International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V72I11P124

Volume 72 Issue 11, 228-235, November 2024


*Original Article*

# From Planning to Rollback: Best Practices for Faster, Safer and Secure Kubernetes Deployments with SRE Principles

Praveen Chaitanya Jakku

*DevOps Engineer, Aubrey, TX, USA.*

*Corresponding Author : pcjakku@gmail.com*

Received: 11 October 2024 Revised: 12 November 2024 Accepted: 26 November 2024 Published: 30 November 2024

***Abstract -*** *This article presents a structured approach to Kubernetes release management inspired by Site Reliability Engineering (SRE) principles. It emphasizes balancing fast software releases with high reliability and availability, aiming for zero-downtime deployments even when issues arise. The approach includes incremental releases, canary deployments, blue-green deployments, and rolling updates to reduce risks and disruptions. It also highlights the importance of feature flags, automated CI/CD pipelines, and strong security practices. Additionally, the article discusses the role of robust monitoring, versioning, chaos engineering, and thorough documentation in optimizing the release process. By adopting these strategies, teams can deliver software efficiently, securely, and with minimal risk, ensuring a seamless experience for users and aligning with business goals.*

***Keywords -*** *Site Reliability Engineering (SRE), Kubernetes release management, Incremental releases, Parallel deployments, Feature flagging, Continuous Integration and Continuous Delivery (CI/CD), Security in kubernetes deployments, Image scanning, Role-Based Access Control (RBAC), Rollback strategies, Chaos engineering, Release prioritization.*

## 1. Introduction

Managing software releases in a Kubernetes environment can be challenging, especially when balancing the need for speed, security, and reliability. Many teams are turning to Site Reliability Engineering (SRE) principles to address these challenges to guide their release management processes. SRE helps organizations release software quickly and safely while maintaining the stability and performance of their systems. This article explores how you can apply SRE-inspired practices to Kubernetes release management. We will discuss strategies like incremental releases, parallel deployments, and feature flagging, which help reduce the risks of introducing large changes in production. We will also dive into automation techniques, including Continuous Integration and Continuous Delivery (CI/CD) pipelines, and the importance of strong security practices like image scanning and Role-Based Access Control (RBAC).

Additionally, we will cover how tools like health checks and chaos engineering can help make your releases more resilient while ensuring that monitoring and feedback loops allow you to improve your process continuously. By aligning your release management with Service Level Objectives (SLOs) and using a balance of safety measures, you can ensure that your Kubernetes deployments are stable, scalable, and prepared for anything that comes your way.

## 2. Release Management and Prioritization

### *2.1. Release Management*

Release management is the process of planning, scheduling, and overseeing the launch of new software updates, versions, or fixes. It involves coordinating with different teams to ensure everything is delivered on time and smoothly. This includes testing updates thoroughly and managing any risks to avoid problems. Good release management helps deliver bug fixes, security patches, and new features efficiently, improving the overall performance and user experience. It also helps reduce mistakes, minimize downtime, and ensure software updates are rolled out organizationally.

#### *2.1.1. Release Cadence*

Release cadence is the schedule for software updates. Some companies update their software often, like weekly or bi-weekly, while others do it less frequently, monthly or quarterly. The update frequency depends on the company's goals, resources, and the type of software. It is about how often users get new features, improvements, or bug fixes.

#### *2.1.2. Risk Assessment*

When you evaluate a software release's risks, you are looking for potential problems that could happen after the release.

*2.1.3. System Downtime*

The system might stop working for a while, disrupting operations.

*2.1.4. Security Issues*

The system could have weaknesses, making it easier for hackers to attack.

*2.1.5. User Experience Problems*

Changes might make the system harder to use, like confusing new features or annoying bugs.

*For example*

- Cluster Version Update: If a cluster is running outdated software, it is more vulnerable to attacks. Upgrading is crucial to maintain security.
- Code Changes Affecting Users: Introducing new code that impacts users comes with the risk that new features or fixes might not function as expected, leading to user issues.

***2.2. Release Prioritization***

Release prioritization decides which software updates are most important and should be released first. We consider two main factors:

*2.2.1. Urgency*

How quickly the update needs to be released, often due to issues like security vulnerabilities or critical bugs that need immediate attention.

*2.2.2. Impact*

How many users will be affected by the update, and how important the change is to them. For example, fixing a major feature many people use has a higher impact than a minor issue. By prioritizing updates based on urgency and impact, we ensure the most critical changes are addressed first, reducing risks and benefiting users the most.

