Release management and branching policy

Audience: Internal engineers (mid-senior), tech leads, and release managers; comfortable with Git and CI/CD basics. **Job ID:** 2cf6c0cc-ff77-4446-8786-4db43ba43385 **Generated:** 2026-02-07

Table of Contents

1. Purpose, scope, and non-goals (`sec-01`)

1.1 Objective

You MUST use this policy to standardize how code becomes a production **Artifact**, balancing delivery speed, stability, and SOX-like auditability across all repositories that ship production changes.

1.2 Scope

This policy applies to:

- Any repository that builds and promotes a production **Artifact** (service, library, batch job, infrastructure module that results in deployed change).
- All changes that can affect production behavior, including code, configuration, database migrations, and dependency updates.

This policy does not replace:

- Incident response procedures (it constrains how **Emergency change** and **Hotfix** work within version control and release governance).
- Access management policy (it assumes least privilege and traceable identities).

1.3 Policy

1.3.1 Normative language

- Requirements in this document are stated using RFC 2119 keywords and MUST be followed when marked **MUST**.

- Recommendations are marked **SHOULD** and are expected unless a documented exception is approved.
- Optional practices are marked **MAY**.

1.3.2 Core governance requirements

- You **MUST** ensure all production changes are traceable from a ticket/change record to a Git commit, a pull request, and a promoted **Immutable artifact** version.
- You **MUST** use mandatory code review and **Branch protection** for all protected branches that can lead to production release.
- You **MUST** release using a cryptographically **Signed tag** (or an equivalent signed release attestation) for any production deployment.
- You **MUST** promote the same **Artifact** across environments (no rebuild-on-promote) to preserve provenance and auditability.
- You **MUST** perform production deployments only during defined change windows, except for an approved **Emergency change**.
- You **MUST** implement and follow a defined process for **Hotfix**, **Backport**, and **Rollback** (referenced in later sections of this policy set).

1.3.3 Definitions (normative for this policy)

- **Release**: A controlled promotion of an **Artifact** to production, identified by a version and a **Signed tag**.
- **Patch release**: A production release that increments the PATCH component of **SemVer** and contains only backward-compatible fixes.
- **Hotfix**: A targeted patch release to address a production issue with minimal scope and expedited timeline under heightened controls.
- **Emergency change**: A change executed under expedited controls to restore service or address critical risk; it is still subject to post-change review and audit completion.

  [!Guardrail] You MUST treat production deployability as a first-class constraint: `main` (or **Main/Trunk**) MUST remain releasable, and exceptions MUST be explicitly documented and time-bounded.

  [!Exception] Emergency change windows MAY be invoked only when the business impact of waiting for the next defined window exceeds the risk of expedited change; the approver and rationale MUST be recorded in the change record.

[!Pitfall] Rebuilding an artifact for staging vs production breaks provenance and invalidates audit trails; you MUST promote the same artifact digest/version across environments.

1.4 Rationale

- Auditability: SOX-like controls require end-to-end traceability (who changed what, who approved, what was deployed, and when).
- Stability: Mandatory reviews, protected branches, and signed releases reduce unauthorized or accidental production changes.
- Delivery performance: Standardized workflows reduce cognitive load and improve **Lead time for changes** while protecting MTTR and **Change failure rate**.
- Security: Promotion of immutable artifacts, plus required checks (defined later), supports supply-chain integrity (e.g., SLSA-aligned practices).

1.5 Examples / Checklist

1.5.1 Applicability checklist

You are in scope if you answer “yes” to any:

- Does this repo produce an **Artifact** that is deployed to production?
- Can changes here alter production behavior via configuration, feature flags, migrations, or dependency updates?
- Does this repo publish versioned packages consumed by production systems?

1.5.2 Minimum audit trail checklist (per production change)

You MUST be able to produce, on request:

1. A change record/ticket ID linked in the pull request description.
2. The pull request with reviewer identities, approvals, and passing required checks.
3. The merge commit (or equivalent) that introduced the change.
4. The build provenance identifying the **Artifact** version/digest.
5. The promotion history across environments to production.
6. The production release marker (a **Signed tag** or equivalent signed release artifact).

1.5.3 RACI overview (high level)

Activity	Responsible	Accountable	Consulted	Informed
Author code change and open PR	Developer	Engineering Manager	Security (as needed)	Team
Approve PR (code review)	Required reviewers / CODEOWNERS	Engineering Manager	QA/Test	Team
Approve production release (go/no-go)	Release Manager	Release Manager	Service Owner, Security	Stakeholders
Execute production deploy within window	Release Manager / On-call (per runbook)	Service Owner	SRE/ Operations	Stakeholders
Approve emergency change	Incident Commander / Service Owner	Service Owner	Security, Release Manager	Stakeholders
Post-change review completion	Service Owner	Service Owner	Release Manager, Security	Audit/ Compliance

1.5.4 Decision record: Release cadence variant (optional)

Decision: Teams MAY use either “continuous release” or a **Release train** for production releases.

Criteria

- Continuous release is appropriate when: automation coverage is high, risk is isolated via feature flags/canary, and change windows are frequent enough to meet delivery needs.

- Release train is appropriate when: coordination across multiple services is required, change windows are limited, or governance requires batching with a fixed **Cutoff** and **Stabilization** period.

Recordkeeping

- If using a release train, the cadence, cutoff rules, and stabilization rules MUST be documented per repo/service and referenced from the release runbook.

1.6 Principles

1.6.1 Objective

You MUST apply a consistent set of release principles to balance delivery speed, stability, ergonomics, and SOX-like auditability across all in-scope repositories.

1.6.2 Scope

This section applies to any repository that publishes an **Artifact**, promotes/deploys to production via CI/CD, or can materially affect production behavior. Internal tooling repositories **SHOULD** comply when they can affect production outcomes.

1.6.3 Policy

1. **Trunk-first**
    - You **MUST** treat `main` as releasable at all times unless an approved freeze exists.
    - You **MUST** keep branches short-lived and integrate frequently to `main`.
2. **Immutable artifacts and promotion**
    - You **MUST** build an immutable **Artifact** once per version and promote that exact version across environments without rebuilding.
    - You **MUST** ensure traceability from change request → code → review → build → test → promotion → deploy.
3. **Automation-first quality controls**
    - You **MUST** use CI/CD-enforced gates (tests, security scans, policy checks) for merges and promotions.

    - You **SHOULD** fail fast with small batches and quick feedback loops.

4. **Separation of duties and least privilege**
    - You **MUST** enforce mandatory code review and protected branches for `main` and release branches.
    - You **MUST** require signed releases (e.g., signed tag `vMAJOR.MINOR.PATCH`) for production promotion eligibility.
    - You **MUST** restrict production promotions/deployments to approved roles and change windows, except via the Emergency change path.

5. **Release vs deploy decoupling**
    - You **SHOULD** decouple deploy from feature exposure using configuration and **Feature flags**.

Guardrail
You **MUST** treat `main` as releasable at all times; any deviation requires an approved freeze with a recorded rationale and exit criteria.

Pitfall
You SHOULD NOT treat “release” and “deploy” as the same event; decouple exposure from deployment using configuration and Feature flags to reduce risk and improve rollback options.

Exception
Production changes outside defined windows **MUST** use the Emergency change process and record reason, approver, scope, rollback plan, and follow-up actions.

1.6.4 Rationale

- Trunk-first and short-lived branching reduce integration risk and improve lead time for changes.
- Immutable artifacts and promotion without rebuilding are required for reproducibility, auditability, and supply-chain integrity.
- Automation-first gates scale enforcement and reduce variance across teams.
- Separation of duties and signed releases create verifiable attestations suitable for SOX-like controls.
- Decoupling deploy from exposure reduces change failure rate and improves MTTR by enabling safer rollouts and rollbacks.

1.6.5 Examples/Checklist

- You can answer “yes” to each:
    1. Is `main` protected and kept releasable?
    2. Does every production change map to a reviewed PR and CI evidence?
    3. Is the production-deployed version an immutable Artifact promoted from lower environments?
    4. Is the release identified by a signed tag like `v1.2.3`?
    5. Are production changes limited to defined windows (or documented as Emergency changes)?
    6. Are risky changes guarded by Feature flags and reversible configuration?

1.7 Success metrics (SLIs/SLOs) for the release process

1.7.1 Objective

You MUST measure release process outcomes using standard delivery and reliability metrics to drive continuous improvement and to detect control weaknesses.

1.7.2 Scope

Applies to all teams and pipelines that ship production changes for in-scope repositories.

1.7.3 Policy

You **MUST** define, collect, and review (at least monthly) the following metrics:

Metric	Definition	Minimum requirement
Lead time for changes	Time from commit on `main` to running in production (or released)	You **MUST** measure and trend
Deployment frequency		You **MUST** measure and trend

Metric	Definition	Minimum requirement
	Production deployments per unit time	
Change failure rate	% of deployments/ releases causing incident, rollback, or hotfix	You **MUST** measure and trend
MTTR	Mean time to restore service after incident	You **MUST** measure and trend

You **MUST** also track quality gate effectiveness:

Quality signal	What you measure	Minimum requirement
CI pass rate	% of pipeline runs passing required checks	You **MUST** measure per `main` and release branches
Test coverage thresholds	Coverage % vs agreed threshold	You **SHOULD** set repo-specific thresholds with recorded rationale
SAST/DAST results	Findings count/severity and time-to-remediate	You **MUST** block promotion on policy-defined criticals
SBOM/provenance availability	Presence of SBOM and build provenance per Artifact	You **MUST** produce and retain for production releases

> **Guardrail**
> If you cannot produce evidence (logs, CI status, signed tags, approvals) for a release, you **MUST NOT** promote it to production.

