [Online]. Available: https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/

[18] Wikimedia Foundation, Wikipedia Web Traffic Dataset. [Online]. Available: https://dumps.wikimedia.org

[19] H. Ahmad, C. Treude, M. Wagner, C. Szabo, "Towards resource-efficient reactive and proactive auto-scaling for microservice architectures," *J. Syst. Softw.*, 112390, 2025.

[20] J. Dogani and F. Khunjush, "Proactive auto-scaling technique for web applications in container-based edge computing using federated learning model," *J. Parallel Distrib. Comput.* 187, 104837, 2024.

[21] H. X. Nguyen, S. Zhu, and M. Liu, "Graph-PHPA: graph-based proactive horizontal pod autoscaling for microservices using LSTM-GNN," in *2022 IEEE 11th International Conference on Cloud Networking (CloudNet)*, 2022, pp. 237–241.

[22] T. Theodoropoulos, et al., "Intelligent proactive fault tolerance at the edge through resource usage prediction," arXiv preprint arXiv:2302.05336, 2023.

[23] Y. Zhou, et al., "Graph neural networks: Taxonomy, advances, and trends," *ACM Trans. Intell. Syst. Technol.* 13 (1), pp. 1–54, 2022.

[24] S. Kakade, et al., "Proactive horizontal pod autoscaling in kubernetes using bi-lstm," in *2023 IEEE International Conference on Contemporary Computing and Communications (InC4)*, vol. 1, 2023, pp. 1–5.

[25] J. Dogani, F. Khunjush, and M. Seydali, "K-agrued: a container autoscaling technique for cloud-based web applications in kubernetes using attention-based gru encoder-decoder," *J. Grid Comput.* 20 (4), 40, 2022.

[26] T. Theodoropoulos, et al., "Multi-service demand forecasting using graph neural networks," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2023, pp. 218–226.

[27] L. Ju, P. Singh, and S. Toor, "Proactive autoscaling for edge computing systems with kubernetes," in *Proc. 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2021, pp. 1–8.

[28] N. Marie-Magdelaine and T. Ahmed, "Proactive autoscaling for cloud-native applications using machine learning," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, 2020, pp. 1–7.

[29] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Trans. Netw. Serv. Manag.* 18 (1), pp. 958–972, 2021.

[30] M. Imdoukh, I. Ahmad, and M. G. Alfailakawi, "Machine learning-based auto-scaling for containerized applications," *Neural Comput. Appl.* 32 (13), pp. 9745–9760, 2020.

[31] E. Rossi, et al., "Temporal graph networks for deep learning on dynamic graphs," arXiv preprint arXiv:2006.10637, 2020.

[32] T. Wang, S. Ferlin, and M. Chiesa, "Predicting CPU usage for proactive autoscaling," in *Proc. 1st Workshop Mach. Learn. Syst.*, 2021, pp. 31–38.

[33] M. Abdullah, et al., "Burst-aware predictive autoscaling for containerized microservices," *IEEE Trans. Services Comput.*, vol. 15, no. 3, pp. 1448–1460, 2022.

[34] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of kubernetes pods," in *Proc. IEEE/IFIP Netw. Oper. Manag. Symp.*, 2020, pp. 1–5.

[35] P. Singh, P. Gupta, and K. Jyoti, "TASM: Technocrat ARIMA and SVR model for workload prediction of web applications in cloud," *Cluster Comput.*, vol. 22, no. 2, pp. 619–633, 2019.

[36] L. Baresi, D. Hu, G. Quattrocchi, and L. Terracciano, "KOSMOS: Vertical and horizontal resource autoscaling for kubernetes," in *Proc. Int. Conf. Service-Oriented Comput. (ICSOC)*, 2021, pp. 821–829.

[37] M. A. Tamiru, J. Tordsson, E. Elmroth, and G. Pierre, "An experimental evaluation of the kubernetes cluster autoscaler in the cloud," in *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2020, pp. 17–24.

[38] M. Tirmazi, et al., "Borg: the next generation," in *Proc. Fifteenth European Conference on Computer Systems (EuroSys '20)*, 2020.

[39] M. Zakarya, et al., "Perficientcloudsim: a tool to simulate large-scale computation in heterogeneous clouds," *J. Supercomput.* 77 (4), pp. 3959–4013, 2021.

[40] M. C. Silva Filho, et al., "CloudSim plus: A cloud computing simulation framework pursuing software engineering principles," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2017, pp. 400–406.

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

[42] A. Adegboyega, "Quantifying cloud workload burstiness: New measures and models," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manag. (IM)*, 2017, pp. 987–990.

[43] B. Shiva Prakash, et al., "A survey on recurrent neural network architectures for sequential learning," in *Soft Computing for Problem Solving: SocProS 2017*, Springer, 2019, pp. 57–66.

[44] J. Chung, et al., "Empirical evaluation of gated recurrent neural networks on sequence modeling," arXiv preprint arXiv:1412.3555, 2014.

[45] S. Wgbrant and V. D. Radic, "Automated network configuration: A comparison between ansible, puppet, and SaltStack," 2022.

[46] I. Korontanis, et al., "Real-time monitoring and analysis of edge and cloud resources," in *Proc. 3rd Workshop on Flexible Resource and Application Management on the Edge (FRAME 23)*, 2023, pp. 13–18.

[47] G. Rattihalli, et al., "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, 2019, pp. 33–40.

[48] M. Masdari and A. Khoshnevis, "A survey and classification of the workload forecasting methods in cloud computing," *Cluster Comput.*, vol. 23, no. 4, pp. 2399–2424, 2020.

[49] A. Verma, et al., "Failure detectors of strong s and perfect p classes for time synchronous hierarchical distributed systems," IGI Global, pp. 246–280, 2019.

[50] E. F. Coutinho, et al., "Elasticity in cloud computing: a survey," *Ann. Telecommun.* 70, pp. 289–309, 2015.

[51] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*, 2011, pp. 500–507.