*Example*

Issue-1: Cluster Version Update Due to Expiring Support.

- Risk: Security vulnerabilities from unsupported Cluster.
- Priority: High because security issues must be addressed immediately to protect the system.

Issue-2: Code Changes for Users.

- Risk: New bugs or disruptions in user experience.
- Priority: Medium, depending on how critical the changes are. If the changes affect many users, they may need to be prioritized higher.

***2.3. SLA and SLO***

*2.3.1. SLA*

An SLA (Service Level Agreement) is a contract that defines the level of service a provider promises to deliver to a customer. It sets clear expectations for how quickly issues will be resolved. For example, an SLA might specify that urgent security updates must be completed within 24 hours, while less critical issues can take longer. This helps the customer understand the service they can expect.

*2.3.1. SLO*

An SLO (Service Level Objective) is a specific target within an SLA that defines the performance standards the provider aims to achieve. For example, an SLO might state that the system should be available 99.9% of the time, meaning it can only be down briefly. SLOs help measure performance and ensure the service meets customer expectations.

*2.3.2. Example with SLA and SLO:*

- Security Patch: A security update might have an SLA of being deployed within 24 hours of detection, with an SLO achieving 99.9% uptime during the update.
- Code Changes for Users: A bug fix or feature release might have an SLA of a 3-5 day deployment window, with an SLO ensuring the update does not negatively affect user experience more than 1% of the time.

***2.4. Final Prioritization***

- Security updates, like the cluster version update, always take top priority because of the high security risk. These updates should meet strict SLAs and SLOs to minimize vulnerabilities.
- User-related changes, like bug fixes or new features, are important, too, but they can wait until security updates are finished. The urgency of these changes depends on how badly they affect users.

By using SLAs and SLOs, organizations can establish clear targets for how fast issues should be resolved and how well the system should perform. This ensures that the system remains secure, stable, and easy for users.

## 3. Incremental Releases

We can split big updates into smaller, manageable parts to avoid major issues when launching new software. This approach, known as incremental releases, lets us test changes gradually before making them available to everyone. By breaking updates into smaller chunks, we lower the risk of serious problems and can fix issues more quickly. Incremental releases provide faster feedback, easier rollbacks, and continuous testing, which helps improve software quality over time. Instead of releasing a large, complex update all at once, we roll it out in stages, starting with fewer users. This helps identify and fix issues early, ensuring the software stays stable.

***3.1. Here are some Techniques to Implement Incremental Releases***

*3.1.1. Feature Flags*

Feature flags are like switches for your software. They let you turn features on or off without updating the entire app.

This allows you first to test new features with a small group of users or quickly disable a feature if it causes issues. Think of them as a safety measure for your software releases.

*3.1.2. Canary Releases*

Think of canary releases as a small test group. You release a new version to a small subset of users or servers. If everything goes well, you gradually increase the number of users or servers receiving the new version. This minimizes the impact of any potential issues. Kubernetes is a popular container orchestration tool providing built-in strategies for canary deployments.

*3.1.3. A/B Testing*

Like a scientific experiment, A/B testing involves comparing two versions of a feature or application. We can route some user traffic to one version and the rest to another. We can determine which version performs better by analyzing user behaviour and feedback. Combining these techniques can minimize the risk of major disruptions and deliver high-quality software to our users.

## 4. Parallel Release Management: A Smooth Transition

Parallel Release Management is a strategy where different versions of software can run at the same time. If a new version has problems, the old version can still be used. It is like having a backup plan for your software. The main benefit is that it reduces downtime and minimizes risks for users. Businesses can test and deploy updates gradually, ensuring a smooth transition between versions. Here are three common strategies for parallel release management:

### *4.1. Blue-Green Deployment*

Imagine you have two identical houses: a blue one and a green one. You live in the blue house, but you want to renovate it. Instead of moving out, you start fixing up the greenhouse. Once the greenhouse is ready, you move in. You can quickly return to the blue house if there are any issues. This process is similar to a "blue-green deployment" in software. You maintain two identical versions of your software. You can switch from the old version (blue) to the new version (green) without downtime. This approach reduces risk and ensures a smooth update for your users.

### *4.2. Rolling Updates*

Instead of updating everything at once, you gradually update parts of your system. For example, if you have 10 servers, you might update a few at a time while the others keep running the old version. This ensures your system stays up and running throughout the update process.

