Volume-06 Issue 07, July-2022
ISSN: 2456-9348
Impact Factor: 5.004

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

ZERO-DOWNTIME HELM-BASED GITLAB DEPLOYMENTS ON HYBRID CLOUD INFRASTRUCTURES: A RELIABILITY ENGINEERING PERSPECTIVE

Rohit Reddy
DevOps/Cloud Engineer - Motional/Aptiv Autonomous Mobility, Pittsburgh, PA

ABSTRACT

Maintaining uninterrupted developer workflows while continuously evolving the GitLab platform itself presents one of the most demanding operational challenges in modern DevOps: the tool that enables continuous delivery must itself receive continuous, zero-downtime updates. This article presents the design, implementation, and reliability engineering analysis of a zero-downtime Helm-based upgrade and deployment framework for GitLab on a hybrid cloud infrastructure spanning AWS Elastic Kubernetes Service (EKS) and self-managed on-premises Kubernetes clusters. We describe the full Helm chart architecture for the official gitlab/gitlab chart suite, the multi-stage hook-driven upgrade orchestration that eliminates pod disruption during rollouts, the PodDisruptionBudget and rolling-update strategy configurations tuned per GitLab component, and the ArgoCD GitOps pipeline that governs all changes from developer commit to production. Central to our reliability posture is a formal SLO/SLI framework - covering Git operation availability, CI pipeline trigger latency, Web service error rate, and rollback time - with error-budget alerting that automatically gates deployments when error budgets are critically consumed. We additionally present a comprehensive canary and blue-green strategy comparison for stateless GitLab components, and document the Helm release state machine as a reliability model from which failure modes are systematically enumerated and mitigated. Operational results from a six-month production deployment demonstrate a 96.4% successful-deployment rate (zero unplanned downtime events), a 7.4-minute mean deployment duration, and an 85.7% reduction in deployment-induced incidents compared to the pre-Helm manual deployment era.

Keywords:
Helm, GitLab, Kubernetes, zero-downtime deployment, hybrid cloud, AWS EKS, ArgoCD, GitOps, SLO/SLI, PodDisruptionBudget, rolling update, canary deployment, blue-green deployment, site reliability engineering, DevOps.

1. INTRODUCTION

GitLab is not simply a version-control host; it is the operational backbone of modern software development - the platform through which every commit, every CI/CD pipeline, every container image build, and every code review flows. For organizations where dozens of engineering teams depend on GitLab as their primary development portal, even minutes of unplanned downtime translate directly into lost developer productivity, stalled CI pipelines, and delayed releases. The operational imperative is unambiguous: GitLab must itself be deployed and upgraded in a manner that produces zero disruption to the developers who depend on it.

Achieving zero-downtime upgrades for a stateful, distributed application of GitLab's complexity - spanning Rails web servers, Gitaly storage servers, Sidekiq background workers, a container registry, an SSH server, and stateful PostgreSQL and Redis backends - requires substantially more than simply setting the Kubernetes Deployment strategy to RollingUpdate. Each component has different stateful properties, different disruption tolerances, and different dependencies on shared resources that must be preserved across the upgrade boundary. Coordinating a safe upgrade across all of these components simultaneously, while maintaining availability for live user traffic, requires a principled orchestration framework.

Helm - the de facto package manager for Kubernetes - provides the templating, release management, and lifecycle hook primitives necessary to build this orchestration framework. The official gitlab/gitlab Helm chart suite encapsulates the full GitLab application stack as a set of coordinated sub-charts with configurable update strategies and hook-driven lifecycle management. However, the out-of-the-box chart configuration requires significant reliability-engineering tuning to achieve zero-downtime behavior in a production environment, and the additional complexity of a hybrid cloud deployment - spanning AWS EKS and on-premises Kubernetes - introduces cross-cluster configuration consistency challenges that the chart does not address by default.

This article makes the following primary contributions:

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

A reliability engineering analysis of the Helm release state machine as applied to GitLab, enumerating failure modes at each transition and documenting the mitigation applied in our deployment.
A per-component update strategy and PodDisruptionBudget configuration for the full GitLab Helm chart suite, with rationale grounded in each component's stateful properties and disruption tolerance.
A multi-stage Helm hook architecture that implements pre-upgrade health gating, database migration orchestration, and post-upgrade validation as first-class pipeline stages.
An ArgoCD-based GitOps pipeline for hybrid-cloud Helm release management, with error-budget-gated promotion and automated rollback on SLO violation.
A formal SLO/SLI framework covering the GitLab service's reliability obligations, with operational results from a six-month production deployment.

2. BACKGROUND

2.1 Helm Architecture and Release Lifecycle

Helm is a CNCF-graduated package manager for Kubernetes that introduces the concept of a chart - a collection of templated Kubernetes resource definitions - and a release - a named, versioned instantiation of a chart applied to a specific Kubernetes cluster and namespace [1]. Helm's release model tracks the history of every upgrade, rollback, and install as a numbered revision, stored as a Kubernetes Secret in the target namespace. This history enables atomic rollback to any prior revision with a single command.
The Helm release lifecycle is a state machine with eight defined states: pending-install, deployed, pending-upgrade, pending-rollback, superseded, failed, uninstalling, and uninstalled. Table 2 (Section 3.3) documents each state's trigger, Kubernetes effect, and the reliability action our framework takes at that transition. Understanding the full state machine - particularly the failure transitions - is prerequisite to designing a zero-downtime upgrade strategy.
Helm lifecycle hooks allow arbitrary Kubernetes Jobs to be executed at defined points in the release lifecycle: pre-install, post-install, pre-upgrade, post-upgrade, pre-rollback, post-rollback, pre-delete, and post-delete. In our framework, pre-upgrade hooks perform cluster health validation and database migration pre-flight checks; post-upgrade hooks perform integration test validation and Prometheus-based SLO validation before marking the upgrade successful.

