# CI/CD Pipeline & Release/Rollback Plan

## Purpose and Scope

This document defines the continuous integration (CI) and continuous delivery/deployment (CD) pipeline for the PineCone Pro ERP/IMS project, along with release management and rollback strategies. It translates the architectural principles (microservices on AKS, Angular front‑end, ASP.NET Core back‑end, Docker and Helm) into a repeatable delivery workflow that emphasises security, quality and traceability. The plan covers all environments—development, QA/staging, UAT and production—and describes how code flows from a commit into a running service. It also outlines procedures for controlled releases and rapid rollbacks when something goes wrong.

The recommendations are drawn from industry best practices for microservices CI/CD pipelines and environment management. Microservice teams require isolated pipelines and independent versioning for each service, progressive delivery strategies such as canary or blue‑green deployments, strong observability, GitOps governance and security scanning【379199922569039†L131-L139】. Environments must be isolated, reproducible and documented【251501613233846†L83-L104】, with QA environments mirroring production and production prioritising high availability and disaster recovery【251501613233846†L130-L139】. Separating dev, UAT and prod environments improves stability, quality and security【984997393224141†L163-L169】.

## CI/CD Pipeline Overview

The pipeline follows a service‑scoped design: each microservice and the Angular front‑end have their own pipelines that can run independently【379199922569039†L156-L163】. Pipelines are defined as code (YAML) and stored in Git (GitOps) so all changes are traceable【379199922569039†L251-L257】. The high‑level stages are:

1. **Source Control & Branching.** Developers commit code to a Git repository using a trunk‑based workflow with feature branches. Pull requests trigger pipeline runs and enforce code reviews.
2. **Continuous Integration.** Pipelines fetch dependencies, compile code and execute unit tests and static analysis. Development environments are isolated and reproducible using Docker or containers and version‑controlled configuration【251501613233846†L120-L124】. Failure in any CI step blocks the pipeline.
3. **Container Build & Packaging.** Services are packaged as Docker images using multi‑stage builds. Images are tagged using [Semantic Versioning (SemVer)](https://semver.org/) and include Git commit SHA. Independent versioning allows each service to release without waiting for others【379199922569039†L192-L205】.
4. **Security Scanning.** Container images undergo vulnerability scanning and licence checks at build time. Detecting known CVEs early and shifting security left protects production【379199922569039†L279-L289】. Secrets are never hard‑coded; instead, Kubernetes Secrets or a vault (e.g. Azure Key Vault) manage credentials【379199922569039†L294-L299】.
5. **Artifact Registry.** Successfully built images are pushed to Azure Container Registry (ACR). ACR maintains an immutable history for rollbacks and supports automated image signing.
6. **Deployment to Non‑Production.** The pipeline uses Helm charts with environment‑specific values to deploy to development and QA clusters in AKS. QA environments mirror production to validate changes under realistic conditions【251501613233846†L130-L139】. Integration tests, API tests and performance tests run automatically. Only tested images are promoted.
7. **Promotion & Approvals.** Standardised templates promote services across Dev → QA → Prod【379199922569039†L259-L277】. Manual approvals enforce governance for production releases【984997393224141†L123-L136】. Role‑based access control ensures only authorized personnel can approve deployments【379199922569039†L244-L249】.
8. **Deployment to Production.** Production deployments use progressive delivery strategies. Blue‑green, rolling and canary deployments minimise downtime and enable fast rollbacks (see § Release Strategies). Deployment manifests include liveness/readiness probes, resource limits and horizontal pod autoscaler settings.
9. **Observability & Feedback.** Pipelines emit logs, metrics and traces for each build, test and deployment【379199922569039†L301-L320】. Centralised dashboards monitor build durations, failure patterns, promotion delays and environment‑specific behaviour. Alerts trigger on build failures, deployment rollbacks or breached SLAs. DORA metrics—lead time, deployment frequency, change failure rate and mean time to recovery (MTTR)—are tracked to gauge pipeline health【379199922569039†L343-L352】.