### *4.3. Multiple Clusters*

This strategy involves using multiple clusters, often located in different places. You can test new updates on one cluster while the others continue to serve users. This way, if an update causes problems in one cluster, it does not affect the others. Businesses can deploy updates smoothly without interrupting service or affecting the user experience using these techniques. This makes their systems more reliable and easier to maintain.

## 5. Feature Branching and Versioning

We rely on two main practices to maintain stable and reliable software: feature branching and versioning. Feature branching allows us to separate work on different features by creating individual branches. This way, developers can work on their tasks without interfering with others. Versioning helps us track and label different software versions, making it easier to manage updates and revert changes when necessary. These practices help us develop and release new features efficiently while keeping the software stable.

### *5.1. Feature Branching*

Feature branching is a practice where each new feature or bug fix is worked on in a separate branch. This allows developers to develop and test new changes without affecting the main codebase (often called the *main* or *master* branch). Once the new feature is complete and fully tested, it can be merged into the main branch. To make it relatable, think of building a house. Instead of constructing the entire house in one go, you start with the foundation, build the walls, and finally put on the roof. In software development, feature branching works the same way. Each feature is like a separate part of the house, developed in isolation to not interfere with the rest of the structure. This approach ensures that new features are developed without disrupting the current system.

### *5.2. Versioning*

Versioning is the practice of keeping track of different versions of your software. It helps ensure that the correct software version is deployed to production and provides a clear way to track changes over time. Versioning also simplifies rollback if a new version introduces issues.Think of versioning like editions of a book. Each new version reflects a specific set of changes, whether bug fixes or new features. In Site Reliability Engineering (SRE), versioning is vital for managing software releases and makes identifying and controlling different software versions easier.

### *5.3. Why Feature Branching and Versioning Matter:*

*5.3.1. Improved Collaboration*

Feature branching enables developers to work on different tasks simultaneously without disrupting each other's progress. This enhances productivity and fosters better collaboration within the team.

*5.3.2. Better Testing and Stability*

By isolating new features in their branches, you can test them without disrupting the main code base, helping keep production stable.