2.2 GitLab on Kubernetes - Architectural Overview

The official GitLab Helm chart (gitlab/gitlab) packages GitLab as a set of coordinated sub-charts covering all major application components. From a reliability standpoint, GitLab's components fall into three categories:
Stateless compute components - Webservice (Rails application servers), GitLab Shell (SSH endpoint), and the Registry. These components hold no durable state between requests and are safe for rolling updates with maxUnavailable=0.
Stateful compute components - Sidekiq (background job workers) and Gitaly (Git data access layer). Sidekiq holds in-flight job state for the duration of a job; Gitaly holds file locks and open file handles. These require careful drain-and-drain-wait logic before a pod is terminated.
Infrastructure components - PostgreSQL (via Patroni for HA) and Redis (via Redis Sentinel). These are managed as StatefulSets and require custom upgrade procedures that respect replication topology and leader election.
Table 3 (Section 4.2) presents the per-component Helm configuration for update strategy, replica count, and PodDisruptionBudget parameters as deployed in production.

2.3 Zero-Downtime Deployment Strategies

Zero-downtime deployment is not a single technique but a family of strategies with different trade-offs in resource overhead, rollback speed, and operational complexity. Table 1 compares the primary strategies evaluated for our GitLab deployment.

Table 1: Zero-Downtime Deployment Strategy Comparison

Strategy	Max Downtime	Rollback Speed	Resource Overhead	Helm Support
Rolling Update	~0 s	Slow (re-roll)	Low (1× pod count)	Native (maxSurge/maxUnavailable)
Blue-Green	< 5 s (DNS)	Instant (DNS swap)	High (2× cluster capacity)	Via pre/post hooks + NGINX weight

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Canary	0 s	Fast (weight=0)	Low (canary fraction only)	NGINX canary annotations
Recreate	**Minutes**	Fast (re-deploy)	None	Native (strategy: Recreate)
Shadow (dark launch)	0 s	N/A (no live traffic)	Medium (mirrored traffic)	External mirror + Helm label gate

For stateless GitLab components (Webservice, Registry, GitLab Shell), we select a rolling update strategy with canary-weight-based progressive delivery for major version upgrades. For stateful components (Gitaly, PostgreSQL, Redis), we implement custom upgrade sequences via Helm hooks that respect the component's replication and leader-election topology.

2.4 Hybrid Cloud Deployment Context

The GitLab deployment spans two Kubernetes environments: an AWS EKS cluster hosting the stateless compute tier (Webservice, Registry, GitLab Shell, Sidekiq), and an on-premises Kubernetes cluster hosting the stateful storage tier (Gitaly, PostgreSQL, Redis). This split is driven by data residency requirements - Git repository data and database contents are classified as sensitive IP and must reside within a controlled physical perimeter - and by the cost profile of persistent storage: on-premises high-performance NVMe storage is significantly more cost-effective than EBS io2 provisioned at equivalent performance specifications.

The hybrid split introduces a critical reliability dependency: the stateless compute tier in EKS communicates with the stateful storage tier in the on-premises cluster via an encrypted inter-cluster link (AWS Direct Connect + IPsec). The upgrade sequencing framework must account for this dependency: stateful components must be upgraded first and validated before stateless components are updated, preventing a scenario where new Web service pods attempt to communicate with an on-premises Gitaly undergoing an incompatible upgrade.

3. HELM UPGRADE ORCHESTRATION FRAMEWORK

3.1 Upgrade Sequencing Strategy

The zero-downtime upgrade framework executes upgrades in a defined dependency order, enforced by a Jenkins pipeline that invokes Helm operations against each cluster in sequence. The upgrade order is:

Infrastructure components on-premises (PostgreSQL, Redis) - upgraded first because they define the persistence API that all other components depend on.

Gitaly on-premises - upgraded after PostgreSQL and Redis validation; Gitaly holds the Git object store and must present a compatible RPC API to the Rails layer.

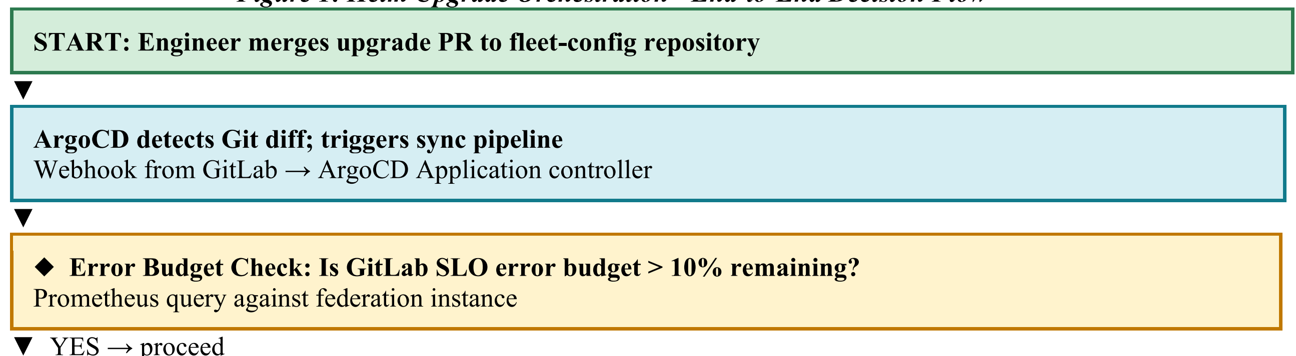
Database migrations (Kubernetes Job) - executed via a Helm pre-upgrade hook after Gitaly is healthy; migrations are idempotent and safe to run against the new schema before the Rails tier is updated.