1.7.4 Rationale

- Delivery metrics (lead time, deployment frequency) measure throughput and bottlenecks.
- Reliability metrics (change failure rate, MTTR) measure operational risk and recovery capability.

- Quality signals ensure controls are effective, not just present, and support audit inquiries with objective evidence.

1.7.5 Examples/Checklist

1. You publish a monthly release health report containing the four core metrics plus CI pass rate and security scan outcomes.
2. You can list the top three contributors to lead time (e.g., review time, flaky tests, manual approvals) and an improvement plan.
3. You can map a production incident to the originating deployment/release and associated Artifact version.

1.8 Decision record: Release trains (optional variant)

1.8.1 Objective

You MAY adopt release trains to bundle changes on a cadence while preserving trunk-first principles and auditability.

1.8.2 Policy

- You **MAY** use a release train if you can still:
  - **MUST** keep `main` releasable and continuously integrated.
  - **MUST** produce immutable Artifacts and promote without rebuilding.
  - **MUST** meet change window constraints and approval requirements.
- You **SHOULD** prefer continuous delivery (on-demand releases) when risk is low and automation maturity is high.

1.8.3 Criteria (choose and record)

Criterion	Choose release trains when…	Choose on-demand when…
Regulatory/change window constraints	Fixed windows force batching and formal go/no-go	Windows are frequent/continuous and controls are automated
Operational risk		

Criterion	Choose release trains when…	Choose on-demand when…
	Many cross-cutting changes require coordinated rollout	Changes are small, reversible, and well-instrumented
Dependency coordination	Multiple teams/ components must align	Components are loosely coupled with independent releaseability

1.8.4 Rationale

Release trains can improve predictability under strict windows and coordination needs, but they may increase batch size and delay feedback; the criteria ensure the choice is intentional and recorded.

1.8.5 Examples/Checklist

- Decision record includes: cadence, cutoff definition, stabilization rules, go/ no-go attendees, and rollback expectations.

1.9 sec-03 — Repository taxonomy and applicability

1.9.1 Objective

You MUST classify each repository into a standard taxonomy so you can apply consistent, auditable branching, release, and promotion controls while respecting real delivery differences (e.g., service vs library vs mobile app).

1.9.2 Scope

This section applies to any repository that:

- **MUST** publish a production Artifact, or
- **MUST** participate in CI/CD that deploys or promotes to production, or
- **MUST** be treated as production-impacting because changes can alter production behavior (including shared libraries, IaC/modules, and policy/ config repos).

Internal tooling repositories that do not directly impact production outcomes **SHOULD** follow the same controls where feasible (notably review, signing, and traceability), with documented deviations.

> [!Guardrail] `main` **MUST** be treated as releasable at all times unless an approved freeze exists.

1.9.3 Policy

1.9.3.1 1) Repository types (taxonomy)

Each repository **MUST** declare exactly one primary type in a tracked location (e.g., `docs/repo-metadata.yml` or repository settings), and **MAY** declare secondary traits (e.g., “monorepo”, “multi-artifact”).

Repo type	Typical artifact(s)	Primary consumers	What “release” means	Default branching posture	Notes
Service (deployable runtime)	Container image, VM image, function bundle	Production runtime	A signed, versioned publication of the deployable Artifact (e.g., signed tag `vMAJOR.MINOR.PATCH`) promotable across envs without rebuilding	Trunk-based with short-lived branches; release branches only for stabilization/hotfix	Deploy and exposure SHOULD be decoupled via config/ feature flags
Library/ SDK	Package (e.g., Maven/ NPM/ PyPI),	Other repos/ services	A signed, versioned package publicatio	More frequent use of release branches	Backports are common and

Repo type	Typical artifact(s)	Primary consumers	What “release” means	Default branching posture	Notes
	shared module		n (SemVer)	for patch lines	MUST be tracked
Infrastructure / IaC / Modules	Terraform module, Helm chart, policy bundle, config package	Platform/ runtime provisioning	A signed, versioned module/ chart/ policy package	Trunk-based with strict promotion gates; release branches used when supporting multiple runtime versions	Changes can be production-impacting even without “deploying code”
Mobile/ Desktop app	App bundle / installer	App stores / end users	A signed, versioned build submitted to distribution channel(s)	Trunk-based with store-track-based stabilization branches as needed	Store review and phased rollout constraints apply
Monorepo (multi-component)	Multiple artifacts from one repo	Mixed	A signed, versioned publication per artifact OR a coordinated train release	Trunk-based with directory ownership; optional release trains	Requires directory-based CODEOWNERS and consistent versioning rules

> [!Pitfall] Release and deploy SHOULD NOT be treated as the same event. You SHOULD deploy an immutable Artifact and control exposure with configuration or feature flags.

1.9.3.2 2) Applicability decision (in-scope vs lighter-weight)

A repository is “in scope” for full controls when the answer is “yes” to any of the following; you **MUST** record the decision (e.g., `docs/audit/applicability.md`).

Question	Yes ⇒ requirement impact
Does this repo publish an Artifact that can reach production?	Full policy applies
Does this repo have CI/CD that deploys or promotes to production?	Full policy applies
Can changes here alter production behavior (runtime, config, policy, infra, shared library)?	Full policy applies
Is it internal tooling only with no production impact?	Lighter-weight variant **MAY** apply, but reviews/signing **SHOULD** remain

> [!Exception] Production changes outside defined windows **MUST** use the Emergency change process and record reason, approver, scope, rollback plan, and follow-up actions.

1.9.3.3 3) Release semantics by repo type (what you MUST be able to prove)

Regardless of type, a “release” **MUST** be auditable and reproducible:

- It **MUST** map from change request → code change → review approvals → CI results → build provenance → signed tag → immutable Artifact version → promotion history → production deploy record.
- The released Artifact **MUST** be immutable and **MUST** be promoted between environments without rebuilding.
- Releases **MUST** be signed (e.g., signed tag `vMAJOR.MINOR.PATCH`), and promotion/deploy records **MUST** reference that release identifier.

Type-specific requirements:

- **Service repos:** a release **MUST** identify the exact container image digest (or equivalent) promoted to production.
- **Libraries/SDKs:** a release **MUST** identify the published package version and repository (e.g., internal registry) and **MUST** preserve patch-line support rules if maintained.
- **Infrastructure/modules:** a release **MUST** identify the module/chart/policy package version and the target environment(s) it was applied to, with change window compliance.
- **Mobile/Desktop apps:** a release **MUST** identify the signed build submitted, the distribution track/channel, and the rollout controls used (e.g., phased rollout), plus the production equivalence point (store availability).

1.9.3.4 4) Monorepo applicability (directory-based ownership and trains)

If a repo is a monorepo:

- You **MUST** implement `CODEOWNERS` with directory-level ownership for each production-impacting component.
- You **MUST** define whether releases are:
    1. per-component (independent versioning), or
    2. train-based (coordinated cadence).
- You **MUST** ensure traceability per artifact, not just per repo.

1.9.3.4.1 Decision record — Release train vs per-component releases (monorepo variant)

Option A: Per-component releases (default when components deploy independently)
Criteria:

- Choose when artifacts have independent deploy cadence, risk profiles, or rollback paths.
- You **SHOULD** use independent versioning and component-scoped pipelines.

Option B: Release trains (optional when coordination is required)
Criteria:

- Choose when artifacts must be released together due to coupling, compliance timing, or shared change windows.
- You **MUST** define cutoff rules, stabilization rules, and go/no-go roles for the train.

Record (example fields you MUST capture):

- Decision: `per-component` or `release-train`
- Rationale
- Cutoff schedule (if train)
- Scope of components included
- Exception handling (hotfix/backport rules)

1.9.4 Rationale

A clear taxonomy prevents “one size fits none” while preserving minimum SOX-like auditability. Different repository types legitimately require different stabilization patterns (e.g., patch lines for libraries, store tracks for mobile), but all must converge on the same traceability chain and immutable Artifact promotion model.

1.9.5 Examples / Checklist

1.9.5.1 Repository classification checklist (you MUST complete)

1. Identify primary repo type (Service / Library / Infrastructure / Mobile-Desktop / Monorepo).
2. Confirm applicability (answer the four questions in the applicability table).
3. Define “release identifier” format (e.g., `vMAJOR.MINOR.PATCH`) and signing method (signed tag).
4. Specify artifact coordinates:
    - Service: image name + digest
    - Library: package name + version + registry
    - IaC/module: module/chart name + version + registry
    - Mobile/Desktop: build number + signing identity + track/channel
5. Declare branching posture:
    - whether release branches are used, and when (stabilization, hotfix, patch lines)
6. Confirm ownership:
    - `CODEOWNERS` paths for production-impacting areas (monorepo: directory-level)
7. Confirm promotion model:
    - artifact is built once and promoted without rebuilding
8. Confirm production window constraints and emergency change path is referenced

1.9.5.2 Example: Applicability statement (template)

- Repo type: Service
- In scope: Yes (publishes container Artifact and deploys to production)
- Release definition: Signed tag `vMAJOR.MINOR.PATCH` mapping to immutable image digest promoted across environments without rebuilding
- Notes: Deploy decoupled from exposure using feature flags

1.10 sec-04 — Branching model overview

1.10.1 Objective

You standardize branch types and naming so changes are traceable from request → code → review → build → signed **Release** → **Promotion** of an **Immutable artifact** into production, with SOX-like auditability and separation of duties.

1.10.2 Scope

This section applies to any repository that builds/publishes a production **Artifact** or participates in CI/CD that deploys/promotes to production. Internal tooling repositories **SHOULD** comply when they can affect production outcomes.

1.10.3 Policy

1.10.3.1 1) Primary branch (`main`)

- `main` **MUST** be treated as releasable at all times unless a freeze is approved and recorded.
- Direct pushes to `main` **MUST** be blocked via Branch protection; changes **MUST** merge via PR with required checks (defined in CI/CD gates) and mandatory review approvals.
- `main` **SHOULD** remain green (all required checks passing); if it is red, you **MUST** prioritize restore-to-green before merging non-critical work.