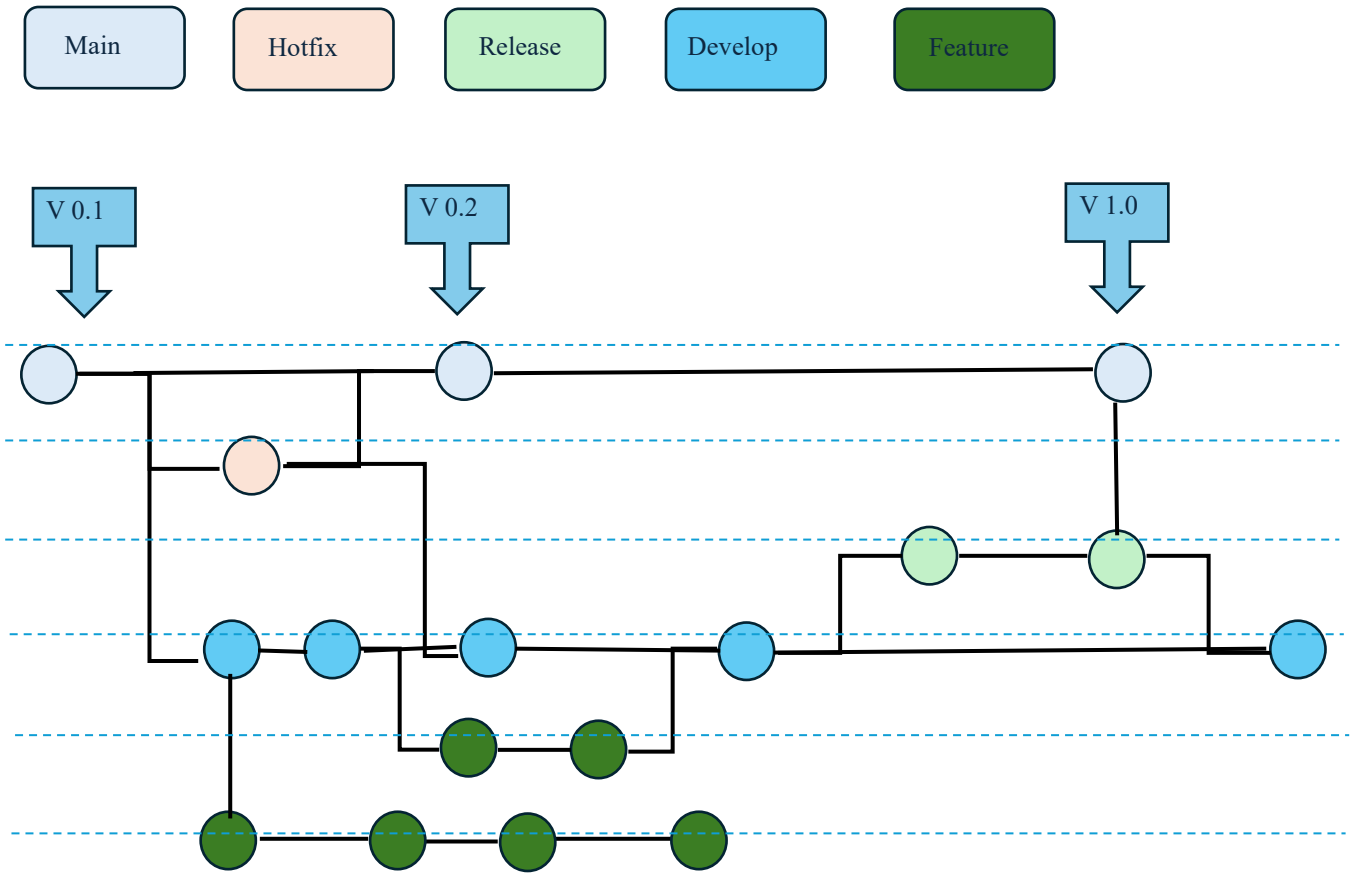


**Fig. 1 Example for Branching and Versioning**

*5.3.3. Simplified Rollbacks*

If a new version causes problems, you can quickly revert to a previous stable version, reducing downtime.

***5.4. Best Practices for Successful Implementation:***

*5.4.1. Automate Testing*

Use automated tests to check that new changes work as expected. This helps identify issues early in development, reducing bugs and errors when the software goes live.

*5.4.2. Establish Clear Versioning Standards*

Set clear guidelines for versioning so that all team members follow the same approach when managing software versions. This keeps everything organized and easy to track.

*5.4.3. Monitor Production*

Keep a close eye on the production environment to catch and resolve issues quickly. By effectively using feature branching and versioning, SRE teams can streamline development, improve software quality, and minimize downtime, leading to a smoother and more reliable deployment process.

## 6. Security Considerations

It is important to prioritize security and efficiency. Kubernetes is flexible and scalable, but this also means there are risks if it is not managed correctly. Following a sound security strategy and using Kubernetes' built-in features can reduce vulnerabilities and make your deployments faster and safer. Below are some best practices and tools that Site Reliability Engineers (SREs) can use to ensure more secure and reliable Kubernetes release management.

***6.1. Image Scanning and Signatures***

Before deploying your containerized applications to Kubernetes, ensuring that the Docker images used are free from known vulnerabilities is essential. By scanning container images before deployment, you can identify potential security issues in both the application code and any underlying libraries. Imagine you are building a house. Before you start building, you want to ensure the materials you use are safe and strong. Similarly, before you deploy your software to Kubernetes, you must ensure the "Docker images" (building blocks) are secure.

*6.1.1. How to do it*

- Scan the Images: Use tools like Trivy or Clair to check the images for security weaknesses. It is like inspecting the wood for termites or the wiring for faulty connections.
- Sign the Images: Sign the images with tools like Notary or cosign to ensure they have not been tampered with. It is like putting a seal on a document to prove its authenticity.

By following these steps, you can be confident that your software is built on a solid foundation and is protected from potential attacks.

### *6.2. RBAC (Role-Based Access Control)*

Role-Based Access Control (RBAC) is essential for managing access within Kubernetes clusters, ensuring that only authorized users and services can perform specific actions. Implementing strict RBAC policies helps restrict access to sensitive resources, minimizing the risk of unauthorized or harmful deployments.

For example, in a Kubernetes environment with multiple teams and roles, RBAC can be used to define and enforce permissions. Developers may be granted read-only access to production namespaces, while senior engineers or DevOps personnel would have the necessary permissions to manage deployments. This approach significantly reduces the risk of accidental or malicious changes that could affect the security or stability of the system.