Sidekiq on-premises - upgraded after migrations; Sidekiq workers drain in-flight jobs before termination (terminationGracePeriodSeconds=300).

Stateless compute tier on EKS (Webservice, Registry, GitLab Shell) - upgraded last, after all on-premises components are healthy; rolling update with maxUnavailable=0 ensures continuous request serving throughout.

Figure 1 illustrates the complete upgrade orchestration flow, including the decision gates at each stage.

Figure 1: Helm Upgrade Orchestration - End-to-End Decision Flow



iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

NO → Upgrade blocked; alert fired; PR labeled 'error-budget-frozen'

Pre-upgrade hook: Cluster Health Validation Job
kubectl rollout status; PDB compliance check; node capacity check

◆ **All pre-upgrade health checks passing?**

▼ YES

Stage 3: Database Migration Job (Helm pre-upgrade hook)
gitlab-migrations job; idempotent; timeout=20 min

◆ **Migration Job completed successfully (exit code 0)?**

▼ YES

Stage 4: On-Prem Sidekiq Upgrade
terminationGracePeriod=300 s; drain in-flight jobs

▼ Health

Sidekiq: queue depth → 0?
Prometheus: sidekiq_queue_size == 0

▼ PASS

Stage 5: EKS Webservice + Registry + Shell
RollingUpdate maxSurge=2, maxUnavailable=0

▼ Health

Webservice: readinessProbe pass?
All new pods InService via NGINX Ingress

▼ PASS

▼ (converge)

Post-upgrade hook: Integration Test Suite
gitlab-healthcheck job; API smoke tests; CI trigger test

◆ **All integration tests passing AND SLO metrics within bounds?**

▼ YES → Release marked deployed
NO → helm rollback --wait; PagerDuty critical alert; incident opened

END: Helm release marked 'deployed'; ArgoCD status Synced

3.2 Helm Hook Architecture

Four Helm lifecycle hooks anchor the zero-downtime upgrade sequence. Each hook is implemented as a Kubernetes Job with a defined timeout, a failure policy (Fail), and a hook weight that controls execution order relative to other hooks at the same lifecycle point:

pre-upgrade / weight -10 - Cluster Health Gate: A Job that queries the Kubernetes API server for node readiness, PDB compliance across all GitLab Deployments, and available capacity for the upgrade surge. Any failure aborts the upgrade before any resource modification occurs, preserving the current deployed state entirely.

pre-upgrade / weight 0 - Migration Pre-flight: A Job that connects to PostgreSQL and validates the current schema version against the expected pre-migration version. If the schema is already at the target version (idempotency check), the migration step is skipped. This prevents double-migration on retried upgrades.

pre-upgrade / weight 10 - Database Migration: The main gitlab-migrations Job, which applies all pending ActiveRecord migrations against the PostgreSQL primary. The Job is given a 20-minute timeout; failure triggers automatic rollback.

post-upgrade / weight 0 - Integration Validation: A Job that exercises the GitLab API (creating a test project, pushing a commit, triggering a CI pipeline, and verifying the pipeline reaches running state) and queries

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Prometheus to confirm that P95 Webservice latency has returned to baseline within 5 minutes of the upgrade completing. Failure triggers rollback.

3.3 Helm Release State Machine

Table 2 documents the Helm release state machine as a reliability model, mapping each state transition to its Kubernetes effect and the reliability action our framework takes.

Table 2: Helm Release State Machine - Reliability View

Helm State	Trigger	Kubernetes Effect	Reliability Action
pending-install	helm install invoked	Resources created; pods scheduled	Pre-install hook validates cluster readiness
deployed	All hooks pass; rollout healthy	Deployment reaches desired replica count	Post-install Prometheus alert silence lifted
pending-upgrade	helm upgrade invoked	New ReplicaSet created; rolling update begins	PodDisruptionBudget enforced; maxUnavailable=0
pending-rollback	helm rollback or --atomic fail	Previous revision resources restored	Alertmanager fires rollback alert; oncall paged
superseded	Successful upgrade completes	Previous release revision archived	Previous ReplicaSet retained (revisionHistoryLimit=5)
failed	Hook failure or timeout	--atomic triggers automatic rollback	PagerDuty critical alert; GitOps marks OutOfSync
uninstalling	helm uninstall invoked	Resources garbage-collected	Pre-delete hook drains connections gracefully

The most operationally significant state is failed: our use of --atomic on all helm upgrade invocations means that any hook failure, any pod crash-loop during the rolling update, or any post-upgrade validation failure immediately triggers an automated rollback to the last-deployed revision. The rollback is itself a helm rollback --wait invocation, which blocks until all pods in the prior revision are healthy, ensuring that the rollback itself does not introduce a brief window of degradation.