  Guardrail `main` **MUST** remain releasable; “we’ll stabilize later” is not an acceptable default posture.

1.10.3.2 2) Short-lived feature branches (`feature/*`)

- Feature branches **MUST** be short-lived and merged via PR back into `main`.

- Naming **MUST** follow: `feature/<ticket>-<slug>` (example: `feature/PROJ-1234-add-invoice-export`).
- Feature branches **SHOULD** be deleted after merge to reduce audit surface and confusion.

  Pitfall Long-lived feature branches accumulate hidden integration risk and erode auditability of “what went live when.”

1.10.3.3 3) Release branches (`release/x.y`) — stabilization only

- Release branches **MAY** be used when you need a stabilization period while `main` continues to evolve.
- Naming **MUST** follow: `release/<major>.<minor>` (example: `release/1.8`).
- After cutoff, the release branch **MUST** accept only stabilization changes (bug fixes, risk reductions, release hardening). Features **MUST NOT** be added post-cutoff.
- Changes into `release/<major>.<minor>` **SHOULD** be applied via **Backport** (e.g., `git cherry-pick`) from commits already merged to `main` to preserve traceability and reduce divergence.
- Release tags **MUST** be created as signed tags: `vMAJOR.MINOR.PATCH` and must map to the commit that produced the promoted **Artifact**.

  Exception A long-lived `release/<major>.<minor>` branch may exist for extended support. This **MUST** be explicitly documented with ownership, support horizon, and backport policy in `docs/decision-records/branching-and-release.md`.

1.10.3.4 4) Hotfix branches (`hotfix/x.y.z-*`) — urgent production fixes

- Hotfix branches **MUST** be cut from the production tag that represents the currently running (or last known-good) production release, not from `main`.
- Naming **MUST** follow: `hotfix/<major>.<minor>.<patch>-<ticket>` (example: `hotfix/1.8.3-PROJ-8821`).
- A hotfix release **MUST** result in a new signed tag `vMAJOR.MINOR.(PATCH+1)` and a corresponding **Artifact** that is promoted without rebuilding across environments.
- Hotfix changes **MUST** be forward-ported to `main` and any active `release/<major>.<minor>` branch to prevent regression.

Guardrail Hotfix does not waive controls: mandatory review, required checks, signed **Release**, and auditable **Promotion** still apply (Emergency change is separate and requires its own record).

1.10.4 Rationale

- Keeping `main` releasable reduces lead time and integration failures while supporting continuous compliance evidence (reviews, checks, provenance).
- Short-lived feature branches minimize merge debt and preserve a clear audit trail.
- Release branches isolate stabilization work without blocking ongoing development, while backport-first reduces divergence and supports reproducibility.
- Hotfix branches from production tags preserve correctness and allow controlled, minimal-scope remediation with clean forward-porting.

1.10.5 Examples/Checklist

1.10.5.1 Branch types and naming (standard)

Branch type	Purpose	Lifecycle expectation	Naming convention	Allowed change types
Primary	Always-releasable integration line	Long-lived	`main`	Any change that passes gates
Feature	Develop a single change set	Short-lived	`feature/<ticket>-<slug>`	Feature work, refactors, tests
Release	Stabilize a specific minor line	Temporary (or documented LTS)	`release/<major>.<minor>`	Stabilization only; fixes via backport
Hotfix	Urgent prod patch	Very short-lived	`hotfix/<major>.<minor>.<p`	Minimal fix + tests +

Branch type	Purpose	Lifecycle expectation	Naming convention	Allowed change types
			`atch>-<ticket>`	release hardening

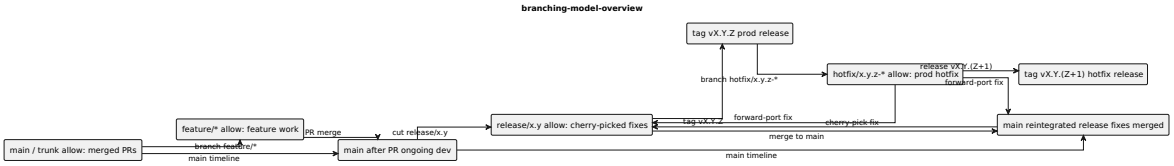
1.10.5.2 Naming examples

- `feature/PROJ-1234-add-invoice-export`
- `release/2.1`
- `hotfix/2.1.0-PROJ-9999`

1.10.5.3 Checklist: branch creation and merge hygiene

1. You **MUST** open a PR for any merge into `main`, `release/<major>.<minor>`, or `hotfix/*`.
2. You **MUST** ensure required checks are passing before requesting approval (CI, security scans, policy checks).
3. You **MUST** ensure merge commits/tags are attributable (review approvals, CI run links, and artifact provenance).
4. You **SHOULD** delete `feature/*` and `hotfix/*` branches after merge and tag creation.

1.10.6 Diagram — Branch types and lifecycle overview



1.10.7 Decision record (optional variants)

You **MAY** operate on-demand releases or release trains; you **MUST** document the choice and criteria in `docs/decision-records/branching-and-release.md`.

Variant	When it fits	Criteria you MUST record
On-demand releases (default)	Low coupling; frequent, independent releases	Expected deployment frequency, change

Variant	When it fits	Criteria you MUST record
		window alignment, required approvers, rollback strategy
Release trains	Multiple teams/ components need coordinated cadence	Train cadence, cutoff rules, stabilization window, exception handling, impact on lead time and change failure rate

1.11 sec-05 — Commit, PR/MR, and merge strategy

1.11.1 Objective

You **MUST** ensure every change is traceable (change request → code → review → build/test → Release → Promotion → production deploy), auditable (SOX-like), and compatible with automated CI/CD gates while keeping `main` releasable.

1.11.2 Scope

This policy applies to any repository that builds/publishes/promotes a production **Artifact** or participates in CI/CD that deploys/promotes to production. Tooling repositories **SHOULD** comply when they can affect production outcomes.

1.11.3 Policy

1.11.3.1 1) Commit message and metadata requirements

- You **MUST** use Conventional Commits (or an equivalent, documented standard) so automation can derive changelogs, versioning signals, and risk routing.
- Each change **MUST** include:
  - A traceable change request identifier (e.g., ticket ID) in the commit body and/or PR title.
  - Ownership metadata via `CODEOWNERS` and an explicit PR owner/ assignee.

    - A risk signal (label or field) that can be used to require additional approvals and gates.
- You **SHOULD** keep commits logically scoped and reversible; large refactors **SHOULD** be split into reviewable PRs.
- You **MAY** use multi-commit PRs during development; merge behavior (see below) defines how history is recorded on protected branches.

Guardrail
`main` **MUST** be treated as releasable at all times unless a freeze is approved and recorded (approver, scope, dates, and exit criteria).

Pitfall
“Drive-by” commits without a ticket, risk label, or review trail break auditability and can invalidate production promotion evidence.

1.11.3.2 2) PR/MR requirements (protected branches)

- You **MUST** use a PR/MR to merge into protected branches (at minimum `main`, and any `release/<major>.<minor>` branches).
- PRs **MUST** include:
    - Passing required CI status checks (at minimum lint + unit + integration as applicable).
    - Security checks (at minimum SAST + dependency/license scanning + SBOM generation; DAST where applicable and defined).
    - Required approvals per `CODEOWNERS`.
    - Evidence-ready audit trail: PR description links the change request, states risk, and states rollback considerations.
- PRs **MUST NOT** be merged while checks are failing or approvals are missing.
- PRs **SHOULD** include a changelog fragment when the repository publishes release notes (path and format defined per repo standards).
- PRs **MAY** require explicit QA sign-off based on risk labels (e.g., `risk:high`, `customer-impacting`, `data-migration`).

Guardrail
Branch protection **MUST** enforce: no direct pushes, required reviews, required status checks, and (where supported) signed commits/tags verification for release operations.

Exception
Emergency change procedures **MAY** bypass normal lead time, but you

MUST record reason, approver, scope, rollback plan, and follow-up actions, and you **MUST** reconcile to the standard process post-facto.

1.11.3.3 3) Merge strategy (what to use and when)

- Feature work merged to `main` **MUST** use squash merge (single commit on `main`) to produce a clean, review-aligned audit record.
- Release and hotfix branches **MAY** use merge commits when preserving the branch context is beneficial for audit, stabilization, and forward-port/backport bookkeeping.
- You **MUST NOT** rebase shared branches (any branch that another person or CI pipeline consumes). You **MAY** rebase locally before opening a PR to reduce noise, provided the PR remains reviewable and traceable.

1.11.3.4 4) Backport and cherry-pick protocol

- Backports **MUST** be performed via `cherry-pick` from `main` (or from the original merged PR commit) into the target `release/<major>.<minor>` branch.
- You **MUST** label backport PRs consistently (e.g., `backport`, plus target version label such as `backport:1.6`).
- The backport PR **MUST** reference:
  - The original PR and commit SHA(s).
  - The rationale for backporting (risk/impact).
  - Any divergence from the original (if conflicts required adjustments).
- Hotfix changes cut from a production tag **MUST** be forward-ported to `main` (and to any active `release/<major>.<minor>` branch) after the production fix is released.

Pitfall
Re-implementing a fix manually instead of `cherry-pick` removes cryptographic linkage to the reviewed change and complicates audit evidence.

1.11.4 Rationale

- Standardized commits and PR metadata enable deterministic audit trails, predictable automation, and repeatable evidence for compliance (who approved what, when, and under which checks).
- Enforced checks and `CODEOWNERS` approvals provide separation of duties and reduce change failure rate.