### *6.3. Secrets Management*

Ensuring the security of sensitive data, such as API keys, passwords, and credentials, is vital for the integrity of your Kubernetes deployments. Kubernetes offers native support for Secrets objects, but it is important to follow best practices to manage them securely and avoid exposing sensitive information. For example, if your application needs to access an external API using an API key, avoid hardcoding the key directly into your application. Instead, store it securely as a Kubernetes Secret. You can create a Secret using the following command:

```
kubectl create secret generic my-api-key --from-literal=API_KEY=your-api-key-here
```

Once the Secret is created, your application can reference it securely, reducing the risk of exposing the key in your source code. Consider integrating a dedicated Secrets Management tool like HashiCorp Vault for enhanced security. Vault offers additional features such as dynamic secrets, automatic key rotation, and fine-grained access control, further strengthening the security of your sensitive data.

### *6.4. Automated Security Testing*

Adding security checks to your CI/CD pipeline is a smart way to catch vulnerabilities early and ensure your Kubernetes deployments are secure. Tools like Snyk or OWASP Dependency-Check help monitor your code and its dependencies for potential security issues. For example, whenever a new Docker image is built or updated in your CI/CD process, a tool like Snyk can automatically scan the dependencies for known vulnerabilities. If it finds any, the pipeline can stop the deployment, preventing any insecure code from being deployed to your Kubernetes cluster. This proactive approach ensures that vulnerabilities are detected early, reducing the chances of pushing faulty or unsafe code to production. It is insufficient to rely on Kubernetes' built-in security features to ensure secure Kubernetes releases. By incorporating best practices like image scanning, role-based access control (RBAC), secrets management, and automated security testing, SREs can enhance security while streamlining deployments. Combining these strategies with Kubernetes' native features helps keep applications secure and performant and reduces the risk of security breaches, leading to smoother and faster deployments.

## 7. Rollback Strategies for Zero-Downtime Deployments

Even with the best planning, issues may arise during a release. A clear and efficient rollback strategy ensures that production systems remain stable.

### *7.1. Automated Rollbacks*

Leverage Kubernetes' native deployment capabilities to automate rollbacks. If a release fails (e.g., pods crash, service health checks fail), Kubernetes can automatically revert to the previous stable version.

### *7.2. Manual Rollback and Approval*

In addition to automatic rollbacks, some teams may prefer a manual approval step, where a failed deployment triggers a notification to a responsible team for review and decision-making before a rollback is triggered.

### *7.3. Versioned Deployments*

Maintain multiple stable versions of the application (e.g., blue-green deployment or canary deployment). If the current version causes issues, you can quickly switch to the previous version, ensuring minimal disruption.

### *7.4. Monitoring and Alerts*

Set up robust monitoring and alerting systems (using tools like Prometheus, Grafana, or ELK stack) to quickly detect issues post-release. Monitoring helps identify performance degradation or failures early, allowing you to take corrective action before a full-blown issue occurs.

Having a solid rollback strategy is key to keeping systems stable during deployments. Automated rollbacks in Kubernetes can quickly fix issues, but manual approvals add an extra layer of control. Monitoring tools help catch problems early so teams can act fast to avoid bigger issues, ensuring a smooth and reliable deployment process.

## 8. Automated CI/CD Pipelines:

A well-designed Continuous Integration and Continuous Delivery (CI/CD) pipeline is crucial for efficiently and securely managing Kubernetes releases. Automation is key in minimizing human error and speeding up the deployment process.

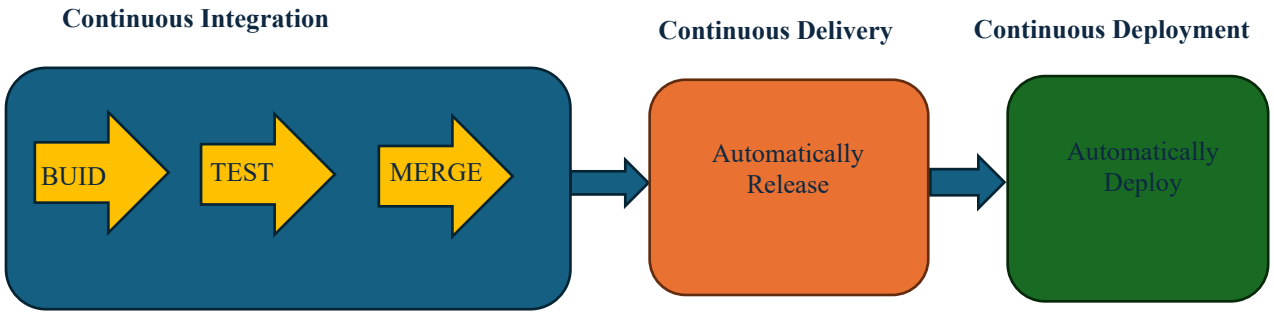


**Fig. 2 Example for Automated CI/CD Process**

### *8.1. Pipeline Automation*

Automate the entire process from code changes to testing and deployment. For example, using tools like Jenkins, GitLab CI/CD, CircleCI, or GitHub Actions can help streamline the workflow for deploying Kubernetes applications.