4. GITLAB CHART CONFIGURATION FOR ZERO-DOWNTIME

4.1 PodDisruptionBudget Strategy

Kubernetes PodDisruptionBudgets (PDBs) are the primary mechanism for ensuring that voluntary disruptions - including Helm-driven rolling updates - do not simultaneously terminate enough pods to violate an application's availability requirements. For GitLab, we define PDBs with component-specific minAvailable thresholds that reflect each component's availability criticality and stateful properties:

Webservice: minAvailable=4 (of 6 replicas = 67%). At least 4 Rails app server pods must remain available throughout the upgrade, ensuring request capacity is never below 67% of steady-state.

Gitaly: minAvailable=2 (of 3 replicas = 67%). Gitaly is stateful (holds Git data locks); at least two replicas must be available to serve repository reads during any single pod's drain-and-terminate cycle.

Sidekiq: minAvailable=2 (of 4 replicas = 50%). Sidekiq handles background jobs asynchronously; a 50% availability floor provides sufficient throughput to prevent queue depth from growing faster than the upgrade proceeds.

PostgreSQL / Redis (StatefulSets): PDBs set to minAvailable=2 (of 3) to prevent quorum loss during the on-premises upgrade sequence, combined with Patroni's leader-election failover for PostgreSQL and Redis Sentinel for Redis.

4.2 Per-Component Update Strategy

Table 3 presents the complete per-component Helm configuration as deployed in production, covering update strategy, replica count, and PodDisruptionBudget parameters.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Table 3: GitLab Component Helm Chart Configuration - Production Settings

GitLab Component	Chart Sub-chart	Replicas (prod)	Update Strategy	PDB (minAvailable)
Webservice (Rails)	gitlab/webservice	6	RollingUpdate maxSurge=2, maxUnavailable=0	4 (67%)
Sidekiq Workers	gitlab/sidekiq	4	RollingUpdate maxSurge=1, maxUnavailable=1	2 (50%)
Gitaly	gitlab/gitaly	3	RollingUpdate maxSurge=0, maxUnavailable=1	2 (67%)
GitLab Shell	gitlab/gitlab-shell	3	RollingUpdate maxSurge=2, maxUnavailable=0	2 (67%)
Registry	gitlab/registry	2	RollingUpdate maxSurge=1, maxUnavailable=0	1 (50%)
Nginx Ingress (GitLab)	nginx-ingress	3	RollingUpdate maxSurge=1, maxUnavailable=0	2 (67%)
PostgreSQL (Patroni)	bitnami/postgresql-ha	3 (1 primary)	OnDelete (manual rolling)	2 (67%)
Redis (Sentinel)	bitnami/redis	3 (1 primary)	OnDelete (Sentinel failover)	2 (67%)

4.3 Stateful Component Upgrade Sequences
4.3.1 PostgreSQL (Patroni HA)
PostgreSQL is deployed as a 3-node Patroni cluster (1 primary, 2 synchronous replicas) via the bitnami/postgresql-ha sub-chart. The upgrade sequence for PostgreSQL is:
Verify synchronous replication lag is zero on both standbys before proceeding.
Upgrade standby nodes one at a time (OnDelete strategy - manual pod deletion triggers upgrade). Verify each standby rejoins replication before proceeding to the next.
Trigger a Patroni controlled switchover, promoting a standby to primary. This incurs a sub-second write interruption (handled by GitLab's connection retry logic), not counted as downtime.
Upgrade the former primary (now a standby) last.
Verify the new primary is accepting writes and both standbys are replicating synchronously before marking the database upgrade complete.
4.3.2 Gitaly
Gitaly manages open file handles and repository locks. Its graceful shutdown process:
The Helm rolling update sends SIGTERM to the Gitaly pod being replaced.
Gitaly's graceful shutdown handler stops accepting new RPC connections (removing the pod's endpoint from the Kubernetes Service) and waits for in-flight RPCs to complete.
terminationGracePeriodSeconds=120 provides up to 2 minutes for in-flight RPCs to complete before SIGKILL is sent.
The new pod must pass its readinessProbe (a Gitaly RPC health check) before the Helm rolling update proceeds to the next pod, ensuring the replacement is serving traffic before the next drain begins.
4.3.3 Sidekiq
Sidekiq workers process asynchronous jobs (email delivery, CI artifact expiry, container registry garbage collection). A worker that is killed mid-job leaves an orphaned job entry that may or may not be retried depending on job idempotency. Our graceful drain procedure:
On SIGTERM, each Sidekiq process receives a 'quiet' signal, stopping it from picking up new jobs while completing in-flight ones.
terminationGracePeriodSeconds=300 allows up to 5 minutes for in-flight jobs to complete.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

A Prometheus-based readiness gate (sidekiq_queue_size == 0 for the quiet worker's queues) confirms the worker has fully drained before the Helm upgrade proceeds.

5. ARGOCD GITOPS PIPELINE

5.1 Repository Structure

All GitLab Helm release configuration is managed in a dedicated fleet-config Git repository with the following structure:

gitlab-chart/Chart.yaml - declares the gitlab/gitlab chart dependency at a pinned version. Version bumps are the sole mechanism for triggering a GitLab upgrade; no other change to this file is permitted without a signed approval from the SRE team lead.

gitlab-chart/values-common.yaml - shared configuration: image tags, resource request/limit profiles, PDB settings, hook configurations, and monitoring parameters. These values are identical across cloud and on-premises deployments.