### Environment Considerations

| Environment | Purpose | Data & Configuration | Deployment Strategy |
| --- | --- | --- | --- |
| **Development** | Individual developers test code. | Synthetic or anonymised data; configuration managed in version control. Containers ensure reproducibility【251501613233846†L120-L124】. | All pipeline stages run, but deployments occur to isolated dev namespaces. Feature flags can be enabled for experimentation. |
| **QA / Staging** | Integration, performance and user acceptance testing. Mirrors production infrastructure and policies【251501613233846†L130-L139】. | Sanitised production data or seeded datasets【984997393224141†L147-L149】. Environment variables mirror prod. | Deploy using Helm with canary or rolling strategies. Automated test suites run; releases are validated. |
| **UAT** | Business stakeholders validate features. Usually shares staging cluster but with separate namespace and data sanitisation policies. | Sanitised production data; feature flags toggled to mimic production behaviour. | Deploy final release candidate; manual smoke tests. |
| **Production** | Live environment serving end users. Requires high availability, scalability, security and disaster recovery【251501613233846†L130-L139】. | Live data; strict configuration management and secrets handling. | Progressive deployment strategies (blue‑green, rolling, canary) with manual approval gates. Rollback plans are in place to minimise downtime. |

## Pipeline Stage Specifications

### Source Control & Branching

* All code resides in a Git hosting platform. Each microservice and the front‑end have separate repositories or sub‑directories to enable isolated pipelines【379199922569039†L156-L163】.
* Developers create feature branches from main and submit pull requests (PRs). PRs require code review and must pass CI checks before merging.
* Branch naming conventions (e.g. feature/<ticket>, bugfix/<ticket>) facilitate traceability.

### Continuous Integration

* The CI stage runs on commit and includes unit tests, static code analysis (linting, style checks) and dependency vulnerability scans. Failing tests break the build.
* Build pipelines run inside containers that replicate the target runtime environment. Isolation and reproducibility reduce “works on my machine” issues【251501613233846†L120-L124】.
* Test results and code coverage are published to dashboards. Flaky tests are flagged and require fixing.

### Build & Packaging

* Each service builds a Docker image using multi‑stage builds to minimize image size. The image includes all runtime dependencies and is tagged with a SemVer version (e.g. 1.2.0 for minor features, 1.2.1 for patch fixes and 2.0.0 for breaking changes)【379199922569039†L192-L205】.
* Version numbers are incremented automatically based on commit metadata and commit messages. Git tags map to container tags【379199922569039†L207-L213】.

### Security Scanning & Secrets Management

* A container image scanning step runs after each build to detect known vulnerabilities and configuration issues【379199922569039†L279-L289】. Only images with zero high‑severity issues proceed.
* Static application security testing (SAST) and software composition analysis (SCA) run during CI to identify insecure coding patterns or vulnerable third‑party libraries.
* Secrets (database passwords, API keys) are not stored in code or environment files. They are injected at runtime from Kubernetes Secrets or an external vault and accessed through service accounts with least privilege【379199922569039†L294-L299】.

### Artifact & Registry Management

* Images are pushed to Azure Container Registry (ACR) using the service’s unique repository (e.g. acr.example.com/pim-service). ACR enforces immutable images to enable version‑pinning.
* Helm charts are stored in an internal chart repository. Charts define Kubernetes objects (Deployments, Services, ConfigMaps, Ingress) and support values files for each environment.

### Deployment & Testing in Non‑Production

* **Development namespace**: On successful build, images are deployed into a developer’s namespace with auto‑provisioned databases and message queues. Developers can test against a full stack of services.
* **QA namespace**: After merging to main, the CI pipeline triggers a Helm deployment to the QA cluster. Automated integration and regression tests run. QA environments mirror production, ensuring that performance and scaling issues are detected early【251501613233846†L130-L139】.
* Promotion from dev to QA is automated and requires no manual intervention once tests pass. Failures notify the development team with logs and metrics.