### *8.2. Test Automation*

Automate the testing process by including unit, integration, and end-to-end tests in your pipeline. Kubernetes-based environments (like development and staging) can be set up to test changes before they go live in production, ensuring everything works properly.

### *8.3. Continuous Deployment/Delivery*

Set up continuous deployment to automatically push changes to stage or production once the tests pass. This reduces manual intervention and allows you to release updates faster. For instance, after a successful test run, the code can be automatically deployed to a staging environment, and after additional validation, it can be pushed to production without delay.

## 9. Chaos Engineering

Chaos Engineering is about intentionally introducing failures into a system to understand how it behaves under stress and to ensure it remains reliable during unexpected situations. By applying this concept to Kubernetes release management, you can ensure that your deployments are resilient and capable of handling disruptions without affecting end users.

Here are some key ways to do this:

### *9.1. Simulate Failures*

Use tools like Gremlin or Chaos Mesh to create realistic failure scenarios, such as:

#### *9.1.1. Container Crashes*

Simulate situations where containers fail or stop running unexpectedly.

#### *9.1.2. Network Issues*

Test how your system responds to network partitions or latency.

#### *9.1.3. Service disruptions*

Introduce failures in backend services to observe how the system handles dependencies. These simulations help identify weak spots in your infrastructure before they affect your live systems.

### *9.2. Test in Staging*

Always test failures in a non-production environment first to avoid impacting real users. This helps ensure your system is ready for production disruptions.

#### *9.2.1. Automate Failure Testing*

Automate chaos experiments in your CI/CD pipeline to continuously test your system's resilience with each new release.

#### *9.2.2. Learn and Improve*

After each test, analyze what went wrong and improve your system over time. By integrating chaos engineering into your Kubernetes release management process, you can ensure that your infrastructure is prepared for the expected and resilient to the unexpected, leading to better user experiences and less downtime.

## 10. Monitoring and Observability Integration:

Monitoring and observability are essential tools for ensuring the success of Kubernetes releases. You can quickly identify and resolve issues by tracking your applications' health and performance, minimizing downtime and improving user experience.

### *10.1. Key Practices*

#### *10.1.1. Centralized Logging*

Gather logs from all parts of your Kubernetes system to pinpoint problems quickly.

#### *10.1.2. Distributed Tracing*

Track the flow of requests through your application to identify performance bottlenecks.

#### *10.1.3. Real-time Monitoring*

Continuously monitor key metrics like CPU usage, memory, and error rates.

*10.1.4. Automated Alerts*

Receive immediate notifications when issues arise, allowing for prompt response.

***10.2. Example***

Imagine you have just deployed a new version of your e-commerce application to Kubernetes. By using monitoring tools, you can:

*10.2.1. Identify slow load times*

If users are reporting slow page loads, distributed tracing can help you identify the specific service causing the delay.

*10.2.2. Detect increased error rates*

Real-time monitoring can alert you to a sudden spike in errors, allowing you to investigate and fix the issue before it impacts many users.

*10.2.3. Track resource usage*

Monitor CPU and memory usage to ensure your application performs efficiently and does not consume excessive resources. By effectively implementing monitoring and observability practices, you can significantly improve the reliability and performance of your Kubernetes deployments.

## 11. Post-Release Monitoring and Feedback Loops

Continuous improvement is a key tenet of SRE, and your release management process should include post-release monitoring and feedback loops.

***11.1. Post-Release Reviews***

After each release, conduct a post-mortem or release review meeting to analyze what went well and what did not. This helps you improve processes and avoid similar issues in the future.

***11.2. Continuous Feedback***

Use feedback from both users and internal stakeholders (via metrics, logs, or direct user reports) to identify and fix issues early after a release.