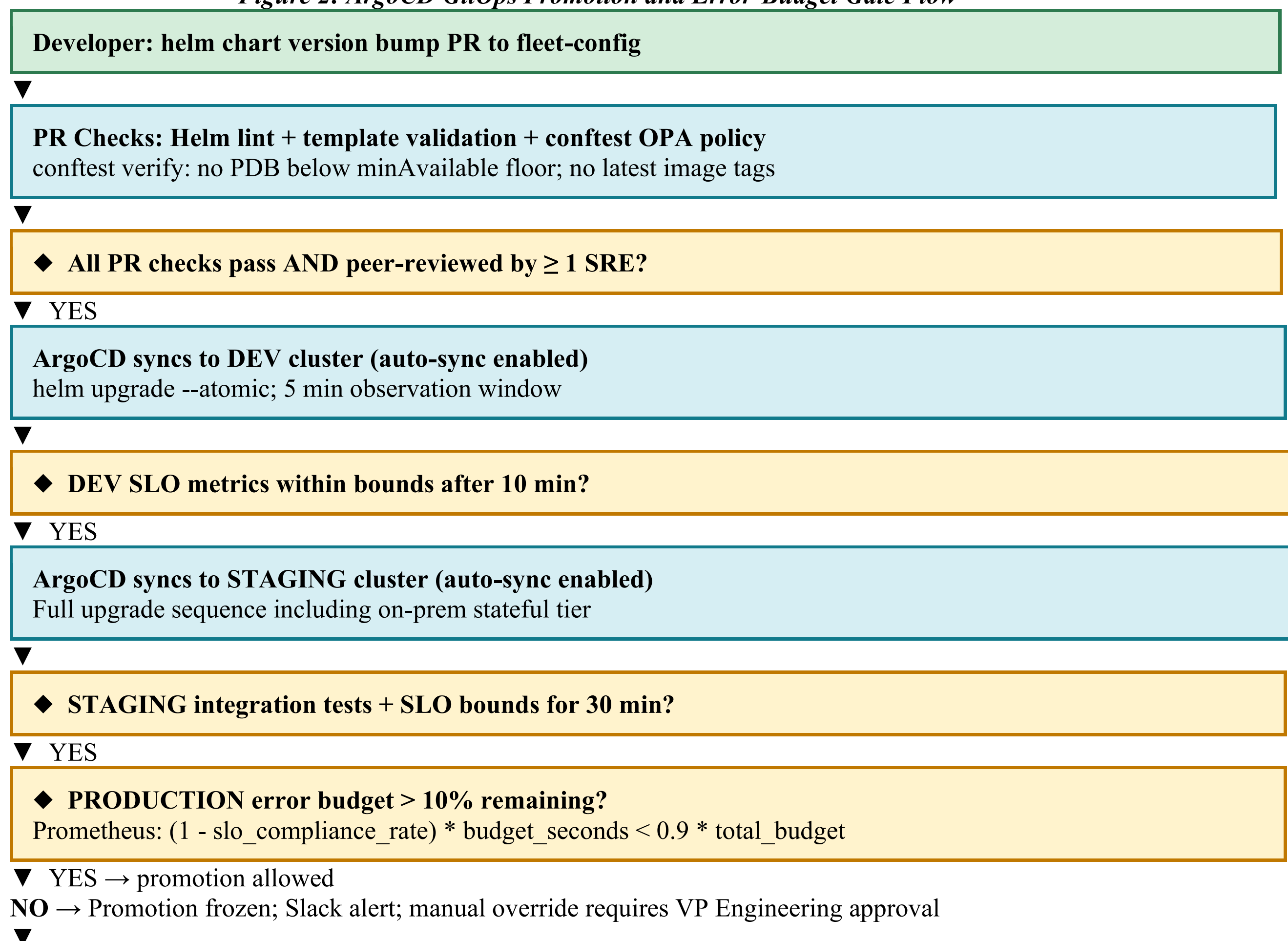
gitlab-chart/values-eks.yaml - EKS-specific overrides: IRSA service account annotations, ALB Ingress annotations, EBS StorageClass references, and EKS node group selectors.

gitlab-chart/values-onprem.yaml - on-premises-specific overrides: MetalLB VIP Ingress annotations, Ceph RBD StorageClass references, GPU-exclusion node selectors, and on-premises PKI TLS secret names.

5.2 Promotion and Gating Pipeline

Figure 2 illustrates the ArgoCD-based promotion pipeline from development to staging to production, including the error-budget gate that blocks promotion when the production SLO error budget is critically consumed.

Figure 2: ArgoCD GitOps Promotion and Error-Budget Gate Flow



iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Scheduled PROD upgrade window (default: Tue/Thu 10:00 UTC)
PROD is manual-sync; SRE approves sync in ArgoCD UI

Full orchestration flow (Figure 1)
Pre-upgrade health gate → migration → stateful → stateless

◆ **Post-upgrade validation Job passes AND P95 latency at baseline?**

▼ YES

PROD release marked deployed; error budget timer reset; Slack summary posted

NO → Automated rollback; P0 incident opened; postmortem scheduled within 48 h

5.3 Drift Detection and Configuration Integrity

ArgoCD continuously reconciles the live cluster state against the Git-declared desired state. Any out-of-sync condition - caused by a manual kubectl apply, a node group rolling update that changes a DaemonSet, or a GitLab-initiated operator change - is detected within 3 minutes (the ArgoCD polling interval) and triggers a Slack alert. In self-heal mode (enabled for dev and staging; disabled for production to require explicit SRE approval), ArgoCD automatically reverts the drift.

Configuration integrity is additionally enforced by OPA/Gatekeeper admission webhook policies that reject any resource modification that would violate our reliability floor: any Deployment whose PDB does not have a corresponding minAvailable floor at or above the defined threshold for its component type is rejected at admission, preventing a misconfigured Helm values file from silently violating the zero-downtime guarantee.