- Squash merges keep `main` readable and align the recorded change with the reviewed PR, while merge commits remain available for stabilization contexts where branch topology is meaningful.
- Formal backport/cherry-pick rules preserve traceability between fixes shipped across versions and reduce regression risk.

1.11.5 Examples/Checklist

1.11.5.1 PR author checklist (minimum)

1. You **MUST** create a branch named `feature/<ticket>-<slug>` (or the repo's approved equivalent).
2. You **MUST** ensure the PR title/description references the ticket and includes a risk label.
3. You **MUST** ensure required checks pass (CI + security + policy checks).
4. You **MUST** obtain `CODEOWNERS` approvals; you **MAY** obtain QA sign-off when triggered by risk.
5. You **MUST** choose the approved merge method:
    - Squash merge for features to `main`.
    - Merge commit allowed for `release/<major>.<minor>` stabilization and `hotfix/<...>` integration when required.
6. You **MUST** confirm the destination branch is protected and the merge will produce an auditable record.

1.11.5.2 Example Conventional Commit (illustrative)

- `feat(api): add idempotency key support\n\nRefs: TICKET-1234\nRisk: medium\nOwners: @team-api`

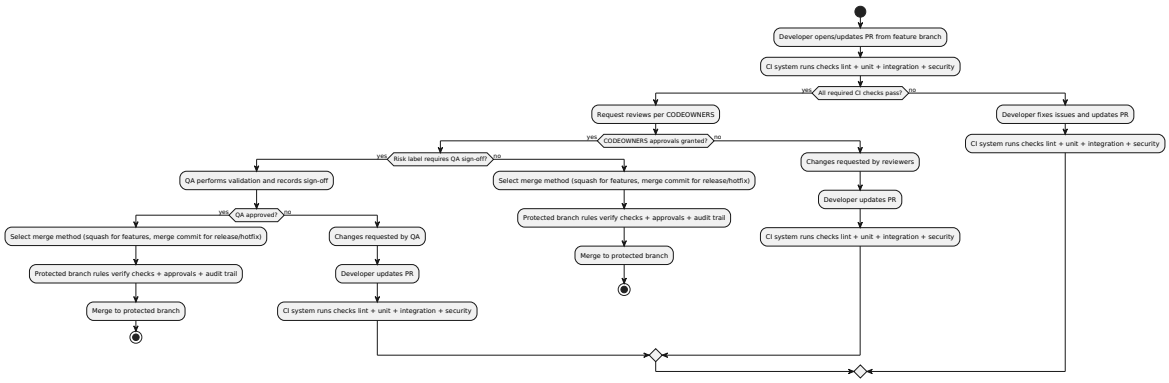
1.11.5.3 Example backport steps (illustrative)

1. Checkout target branch: `git checkout release/1.6`
2. Cherry-pick: `git cherry-pick <sha>`
3. Push branch and open PR with label `backport` and `backport:1.6`
4. Link original PR and include conflict notes (if any)

1.11.6 Decision record (optional variant): QA sign-off routing by risk label

Variant	When you SHOULD use it	Criteria (you MUST record)	Trade-offs
Risk-based QA approval	Repos with mixed-risk change volume	Risk label scheme (e.g., `risk:low	medium
Always-required QA approval	Highly regulated or safety-critical services	Defined QA role, required evidence, and SLA for review	Highest governance; lowest throughput

---



1.12 Versioning and tagging

1.12.1 Objective

You MUST standardize version identifiers and Git tagging so every **Release** can be traced to an exact commit SHA and corresponding immutable **Artifact**, meeting auditability and separation-of-duties constraints.

1.12.2 Scope

This section applies to all in-scope repositories that build, publish, or promote production **Artifacts**. Tooling repositories that influence production outcomes **SHOULD** comply.

1.12.3 Policy

1.12.3.1 Source of truth

- The Git repository history and release metadata MUST be the source of truth for:
  - the released version,
  - the exact commit SHA,
  - the immutable **Artifact** identifiers (e.g., image digest),
  - the **Promotion** history across environments.
- The released version MUST be represented by a Git tag that points to a single commit SHA; tags MUST NOT be moved or recreated.

1.12.3.2 Version scheme

- Services and libraries **MUST** use SemVer: `MAJOR.MINOR.PATCH`.
- `PATCH` **MUST** be incremented for **Patch release** and **Hotfix** deliveries.
- `MAJOR` and `MINOR` increments **SHOULD** follow a documented change classification policy (e.g., breaking vs. additive), and MUST be consistent with public API/contract commitments where applicable.
- Calendar-based versions (CalVer) **MAY** be used for products where SemVer is not meaningful (e.g., packaged business releases), but you MUST implement a deterministic mapping to immutable artifacts and Git tags (see Decision record).

1.12.3.3 Tag format and creation

- Release tags **MUST** be annotated and follow: `vMAJOR.MINOR.PATCH` (example: `v1.4.2`).
- Release tags MUST be created by the CI release job (not by developer workstations) to ensure consistent provenance and audit trails.
- Tags MUST reference the exact commit used to build the released immutable **Artifact**; rebuilds for the same version are prohibited.

1.12.3.4 Signing requirements

- Release tags **MUST** be cryptographically signed where signing infrastructure is available; where mandated (e.g., regulated products), signed tags are **MUST**.
- The CI release job MUST enforce signature verification for any tag used to trigger **Promotion** to production.

- Provenance attestations (e.g., SLSA) and SBOM generation **MUST** be attached to or referenced from the release record for each version.

1.12.3.5 Artifact immutability and promotion linkage

- The same immutable **Artifact** (same digest/checksum) MUST be promoted across environments; you MUST NOT rebuild artifacts when moving from dev/test to staging to production.
- Each release record MUST include:
  - tag `vX.Y.Z`,
  - commit SHA,
  - artifact identifier(s) (e.g., `registry/repo@sha256:...`),
  - SBOM reference,
  - provenance reference,
  - **Promotion** approvals and timestamps per environment.

  Guardrail You MUST treat a version as an immutable pointer to a specific commit SHA and artifact digest. If either changes, you MUST mint a new version.

  Pitfall Rebuilding “the same version” for production (even with identical source) breaks immutability and auditability; you MUST instead promote the already-built artifact.

  Exception If a legacy tool cannot produce annotated or signed tags, you MAY use an interim mechanism (e.g., signed release manifest file) only with a time-bound remediation plan and recorded rationale.

1.12.4 Rationale

Consistent versioning and CI-created, signed tags provide an auditable chain from change request → code change → review → build/test → release → **Promotion** → production deploy. Immutability prevents “it worked in staging but not in prod” drift and supports SOX-like traceability requirements.

1.12.5 Decision record: CalVer variant (optional)

Decision: Use CalVer instead of SemVer.

You MAY choose CalVer only if all criteria are met:

1. Product releases bundle multiple components where API compatibility is not communicated via version number.

2. Release cadence is calendar-driven (e.g., monthly trains) and users interpret versions by date.
3. A deterministic mapping exists from CalVer → Git tag → commit SHA → immutable artifact digest(s).

Requirements when using CalVer:

- Tags MUST still be annotated and signed, and MUST still be CI-created.
- Tag format MUST remain `v<version>`; the `<version>` portion MAY be CalVer (example: `v2026.02.0`), but you MUST document:
  - how patch/hotfix versions are expressed,
  - how multiple artifacts are represented under one version,
  - how backports/hotfixes relate to a prior CalVer tag.

1.12.6 Examples / Checklist

1.12.6.1 Tagging checklist (release job)

1. You MUST verify branch protection and required checks are satisfied for the release commit on `main` or the approved release branch.
2. You MUST compute or retrieve immutable artifact digest(s) produced by CI.
3. You MUST generate SBOM and provenance attestation references.
4. You MUST create an annotated tag `vX.Y.Z` pointing to the release commit.
5. You MUST sign the tag (or enforce the approved exception path).
6. You MUST publish a release record containing tag, commit SHA, artifact digest(s), SBOM, provenance.
7. You MUST promote the same artifact digest(s) across environments; no rebuilds.

1.12.6.2 Commands (illustrative; CI-owned)

- Create annotated tag: `git tag -a v1.4.2 -m "Release v1.4.2"`
- Push tag: `git push origin v1.4.2`

1.12.6.3 Release metadata (minimum fields)

- `version`: `v1.4.2`
- `commit_sha`: `<40-hex SHA>`
- `artifact`: `registry.example.com/app@sha256:<digest>`
- `sbom_ref`: `<uri-or-artifact>`
- `provenance_ref`: `<uri-or-artifact>`
- `approvals`: `<release-approver identities and timestamps>`

1.13 CI/CD pipeline stages and quality gates {#sec-07}

1.13.1 Objective

You **MUST** implement a standardized, auditable CI/CD pipeline that (a) produces an **immutable Artifact** once, (b) promotes that same artifact across environments, and (c) enforces quality and approval gates required for SOX-like controls and traceability.

1.13.2 Scope

This section applies to any repository that builds, publishes, deploys, or promotes a production **Artifact**, and to CI/CD automation that can affect production outcomes. Tooling repositories **SHOULD** comply when they influence production (e.g., shared pipelines, deployment tooling, policy-as-code).