### Promotion & Approvals

* Successfully tested builds are tagged as release candidates. A promotion pipeline deploys the candidate to the UAT and production clusters using standard templates【379199922569039†L259-L277】.
* Production releases require manual approval from an authorized release manager or product owner. The change management process, governed by project managers, reviews risk and scheduling【984997393224141†L123-L136】.
* All promotion and deployment activities are recorded in Git and the CI/CD system for auditability【379199922569039†L251-L257】.

### Progressive Deployment Strategies

Production deployments adopt one of three strategies depending on the risk profile:

1. **Blue‑Green Deployment.** Two identical production environments—blue and green—run side by side【440730242847635†L140-L168】. The green environment serves live traffic while the blue environment runs the new version. After tests on blue pass, traffic is switched by updating the load balancer. If the blue environment becomes unhealthy, the load balancer is pointed back to the green environment, enabling an immediate rollback with minimal downtime【440730242847635†L166-L169】【440730242847635†L175-L182】.
2. **Rolling Deployment.** A rolling update gradually replaces old instances with new instances【440730242847635†L196-L214】. Both versions run concurrently until the rollout completes. This approach requires only one additional node and keeps services available. If a failure occurs, traffic can be redirected to the old instances and the rollout halted【440730242847635†L219-L227】.
3. **Canary Deployment.** A canary release exposes the new version to a small percentage of users for early feedback【440730242847635†L241-L270】. Metrics such as latency and error rates are monitored【440730242847635†L265-L270】. If the canary performs well, it gradually replaces the old version; otherwise, traffic is rolled back quickly to the stable version【440730242847635†L271-L274】【440730242847635†L285-L289】. Feature flags can also implement canary releases by toggling functionality for selected users【379199922569039†L231-L237】.

### Observability & Feedback Loops

* Logs, metrics and traces are collected during each build, test and deployment stage【379199922569039†L301-L320】. Central dashboards show pipeline health, build durations, test coverage, failure patterns and environment‑specific behaviour.
* Alerts notify teams of build failures, test regressions, security scan findings or deployment rollbacks, enabling rapid response【379199922569039†L301-L320】.
* DORA metrics—lead time for changes, deployment frequency, change failure rate and MTTR—are used to continually improve the pipeline【379199922569039†L343-L352】.

## Release Management & Scheduling

Release management ensures that features are delivered predictably and with minimal disruption. Key practices include:

* **Release cadences.** Establish regular release windows (e.g. weekly or bi‑weekly) and freeze periods before major events. Emergency patches follow a separate expedited process.
* **Change control & governance.** Each release must be approved by a change advisory board or product owner according to the project’s governance policies【984997393224141†L123-L136】. The board reviews risk, compliance, and scheduling before deployment.
* **Release documentation.** Maintain release notes, migration scripts, and runbooks. Communicate changes to stakeholders and provide user training as needed.
* **Environment gating.** Releases progress sequentially through Dev, QA, UAT and Production. Automated quality gates (test results, security scans) and manual approval gates ensure only high‑quality releases are promoted【379199922569039†L259-L277】.

## Rollback Plan

Despite rigorous testing, issues can still occur. A documented rollback plan is essential for recovering quickly:

### Application Rollback

* **Using progressive strategies:**
  + **Blue‑Green:** Roll back by switching traffic back to the previous environment using the load balancer, restoring the old version within minutes【440730242847635†L166-L169】【440730242847635†L175-L183】.
  + **Rolling:** If issues appear mid‑rollout, stop the deployment and direct traffic to the remaining old instances【440730242847635†L219-L227】.
  + **Canary:** If canary metrics degrade, immediately route all users back to the stable version and deactivate feature flags【440730242847635†L271-L274】【440730242847635†L285-L289】. Since only a subset of users were affected, impact is limited.
* **Version pinning:** Always deploy immutable containers and pin Helm releases to a specific container tag. To roll back, redeploy the previous Helm release or container tag.
* **Automated rollback triggers:** Configure health probes and alerts to auto‑rollback if error rates, latency or CPU usage exceed thresholds.