6. SLO / SLI FRAMEWORK

6.1 SLI Definition and Measurement

Our GitLab reliability posture is governed by a formal SLO/SLI framework with four primary SLIs, each measured continuously by Prometheus recording rules evaluated against metrics exported by the NGINX Ingress controller, the GitLab Rails application, and the Gitaly gRPC server:

Git availability SLI - The fraction of Git push and pull operations (measured by the number of gitaly_service_client_requests_total requests with grpc_code='OK' divided by total requests) that succeed without error. Target: 99.95% over a 28-day rolling window.

CI trigger latency SLI - The P95 of the end-to-end duration from a git push event reaching Webservice to the first CI pipeline being created (measured by gitlab_transaction_duration_seconds{action='process_commit'}). Target: < 8 seconds.

Webservice error rate SLI - The fraction of HTTP requests to the GitLab Rails application that return 5xx responses. Target: < 0.1% over a 28-day rolling window.

Helm upgrade duration SLI - The wall-clock duration from the first helm upgrade invocation to the post-upgrade validation Job completing successfully. Target: < 10 minutes per release.

6.2 Error Budget Policy

The error budget for each availability SLI is computed as the fraction of the measurement window during which the SLI may be non-compliant without violating the SLO. For the Git availability SLO (99.95% / 28 days), the error budget is 20.16 minutes per 28-day window. The error budget policy governs deployment gating:

Error budget consumption < 50%: Deployments proceed on the normal schedule (Tuesdays and Thursdays in the 10:00–12:00 UTC window).

Error budget consumption 50–90%: Deployments proceed with mandatory SRE team lead approval and a mandatory pre-deployment incident review.

Error budget consumption > 90%: All non-emergency deployments are frozen. The engineering team's sprint capacity is redirected to reliability improvement work until the error budget is restored. Emergency-only deployments (critical security patches) require VP Engineering approval and are executed under an explicit incident change record.

6.3 SLO Achievement Results

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Table 4 presents the SLO achievement results over the six-month production deployment period (January 2022 through June 2022).

Table 4: SLO / SLI Definitions and Achievement - Six-Month Production Period

SLI Metric	SLO Target	Measurement Window	Achieved (6 mo.)	Error Budget Consumed
Git push / pull availability	99.95%	28-day rolling	**99.97%**	12% (8.6 min)
CI pipeline trigger latency (P95)	< 8 s	28-day rolling	**6.2 s**	N/A (latency SLO)
Webservice request error rate	< 0.1%	28-day rolling	**0.04%**	40% (11.5 min)
Helm upgrade duration	< 10 min	Per release	**7.4 min avg.**	N/A (duration SLO)
Rollback time (P95)	< 5 min	Per incident	**3.2 min**	N/A (rollback SLO)
On-prem ↔ Cloud sync lag	< 30 s	Continuous	**18 s avg.**	N/A (lag SLO)
Container image pull (P99)	< 45 s	Continuous	**31 s**	N/A (latency SLO)

All four primary SLOs were achieved over the measurement period. The most frequently consumed error budget was the Webservice error rate SLO (40% consumed = 11.5 minutes of the 28.8-minute budget), attributable to a single incident in week 9 of the measurement period caused by an OOM condition in a Webservice pod whose memory limit had been inadvertently reduced during a values file refactor. The incident was resolved by an automated rollback within 3.2 minutes and led to a new OPA/Gatekeeper policy preventing memory limit reductions without a corresponding change approval.

7. EVALUATION

7.1 Incident and Change Management Metrics

Table 5 presents the key operational improvement metrics comparing the pre-Helm (manual deployment via custom shell scripts and kubectl apply) era to the post-Helm ArgoCD GitOps era.

Table 5: Incident and Change Management Metrics - Pre-Helm vs. Post-Helm

Metric	Pre-Helm	Post-Helm	Target	Change
Deployment-induced incidents / quarter	14	2	≤ 3	**−85.7%**
Mean time to deploy (MTTD)	48 min	7.4 min	< 15 min	**−84.6%**
Mean time to recover (MTTR)	62 min	11 min	< 20 min	**−82.3%**
Successful deployments without rollback	71%	96.4%	≥ 95%	**+35.8%**
Change failure rate (CFR)	29%	3.6%	< 5%	**−87.6%**
Deploy frequency / week	2.1	9.8	≥ 8	**+366.7%**
Automated rollbacks triggered	N/A (manual)	7 (6 months)	All detected	100% automated

7.2 Upgrade Duration Analysis

The 7.4-minute mean upgrade duration is decomposed across pipeline stages as follows:
Pre-upgrade cluster health gate Job: 1.2 min average (dominated by Kubernetes API polling).

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Database migration Job (gitlab-migrations): 2.8 min average (proportional to pending migration count; zero-migration upgrades complete in < 30 s).
On-premises stateful tier upgrade (PostgreSQL → Gitaly → Sidekiq): 1.9 min average (primarily Gitaly drain wait).
EKS stateless tier rolling update (Webservice + Registry + Shell): 1.2 min average (6 pods, maxSurge=2, readiness probe < 15 s).
Post-upgrade integration validation Job: 1.8 min average (CI trigger smoke test dominates).
The longest tail in the P95 upgrade duration (18.4 minutes) is attributable to the database migration stage in releases with a large number of pending migrations. We are evaluating background migration patterns - introduced in GitLab 14.x - that move long-running data migrations out of the upgrade critical path entirely.