1.13.3 Policy

1.13.3.1 Required pipeline stages (minimum)

Your pipeline **MUST** include the following stages (names may vary, intent may not):

1. `lint` and `unit-test`
2. `build`
3. `integration-test` (or contract/e2e appropriate to the repo type)
4. Security gates:
    - `sast`
    - `dependency-license-scan`
    - `sbom-generate` and store SBOM with the artifact metadata
    - `dast` **MUST** run when the component is user-facing or network-reachable in an environment suitable for testing; otherwise it **SHOULD** be explicitly marked “not applicable” with rationale.
5. `package`
6. `publish` (to an artifact registry/repository) — immutable and content-addressed when available (e.g., digest)
7. `deploy-dev` (promotion using the published artifact)
8. `verify-dev` (smoke tests + basic monitoring checks)
9. `deploy-staging` (promotion using the same artifact)

10. `verify-staging` (smoke tests + monitoring + any required release-candidate checks)
11. `approve-production` (manual gate; see §7.3)
12. `deploy-production` (promotion using the same artifact)
13. `verify-production` (smoke tests + monitoring checks + rollback readiness)

    Guardrail
    You **MUST** "build once, promote many." Rebuilding the same version for different environments is prohibited because it breaks provenance and auditability.

1.13.3.2 Artifact immutability and provenance

- The pipeline **MUST** publish an **immutable Artifact** identified by a version and a unique identifier (e.g., container digest).
- The pipeline **MUST** promote the *same* artifact identifier from `dev` → `staging` → `production` without rebuilding.
- The pipeline **MUST** record traceability metadata for each promotion: change request ID (if applicable), commit SHA, PR link, artifact identifier(s), SBOM/ provenance references, timestamps, and approver identity for manual gates.

  Pitfall
  "Release" and "deploy" are not the same. You **SHOULD** decouple deploy from feature exposure via configuration and **Feature flags** so that `main` remains releasable.

1.13.3.3 Environment progression and approvals

Production promotions are controlled events:

- `dev` deployments **MAY** be automatic after `publish`, subject to required checks.
- `staging` deployments **SHOULD** be automatic after `dev` verification unless capacity/risk requires batching.
- `production` deployments **MUST**:
  - occur only in defined change windows, except via **Emergency change**,
  - require a manual approval step with an auditable rationale,
  - be executed with least privilege and separation of duties (the approver **MUST NOT** be the sole author and sole releaser where tooling supports enforcement).

Exception
Production changes outside the defined window are only allowed via the **Emergency change** process and **MUST** include recorded reason, scope, approver, rollback plan, and follow-up actions.

1.13.3.4 Required checks on protected branches

For protected branches (including `main` and any `release/*` branch), merges **MUST** require passing checks and approvals. At minimum:

- Successful `lint` + `unit-test`
- Successful `build`
- Successful required `integration-test` suite
- Security gates: `sast`, `dependency-license-scan`, `sbom-generate` (and `dast` where applicable)
- Required `CODEOWNERS` approvals and mandatory code review per policy
- Signed release tagging when creating a **Release** (see §7.6)

1.13.3.5 Flaky test policy

- Flaky tests **MUST NOT** be ignored by disabling the gate without a recorded exception and remediation plan.
- Quarantining a flaky test **MAY** be used as a short-term control if:
    1. the quarantine is time-bound,
    2. it is tracked as work (ticket linked),
    3. it does not remove required coverage for high-risk paths.

Guardrail
If a required gate is bypassed, you **MUST** create an auditable record with the approver, rationale, scope, and remediation date.

1.13.3.6 Release tagging and signed releases

- A **Release MUST** be represented by an annotated tag `vMAJOR.MINOR.PATCH` created by CI from a commit that has passed required checks.
- Tags/releases **MUST** be cryptographically signed where supported/mandated.
- Release records **MUST** include: tag, commit SHA, artifact identifier(s), SBOM/provenance references, and promotion approvals/timestamps.

1.13.4 Rationale

A consistent set of CI/CD stages and gates ensures:

- audit-grade traceability from change request → code → review → build/test → **Promotion** → production deploy,
- supply-chain integrity through SBOM/provenance and “build once, promote many,”
- reduced change failure rate by enforcing security and verification gates,
- separation of duties and controlled change windows required for SOX-like environments.

1.13.5 Examples/Checklist

1.13.5.1 Environment progression model (standard)

Environment	Purpose	Promotion method	Approval required
`dev`	Fast feedback, integration	Promote published artifact by version/digest	No (default)
`staging`	Release-candidate validation	Promote same artifact by version/digest	No (default)
`production`	Customer-facing runtime	Promote same artifact by version/digest	Yes (manual + auditable)

1.13.5.2 Required checks (minimum baseline)

Check	Applies to	Required outcome	Evidence to retain
`lint`	PRs to protected branches	Pass	CI run link + logs
`unit-test`	PRs to protected branches	Pass	CI run link + results

Check	Applies to	Required outcome	Evidence to retain
`integration-test`	PRs to protected branches	Pass	CI run link + results
`sast`	PRs to protected branches	Pass (or approved exception)	Report reference
`dependency-license-scan`	PRs to protected branches	Pass (or approved exception)	Report reference
`sbom-generate`	On publish/ release	Generated + stored	SBOM artifact/ link
`dast`	As applicable	Pass (or justified N/A)	Report reference / N/A record
`smoke-tests`	Post-deploy	Pass	Run link + results
`monitoring-checks`	Post-deploy	Pass thresholds	Dashboard/ snapshot link

1.13.5.3 Promotion checklist (staging → production)

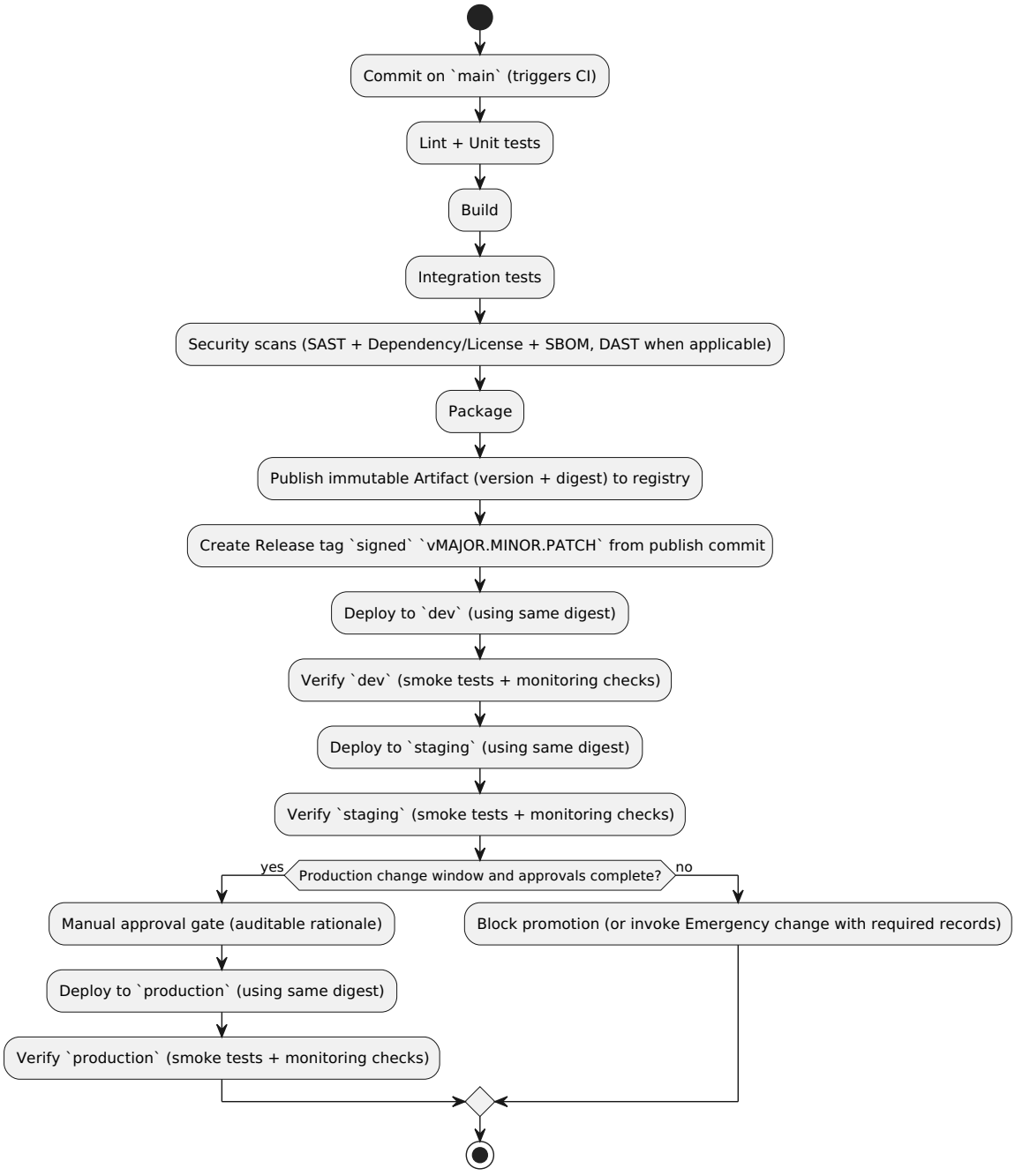
1. You **MUST** confirm the artifact identifier (e.g., digest) promoted to staging matches the candidate for production.
2. You **MUST** confirm change window compliance, or invoke **Emergency change** with required records.
3. Release Manager approves the `approve-production` gate with rationale and references (PR, incident, change request).
4. You **MUST** run `verify-production` smoke tests and monitoring checks.
5. You **MUST** confirm rollback plan viability (previous artifact available and promotion path tested).

1.13.6 Decision record (optional variants)

Repositories **MAY** use either on-demand releases or release trains, but you **MUST** document the choice and criteria in `docs/decision-records/branching-and-release.md`.

Variant	When to choose	Criteria you MUST document
On-demand releases	High-velocity services, strong automation	risk controls, staffing for approvals, rollback readiness
Release trains	Coordinated multi-team delivery, regulatory batching	cutoff rules, stabilization period, go/no-go process, exception handling

1.13.7 Diagram: Artifact build once, promote through environments



1.14 sec-08 — Release cadence and release trains

1.14.1 Objective

You standardize how releases reach production with predictable timing, explicit risk control, and SOX-like auditability, while keeping `main` releasable and enabling continuous delivery by default.