## 12. The Importance of Documentation

Effective documentation is a critical component of Kubernetes release management. While the technical aspects of deployment and monitoring are essential, clear and well-organized documentation ensures that everyone on the team is on the same page and can easily navigate the release process. It also plays a vital role in maintaining consistency, understanding, and accountability across the entire development and operations lifecycle.

***12.1. Clear Process Documentation***

Documenting the release management process, from planning and risk assessment to deployment and rollback strategies, helps ensure that all team members know their responsibilities and the steps involved in each release. This reduces confusion, improves collaboration, and minimizes errors due to miscommunication.

***12.2. Standard Operating Procedures (SOPs)***

Creating SOPs for different scenarios (e.g., canary releases, blue-green deployments, rollbacks, or incident response) ensures the team can handle various situations consistently and efficiently. This can significantly reduce the time needed to resolve issues and maintain production uptime.

***12.3. Knowledge Sharing***

Documentation provides a single source of truth for teams. It enables new team members to quickly understand the release process, configurations, and troubleshooting steps without relying on others for explanations. This helps onboard new members faster and builds long-term knowledge retention within the team.

***12.4. Audit and Compliance***

Proper documentation is vital for tracking changes, deployments, and issues, especially for audit or compliance purposes. It helps teams identify what was deployed when it was deployed, and what the associated risks or changes were. This can be essential for troubleshooting, reporting, or regulatory audits.

***12.5. Post-Release Reviews***

Documenting post-release reviews and feedback from previous deployments enables the team to learn from past experiences. This feedback loop helps continuously improve the release management process, refine practices, and adjust the deployment pipeline based on real-world insights.

Documentation ensures consistency, knowledge sharing, and efficient collaboration in Kubernetes release management. It empowers teams to execute deployments confidently and respond to issues swiftly while maintaining clear records of all activities.

## 13. Conclusion

By incorporating SRE principles into Kubernetes release management, teams can significantly enhance their deployment processes' reliability, efficiency, and security. This approach involves careful planning, incremental releases, and automated pipelines to minimize downtime and maximize user satisfaction. Key strategies include feature flagging, canary releases, and blue-green deployments to introduce new features and mitigate risks gradually.

Robust security measures, such as image scanning and RBAC, are essential to protect the system from vulnerabilities and unauthorized access. Effective monitoring and observability tools, like Prometheus and Grafana, enable teams to identify and address issues proactively. Moreover,

comprehensive documentation and knowledge sharing are crucial for maintaining consistency, facilitating onboarding, and streamlining troubleshooting. By embracing these practices, organizations can achieve a smooth and predictable deployment pipeline, delivering high-quality software that meets user needs and business objectives.

## References

[1] A Guide to Release Management: What it is, Why it's Important, and Best Practices, Instatus Blog, 2024. [Online] Available: https://instatus.com/blog/release-management

[2] Tram Tran, 5 Ways to Prioritize Your Software Release Backlog, DevOps digest, 2016. [Online] Available: https://www.devopsdigest.com/5-ways-to-prioritize-your-software-release-backlog

[3] David Usifo, The Difference Between Iterative and Incremental Development, Dee Project Manager, 2024. [Online] Available: https://deeprojectmanager.com/incremental-vs-iterative-development/

[4] Swati Khatri, How to Implement Blue-Green Deployment for Faster Releases, Index, 2024. [Online] Available: https://www.index.dev/blog/how-to-implement-blue-green-deployment-strategies

[5] SLA vs SLO: Tutorial & Examples, Squadcast, 2022. [Online] Available: https://www.squadcast.com/sre-best-practices/sla-vs-slo

[6] Jake Brereton, Mastering Git Branching Strategies for Software Development Success, Launch Notes, 2021. [Online] Available: https://www.launchnotes.com/blog/mastering-git-branching-strategies-for-software-development-success

[7] Harness Team, Software Rollback, Harness, 2024. [Online] Available: https://www.harness.io/blog/understanding-software-rollbacks

[8] A Guide to Understanding Observability and Monitoring in SRE Practices, SRE Fundamentals, Blameless, 2021. [Online] Available: https://www.blameless.com/blog/observability-and-monitoring

[9] Chaos Engineering: The History, Principles, and Practice, Gremlin, 2023. [Online] Available: https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-and-practice

[10] Isaac Sacolick, What Is CI/CD? Continuous Integration and Continuous Delivery Explained, InfoWorld, 2024. [Online] Available: https://www.infoworld.com/article/2269266/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html