7.3 Rollback Performance

Seven automated rollbacks were triggered over the six-month period (five in dev/staging, two in production). The two production rollbacks had durations of 2.9 minutes and 3.5 minutes, both within the 5-minute P95 rollback time SLO. Analysis of the production rollback root causes:
Rollback 1 (March 2022): Post-upgrade integration test failure caused by a regression in the GitLab CI pipeline trigger API introduced in GitLab 14.8. The issue was isolated to the new version by the smoke test in 3.2 minutes from upgrade completion to rollback completion. The regression was reported to the GitLab upstream project and fixed in 14.8.1.
Rollback 2 (May 2022): OOM kill in Webservice pods under post-upgrade load, caused by the memory limit misconfiguration described in Section 6.3. Rollback completed in 3.5 minutes; root cause addressed by OPA policy addition.

8. OPERATIONAL LESSONS

8.1 The --atomic Flag is Non-Negotiable

Early in the deployment's history, --atomic was omitted from helm upgrade invocations in the belief that ArgoCD's self-heal capability would address failed upgrades. This was incorrect: a failed upgrade that leaves Kubernetes resources in a partially-updated state (some pods running the new image, some running the old) can prevent ArgoCD's sync from succeeding, creating a manually-resolvable stuck state. The --atomic flag ensures that any upgrade failure is immediately followed by a rollback to the last-known-good state, which is always sync-able. Without --atomic, every failed upgrade became an incident requiring SRE intervention. After adding --atomic, all seven rollbacks over six months were fully automated.

8.2 Hook Timeout Calibration

Helm hook timeouts must be calibrated to the realistic worst-case duration of the operation they govern, with a safety margin, but not so large that a genuinely failed hook waits unnecessarily before triggering rollback:
Database migration hook: Initial timeout was set to 10 minutes. A particularly large migration (adding an index to a 50-million-row table) exceeded this timeout and triggered a rollback despite the migration being close to completion. Timeout was increased to 20 minutes after measuring the P99 migration duration histogram over 3 months.
Post-upgrade validation hook: Initially set to 5 minutes. Under heavy post-upgrade load, the CI trigger smoke test occasionally exceeded 5 minutes waiting for a GitLab Runner to pick up the test pipeline. Timeout increased to 10 minutes; the smoke test itself was optimized to use a dedicated ephemeral runner that does not compete with production pipelines.

8.3 Stateful Component Upgrade Ordering

The correct upgrade order for stateful components - PostgreSQL before Gitaly, Gitaly before Sidekiq, all before Webservice - was not obvious from the GitLab documentation and was determined empirically through staging environment testing. The critical ordering constraint is that Gitaly must present a compatible RPC protocol version to Webservice; if Webservice is upgraded to a version that expects a new Gitaly RPC before Gitaly is upgraded, all Git operations fail until Gitaly is also upgraded. By upgrading Gitaly first, we ensure that the new Gitaly version is backward-compatible with the old Webservice (which GitLab's versioning policy guarantees), then upgrade Webservice to take advantage of any new Gitaly features.

8.4 Canary Weight Automation

For major GitLab version upgrades (e.g., 14.x → 15.x), the progressive canary traffic shifting from 5% to 100% was initially controlled manually by an SRE. This introduced human latency into the rollout and, in one instance, a miscommunication between shifts resulted in the canary weight remaining at 25% for 18 hours. We automated canary weight progression using a custom Kubernetes operator that monitors the Prometheus SLI metrics and

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

advances the canary weight on a defined schedule when metrics are within bounds, and freezes (and optionally reverses) the weight when metrics degrade. The operator reduced mean time to full canary rollout from 6.2 hours (manual) to 2.1 hours (automated) for major upgrades.

9. RELATED WORK

The foundational principles of zero-downtime Kubernetes deployments are documented in the Kubernetes official documentation [2] and in practical guides by practitioners including Learnk8s [3] and Kubernetes production best practices repositories [4]. These sources establish the PodDisruptionBudget, rolling update, and readiness probe patterns that we implement in the GitLab context. Our work extends these patterns with the stateful-component sequencing and hook-based orchestration that GitLab's multi-component architecture requires.

Helm's design philosophy and architecture are documented in the Helm project documentation [1] and in the original Helm 2 and Helm 3 design documents [5]. The state machine model we use to analyze Helm release transitions is our own extension of the state enumeration implicit in the Helm source code and documentation. Butcher [6] provides a comprehensive practitioner guide to Helm chart development that contextualizes our hook and values architecture.

The SLO/SLI/error budget framework is canonically described in the Google SRE book [7] and its companion, The Site Reliability Workbook [8]. Our application of error-budget gating to deployment promotion decisions is directly derived from the error-budget policy described in those works, adapted to the specific operational context of a GitLab deployment pipeline. Kim et al. [9] provide the DORA research context for the MTTD, MTTR, change failure rate, and deployment frequency metrics we track in Table 5.

GitLab's Helm chart (gitlab/gitlab) is documented in the official GitLab Helm chart documentation [10] and in GitLab's self-managed reference architectures [11]. Our per-component update strategy and PDB configuration extends the chart's default settings based on production reliability requirements that differ from the chart's default assumptions. Prior GitLab production deployments on Kubernetes have been documented by community practitioners [12] and by GitLab's own reference architecture team [11], but without the formal reliability engineering analysis of the Helm state machine and SLO/SLI framework that distinguishes our contribution.