1.14.2 Scope

This section applies to teams that bundle changes into scheduled releases (release trains) in addition to the continuous delivery model defined elsewhere.

1.14.3 Policy

- Continuous delivery is the default:
  - You **MUST** keep `main` releasable at all times (no “integration debt”); freezes **MUST** be approved and recorded with scope, dates, and exit criteria.
  - You **MUST** build an immutable **Artifact** once per release candidate and **MUST** promote the same artifact version/digest across environments without rebuilding.
- Release trains are an opt-in variant:
  - Teams **MAY** use release trains when coordination, regulatory needs, or operational risk requires bundling and synchronized go/no-go decisions.
  - Teams using release trains **MUST** define and publish (in the repo or release runbook): cadence, cutoff, stabilization duration, go/no-go attendees, and production change window alignment.
- Cutoff and stabilization controls:
  - A release train **MUST** have a documented cutoff (time and/or commit reference).
  - After cutoff, a `release/<major>.<minor>` branch **MUST** be cut from `main`.
  - During stabilization, merges to the release branch **MUST** be limited to critical fixes only, sourced from `main` via **Cherry-pick** (backport discipline).
  - Stabilization changes **MUST** remain auditable: linked ticket, risk classification, test evidence, and approvals per branch protection/ CODEOWNERS.

- Go/no-go and production governance:
  - Promotion to production from a release train **MUST** occur only during defined production change windows unless using the **Emergency change** process.
  - The production release **MUST** be a signed, versioned **Release** (signed tag `vMAJOR.MINOR.PATCH`) and traceable to artifact provenance (commit SHA + artifact digest).
  - Go/no-go decisions **MUST** be recorded (decision, attendees, risks accepted, rollback plan, and approver).
- Roles and communication:
  - A Release Captain (rotating) **MUST** be assigned for each train; Release Manager approves go/no-go and production promotion.
  - Teams **SHOULD** use a single, named communication channel per train (e.g., `#release-<product>`), and you **MUST** log key decisions/links in the release record.

> [!Guardrail] You **MUST** not “stabilize on `main`” by halting normal integration without an approved and recorded freeze. Stabilization happens on `release/<major>.<minor>` after cutoff, with `main` continuing in parallel (preferably behind feature flags).

> [!Exception] You **MAY** skip a release branch for a train when changes are trivially low-risk and all required gates can be satisfied on `main` within the production window; you **MUST** record the rationale in the release record and still cut a signed tag tied to the promoted artifact.

> [!Pitfall] Rebuilding artifacts in staging to “pick up fixes” breaks immutability and auditability. You **MUST** rebuild only by creating a new versioned artifact from a new commit, then promote that artifact forward.

1.14.4 Rationale

Release trains provide coordination points (cutoff, stabilization, go/no-go) that reduce operational risk for systems with complex dependencies. By cutting a release branch at cutoff and enforcing backport-only stabilization, you keep `main` moving (often behind feature flags) while preserving a controlled, auditable path to production.

1.14.5 Decision record (optional variants)

Variant	You choose it when	Criteria you MUST document	Trade-offs
Continuous delivery (default)	You can ship frequently with low coordination cost	On-call readiness, automated gates, production window fit	Fastest lead time; requires strong automation and feature flag discipline
Scheduled release train	You need synchronization across teams/components or stricter change control	Cadence, cutoff definition, stabilization duration, go/no-go RACI, comms channel	Slower lead time; clearer coordination and audit checkpoints
Multiple trains (e.g., weekly + monthly)	You have mixed risk profiles across subsystems	Which repos/services map to which train; dependency handling; exception path	More complexity; can optimize throughput vs risk

1.14.6 Workflow (release train scenario)

1. Before the train starts, you **MUST** publish the train schedule (cutoff time, stabilization window, go/no-go time, and target production window).
2. Development proceeds on `main` using short-lived `feature/<ticket>-<slug>` branches and feature flags where appropriate.
3. At cutoff, the Release Captain **MUST**:
    1. Identify the cutoff commit on `main`.
    2. Create `release/<major>.<minor>` from the cutoff commit.
    3. Start stabilization with stricter merge criteria (critical fixes only).
4. During stabilization:
    1. Fixes are made on `main` first.
    2. The fix is **Cherry-pick**'d to `release/<major>.<minor>` (recording the source commit).

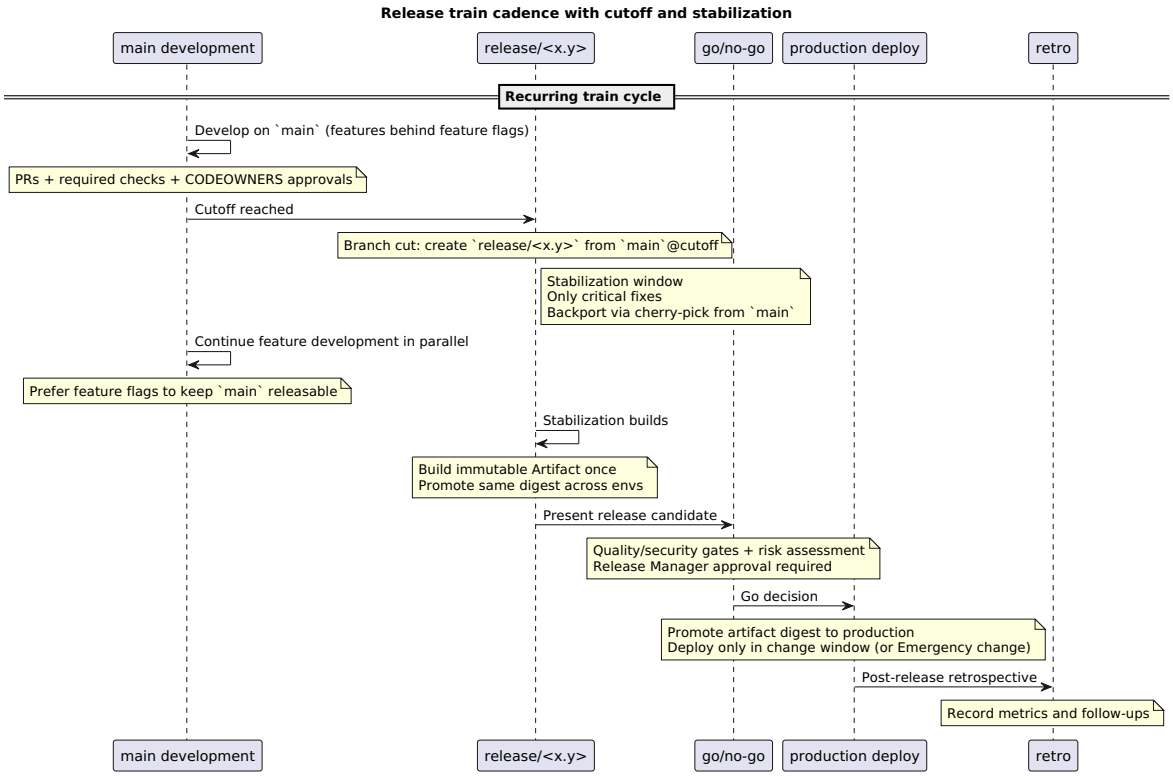
    3. CI/CD gates run; artifacts produced for the release candidate are immutable and promoted forward.
5. At go/no-go, the Release Manager approves promotion based on the checklist below.
6. Production deploy occurs only in the defined change window (or emergency process), using promotion of the same artifact digest.
7. After production, the Release Captain runs a retrospective and records follow-ups.

1.14.7 Go/no-go checklist (minimum)

You **MUST** record outcomes and links in the release record.

- Artifact/provenance:
    - Immutable artifact digest identified and matches the release tag commit SHA
    - Signed tag `vMAJOR.MINOR.PATCH` ready/created (per your release process)
- Quality and security:
    - All required CI checks green (tests, integration as applicable)
    - SAST, dependency/license scan, SBOM generation complete; DAST complete when applicable
- Change governance:
    - Production window confirmed; communications posted
    - Rollback plan validated (redeploy prior artifact + config/flag rollback steps)
    - Open risks enumerated; explicit approval to accept residual risk
- Operational readiness:
    - Monitoring/alerting updated if needed; runbooks updated
    - Support/on-call coverage confirmed

1.14.8 Diagram — Release train cadence with cutoff and stabilization



1.15 sec-09 — Release branch workflow (stabilization)

1.15.1 Objective

You **MUST** provide a repeatable, auditable stabilization workflow that cuts `release/<major>.<minor>` from a known-green `main`, restricts post-cutoff changes to approved fixes, and results in a signed **Release** (`vMAJOR.MINOR.PATCH`) tied to a promoted **immutable artifact**.

1.15.2 Scope

This section applies when you choose the optional stabilization pattern using `release/<major>.<minor>` branches for a specific upcoming release while `main` continues to accept feature work (ideally behind **feature flags**).

1.15.3 Policy

1.15.3.1 1) Branch cut prerequisites

- You **MUST** cut `release/<major>.<minor>` from `main` at a commit that has:
  - All required CI checks passing (see sec-07/sec-08 gates).
  - Required reviews satisfied (including `CODEOWNERS`), and branch protection enforced.
  - A build provenance record that can be traced to a publishable **Artifact**.
- You **MUST** record the cutoff commit SHA, timestamp, and Release Captain in the change record/release tracking ticket.

  Guardrail
  `main` **MUST** remain releasable during stabilization unless an approved, recorded freeze exists (approval, scope, dates, exit criteria).