### Database & Schema Rollback

* **Versioned migrations.** Database changes are applied using migration tools (e.g. Flyway or EF Core) with versioned scripts. Rollback scripts accompany each change and are tested in QA.
* **Backward compatibility.** Use additive schema changes (e.g. new columns or tables) before removing or altering existing structures. This enables rolling forward and backward without breaking older versions.
* **Backups and recovery.** Automated backups run before every production deployment. In the event of catastrophic failure, restore the database from the last snapshot and redeploy the previous version.

### Configuration & Secrets Rollback

* Configuration changes are stored in version control (values files, Kubernetes ConfigMaps). Rollback is as simple as re‑applying the previous configuration.
* Secrets are managed via Kubernetes Secrets or Vault; rotating secrets does not impact rollback as long as previous secrets remain valid until the rollback completes.

### Communication & Incident Handling

* **Notification:** When a rollback occurs, inform stakeholders via the incident communication plan. Provide status updates and root‑cause analysis once resolved.
* **Post‑mortem:** After recovery, document what went wrong, how it was detected, and actions taken. Feed learnings back into the pipeline by adding tests, monitoring or process improvements.

## Security & Compliance Considerations

* **RBAC & ownership.** Pipelines enforce fine‑grained role‑based access control so only authorized developers can modify pipeline definitions or deploy to specific environments【379199922569039†L244-L249】.
* **GitOps & audit trails.** Pipeline definitions and environment configurations are stored in Git. Every change is logged and traceable for auditability and compliance【379199922569039†L251-L257】.
* **Secrets management.** Sensitive data is never stored in code. Integration with Azure Key Vault or HashiCorp Vault ensures secrets are encrypted and rotated【379199922569039†L294-L299】.
* **Compliance scanning.** Security scans check for license violations and compliance issues. Pipeline steps enforce vulnerability fix SLAs before promotion to production.

## Tools & Implementation Technologies

* **Version control:** Git (GitHub or Azure DevOps Repos) stores code, pipeline definitions and Helm charts.
* **CI/CD orchestration:** Azure DevOps Pipelines or GitHub Actions run the CI and CD stages. Pipelines are written in YAML and use container‑based agents.
* **Container registry:** Azure Container Registry stores signed images. Each image is immutable and versioned.
* **Kubernetes & Helm:** AKS hosts all services. Helm charts manage deployments, services, ConfigMaps, Secrets and ingress. Helm releases map to versions, simplifying rollbacks.
* **Monitoring & Logging:** Azure Monitor, Application Insights and Prometheus/Grafana collect metrics and logs. Alerts integrate with Slack or Teams.
* **Service mesh (optional):** Istio or Linkerd can assist with traffic splitting for canary deployments and provide observability.

## Continuous Improvement

The pipeline is not static. Teams should continuously review DORA metrics, build durations and failure rates to identify bottlenecks and improve efficiency【379199922569039†L343-L352】. Feedback from post‑mortems and lessons learned from rollbacks should inform new tests, improved monitoring and refined deployment strategies.

## Conclusion

Implementing a robust CI/CD pipeline with progressive delivery and a documented release/rollback plan is critical to deliver PineCone Pro’s ERP/IMS with confidence. By isolating pipelines per microservice, enforcing semantic versioning, scanning for vulnerabilities, standardising templates, leveraging blue‑green/rolling/canary strategies and maintaining thorough observability, the team can deploy frequently and recover quickly from failures. Combined with strong governance, role‑based access control and environment separation, this pipeline will enable the organisation to deliver features rapidly while maintaining high quality and compliance.