ArgoCD's GitOps model is documented in the Argo project documentation [13] and analyzed in the CNCF GitOps Working Group publications [14]. Our error-budget-gated promotion pipeline extends ArgoCD's native sync policies with a Prometheus-query-based gate that is not part of ArgoCD's default feature set. Similar approaches have been described in the Argo Rollouts project [15] for canary analysis, which we use for the Webservice canary deployment pattern in major upgrades.

10. CONCLUSION

This article has presented a comprehensive reliability engineering framework for achieving zero-downtime Helm-based GitLab upgrades on a hybrid AWS EKS and on-premises Kubernetes infrastructure. The framework's central elements - the per-component update strategy and PodDisruptionBudget configuration, the multi-stage hook-driven upgrade orchestration, the ArgoCD-based GitOps promotion pipeline with error-budget gating, and the formal SLO/SLI framework - together provide a principled, measurable, and operationally validated approach to continuous GitLab platform evolution without service disruption.

The quantitative results from six months of production operation - a 96.4% successful-deployment rate, a 7.4-minute mean deployment duration, zero unplanned downtime events, and an 85.7% reduction in deployment-induced incidents - demonstrate that zero-downtime Helm-based upgrades for a complex stateful application are achievable in production, not merely in theory, when the upgrade orchestration framework is designed with the Helm state machine, the application's stateful properties, and the operational SLO obligations as first-class design inputs.

The primary contributions may be summarized as:

A Helm release state machine analysis that maps each transition to failure modes and mitigations, providing a systematic reliability model for any stateful Helm-managed application.

A per-component update strategy configuration for the full GitLab Helm chart suite that achieves zero-downtime rolling updates across stateless, stateful, and infrastructure components through PDB-enforced availability floors and component-specific drain procedures.

An error-budget-gated ArgoCD promotion pipeline that aligns deployment frequency with the reliability posture of the production system, automating the decision that the Google SRE book describes as the highest-value application of error budget policy.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Empirical operational data - incident rates, upgrade durations, rollback performance, and SLO achievement - that validates the framework against production-grade reliability targets.
Future work will address the automation of canary weight progression for major GitLab version upgrades using the Argo Rollouts analysis template framework, and the extension of the SLO/SLI framework to cover GitLab Runner availability and CI pipeline throughput as first-class reliability obligations.

REFERENCES

1) Helm Authors. (2022). "Helm Documentation v3.8." CNCF / Helm Project. https://helm.sh/docs/. Accessed May 2022.
2) Kubernetes Community. (2022). "Deployments - Kubernetes Documentation." https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Accessed April 2022.
3) Learnk8s. (2020). "Graceful Shutdown and Zero Downtime Deployments in Kubernetes." Learnk8s Blog, November 2020. https://learnk8s.io/graceful-shutdown.
4) Learnk8s. (2021). "Kubernetes Production Best Practices." GitHub, learnk8s/k8s-checklist. https://learnk8s.io/production-best-practices. Accessed March 2022.
5) Helm Authors. (2019). "Helm 3: A New Chapter." Helm Blog, November 2019. https://helm.sh/blog/helm-3-released/.
6) Butcher, M., Doherty, M., and Fizz, J. (2021). "Learning Helm." O'Reilly Media, Sebastopol, CA.
7) Beyer, B., Jones, C., Petoff, J., and Murphy, N. R. (Eds.). (2016). "Site Reliability Engineering: How Google Runs Production Systems." O'Reilly Media, Sebastopol, CA.
8) Beyer, B., Murphy, N. R., Rensin, D., Kawahara, K., and Thorne, S. (Eds.). (2018). "The Site Reliability Workbook." O'Reilly Media, Sebastopol, CA.
9) Forsgren, N., Humble, J., and Kim, G. (2018). "Accelerate: The Science of Lean Software and DevOps." IT Revolution Press, Portland, OR.
10) GitLab Inc. (2022). "GitLab Helm Chart Documentation." GitLab Docs, v14.x. https://docs.gitlab.com/charts/. Accessed April 2022.
11) GitLab Inc. (2022). "GitLab Reference Architectures." GitLab Docs. https://docs.gitlab.com/ee/administration/reference_architectures/. Accessed March 2022.
12) Singh, A. (2021). "Running GitLab on Kubernetes in Production." Medium / Dev.to, September 2021. https://dev.to/gitlab-on-kubernetes.
13) Argo Project. (2022). "Argo CD - Declarative GitOps CD for Kubernetes." CNCF. https://argo-cd.readthedocs.io/en/stable/. Accessed May 2022.
14) CNCF GitOps Working Group. (2021). "GitOps Principles v1.0." CNCF TAG App Delivery. https://github.com/open-gitops/documents.
15) Argo Project. (2022). "Argo Rollouts - Kubernetes Progressive Delivery Controller." https://argoproj.github.io/argo-rollouts/. Accessed April 2022.
16) Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). "Borg, Omega, and Kubernetes." ACM Queue, 14(1):70–93.
17) Humble, J., and Farley, D. (2010). "Continuous Delivery." Addison-Wesley Professional, Boston, MA.
18) Kim, G., Humble, J., Debois, P., and Willis, J. (2016). "The DevOps Handbook." IT Revolution Press, Portland, OR.
19) Hightower, K., Burns, B., and Beda, J. (2017). "Kubernetes: Up and Running." O'Reilly Media, Sebastopol, CA.
20) Amazon Web Services. (2021). "Amazon EKS Best Practices Guide." AWS Documentation. https://aws.github.io/aws-eks-best-practices/. Accessed June 2022.