1.15.3.2 2) What changes are allowed after branch cut

- On `release/<major>.<minor>`, you **MUST** allow only:
  - Release-blocking bug fixes, security fixes, and compliance-mandated changes.
  - Changes required to pass gates (test fixes) only when they do not mask product defects.
- You **MUST NOT** introduce new features on `release/<major>.<minor>`. Features continue on `main` and **SHOULD** be protected by **feature flags** and safe rollout controls.

  Pitfall
  “Small” feature scope creep during stabilization destroys auditability and predictability. Treat every non-blocker change as a deferrable candidate to the next release.

1.15.3.3 3) Fix origination and cherry-pick/backport rules

- Fixes intended for the release branch **MUST** originate on `main` first, then be **Cherry-pick**ed to `release/<major>.<minor>`, unless the change is release-branch-only (e.g., version metadata) and explicitly justified in the tracking record.
- If the fix already exists on `main`, you **MUST** cherry-pick the existing commit(s) into `release/<major>.<minor>` and link:
  - Source PR on `main`

    - Cherry-pick PR on `release/<major>.<minor>`
    - The release-blocker ticket
- If the fix does not exist on `main`, you **MUST**:
    1. Implement the fix on `main` with tests (or documented test exception).
    2. Merge via PR meeting all required checks/approvals.
    3. Cherry-pick the resulting commit(s) to `release/<major>.<minor>` via PR.
- You **MUST NOT** rebase shared branches (including `release/<major>.<minor>`). Conflict resolution **MUST** be captured in the cherry-pick PR.

  Exception
  A release-branch-only fix **MAY** be permitted for release metadata or build pipeline configuration that cannot safely land on `main`. You **MUST** document why it cannot land on `main`, who approved it, and how it will be reconciled post-release.

1.15.3.4 4) Stabilization validation and exit criteria

- After branch cut, you **MUST** run full regression appropriate to the repo type and risk profile (e.g., unit/integration/e2e, SAST, dependency/license, SBOM; DAST when applicable).
- After each accepted fix to `release/<major>.<minor>`, you **MUST** re-run the required gate set (at minimum: build + targeted regression + security delta checks).
- Exit criteria for release readiness **MUST** be explicit and recorded, including:
    - Zero open release blockers (or explicit go/no-go exception approvals)
    - All required gates passing on the release candidate commit
    - Release notes draft complete and reviewed

1.15.3.5 5) Tagging, signing, and promotion (immutability)

- You **MUST** create a signed annotated tag `vMAJOR.MINOR.PATCH` on the release branch commit approved at go/no-go.
- Tagging **MUST** map to a single promotable **Artifact** (by digest/version) that is promoted across environments without rebuilding.
- You **MUST** promote the same artifact version from staging to production (no rebuild), using the audited approval gate and within defined production windows (unless Emergency change).

1.15.3.6 6) End-of-life (EOL) for the release branch

- After production promotion and verification, you **MUST** close `release/<major>.<minor>` by:
    1. Ensuring all stabilization fixes are present on `main` (by construction via “fix on main first”, or via forward-port PR if an approved exception was used).
    2. Finalizing and publishing release notes linked to the signed tag.
    3. Deleting `release/<major>.<minor>` per repository retention policy, preserving audit records (PRs, checks, approvals, tag signatures, artifact provenance).

1.15.4 Rationale

A stabilization branch exists to reduce release risk while preserving throughput on `main`. Requiring fixes to land on `main` first ensures a single source of truth, prevents “lost fixes,” and supports SOX-like traceability: change request → PR/review → checks → signed **Release** → **Promotion** of an **immutable artifact** → production deploy within controlled windows.

1.15.5 Examples/Checklist

1.15.5.1 Release branch checklist (operator-focused)

1. Cut branch:
    - Create `release/<major>.<minor>` from green `main` commit `abc1234`.
    - Record cutoff in tracking ticket (SHA, timestamp, Release Captain).
2. Lock scope:
    - Confirm “no new features” rule and define release blockers.
3. Validate baseline:
    - Run full regression and required security gates on `release/<major>.<minor>`.
4. Manage fixes:
    - For each blocker: fix on `main` → merge with checks → cherry-pick to `release/<major>.<minor>` via PR → rerun gates.
5. Go/no-go:
    - Confirm exit criteria met; record approvals and production window.

6. Release:
    - Create signed tag `vX.Y.Z`; ensure tag references artifact digest/provenance.
7. Promote:
    - Promote the same artifact across environments without rebuilding.
8. Close:
    - Ensure `main` contains all fixes; publish release notes; delete `release/<major>.<minor>`.

1.15.5.2 Required tracking fields (minimum)

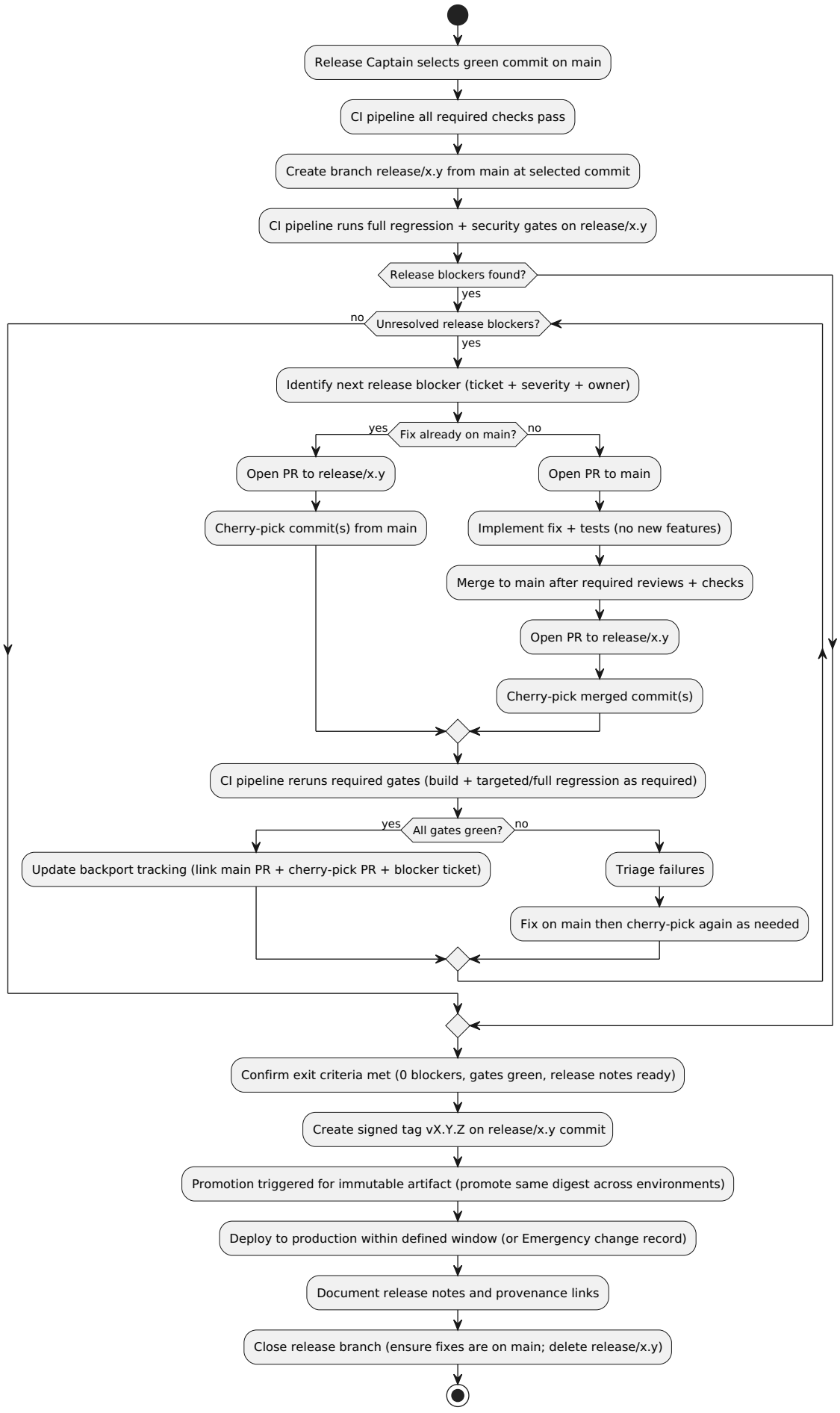
- Release train/target version (`X.Y.Z`)
- Cutoff commit SHA on `main`
- List of release blockers and linked tickets
- Cherry-pick PR links (source on `main`, target on `release/<major>.<minor>`)
- Go/no-go approvers and timestamp
- Signed tag reference and artifact digest
- Production window ID (or Emergency change record)

1.15.6 Decision record (optional variant): Release trains vs on-demand stabilization

Variant	You choose it when	Criteria you MUST meet	Trade-offs
Scheduled release train (e.g., weekly)	Multiple teams ship frequently and need predictable cutoffs	Cutoff time, stabilization window, and go/no-go cadence are documented; staffing for Release Captain/CI support is scheduled	Predictability; may delay urgent non-emergency changes until next train
On-demand stabilization	Releases are infrequent or driven by discrete milestones	Cutoff and exit criteria are defined per release; approvals and	Flexible; higher coordination overhead per release

Variant	You choose it when	Criteria you MUST meet	Trade-offs
		production windows are prebooked	

1.15.7 Release branch stabilization decision flow (diagram)


Release Captain selects green commit on main
CI pipeline all required checks pass
Create branch release/x.y from main at selected commit
CI pipeline runs full regression + security gates on release/x.y
Release blockers found?
yes
no
Unresolved release blockers?
yes
Identify next release blocker (ticket + severity + owner)
yes
Fix already on main?
no
Open PR to release/x.y
Cherry-pick commit(s) from main
Open PR to main
Implement fix + tests (no new features)
Merge to main after required reviews + checks
Open PR to release/x.y
Cherry-pick merged commit(s)
CI pipeline reruns required gates (build + targeted/full regression as required)
yes
All gates green?
no
Update backport tracking (link main PR + cherry-pick PR + blocker ticket)
Triage failures
Fix on main then cherry-pick again as needed
Confirm exit criteria met (0 blockers, gates green, release notes ready)
Create signed tag vX.Y.Z on release/x.y commit
Promotion triggered for immutable artifact (promote same digest across environments)
Deploy to production within defined window (or Emergency change record)
Document release notes and provenance links
Close release branch (ensure fixes are on main; delete release/x.y)

1.16 Hotfix and emergency change process {#sec-10}

1.16.1 Objective

You **MUST** minimize MTTR while preserving SOX-like auditability, traceability, and separation of duties for any production-impacting change executed as a **Hotfix** or **Emergency change**.

1.16.2 Scope

This section applies to any in-scope repository that:

- Builds/publishes a production **Artifact**, or
- Participates in CI/CD that deploys/promotes to production.

Tooling repositories **SHOULD** follow this process when they can affect production outcomes.

1.16.3 Policy

1.16.3.1 Classification and triggers

1. You **MUST** classify the event before making changes:
   - **Hotfix**: a targeted patch release to remediate a production issue, shipped as a signed patch **Release** (SemVer PATCH).
   - **Emergency change**: an out-of-window production change executed under expedited controls to restore service or address critical risk.
2. You **MUST** create or reference a change record (incident/ticket) that links: request → code → review → build/test → **Promotion** → production deploy.
3. Production changes **MUST** occur only during defined windows unless executed as an **Emergency change**.

   Guardrail
   `main` **MUST** remain releasable at all times. Any freeze **MUST** be approved and recorded with scope, dates, and exit criteria.

1.16.3.2 Branching and source of truth for hotfixes

1. You **MUST** branch hotfixes from the tagged production **Release** commit (preferred) using `hotfix/<major>.<minor>.<patch>-<ticket>`.
    - Example: `hotfix/1.4.3-INC12345`
2. You **MUST NOT** rebuild an existing version. The hotfix **MUST** produce a new patch version and a new immutable **Artifact** with new digest/version.
3. The resulting release tag **MUST** be annotated `vMAJOR.MINOR.PATCH` and **MUST** be created by CI and cryptographically signed where supported/mandated.

    Pitfall
    Cutting a hotfix from `main` instead of the production tag breaks traceability and can accidentally include unrelated changes.

1.16.3.3 Expedited approvals and required checks

1. You **MUST** use a PR to merge the hotfix into the target branch (the hotfix branch back into the production line or release branch as defined by your repo model). Direct pushes to protected branches **MUST** be blocked.
2. Minimum required controls for a hotfix:
    - Code review **MUST** occur (expedited allowed), enforcing `CODEOWNERS`.
    - CI build/test **MUST** pass.
    - SAST, dependency/license scan, and SBOM generation **MUST** run (allow policy-defined emergency waivers only with recorded approval and compensating controls).
    - A production promotion approval **MUST** be recorded and **MUST** preserve separation of duties (Release Manager approves).
3. Security review:
    - A Security Reviewer **MUST** approve when the change touches security-sensitive components or severity meets the security escalation threshold.
    - Security review **MAY** be skipped only via a documented exception with explicit risk acceptance and follow-up actions.

    Exception
    If a required check cannot run due to platform outage, you **MAY** proceed only under **Emergency change** controls with: recorded approver, scope, rollback plan, and an обязатель post-facto validation run attached to the incident.

1.16.3.4 Deployment, release, and promotion rules

1. You **MUST** build the immutable **Artifact** once and **MUST** promote the same version/digest across environments without rebuilding.
2. You **MUST** deploy the hotfix via the standard CI/CD pipeline with an auditable promotion gate.
3. You **MUST** create a signed release tag (`vMAJOR.MINOR.PATCH`) that maps to:
    - Commit SHA
    - Artifact identifier(s) (e.g., image digest)
    - SBOM/provenance references
    - Approval identity and timestamps for promotions

1.16.3.5 Forward-porting (mandatory anti-regression control)

1. After the hotfix is released to production, you **MUST** forward-port the fix to:
    - `main`, and
    - any active `release/<major>.<minor>` stabilization branches
2. Forward-porting **MUST** be done by `cherry-pick` (preferred for traceability) or forward-merge, as appropriate to your branching posture.
3. The forward-port PRs **MUST** reference the incident/ticket and the production release tag.

    Pitfall
    Failing to forward-port guarantees regression when `main` (or a release train) later ships without the fix.

1.16.3.6 Post-release actions (required for auditability and prevention)

Within the defined SLA after restoration:

1. You **MUST** update the incident record with:
    - Root cause summary
    - Timeline and decision log (including go/no-go)
    - Links to PR(s), tag(s), artifact digest(es), and promotion approvals
2. You **MUST** run/extend regression coverage to prevent recurrence (tests, monitors, alerts).
3. You **MUST** create follow-up work items for preventative controls (e.g., hardening, additional gates, feature flags, expand/contract migrations).

1.16.4 Rationale

Hotfixes and emergency changes optimize MTTR while preserving compliance: strict traceability, immutable artifact promotion, enforced review/approval, and forward-porting ensure the production fix is both auditable and not lost in subsequent releases.

1.16.5 Decision record: Optional variants (release trains vs. continuous hotfixing)

Variant	When you SHOULD use it	Criteria you MUST document if chosen
Continuous hotfixing (ship immediately)	High-severity incidents, strict MTTR goals	Severity, customer impact, rollback plan, expedited approvers
Hotfix aligned to release train (ship at next window)	Low/moderate impact, stable workarounds exist	Risk of delay, interim mitigations, train cutoff alignment, stakeholder sign-off

---

1.16.6 Examples/Checklist

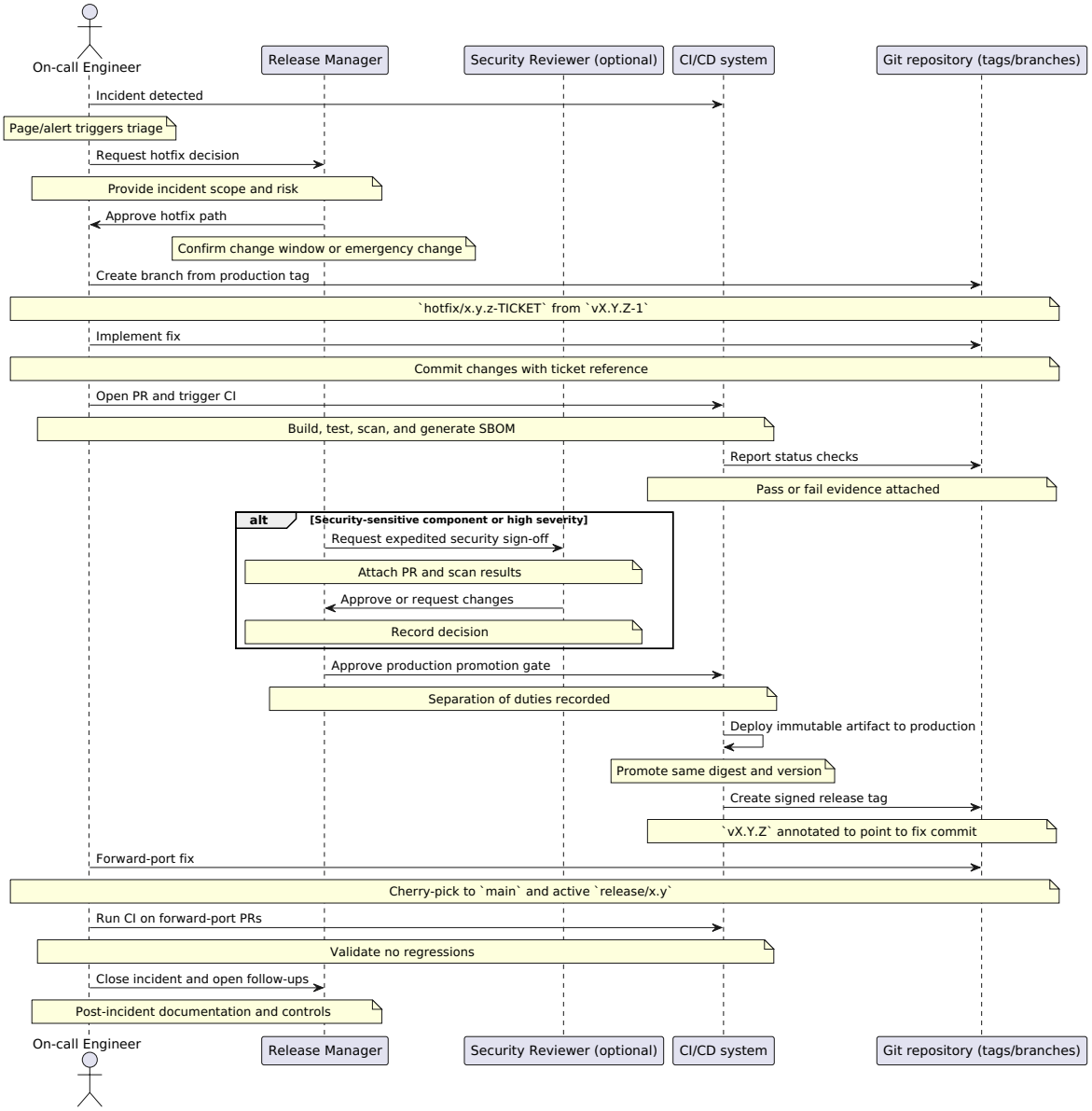
1.16.6.1 Hotfix procedure (branch → release → forward-port)

1. Identify the production release tag (e.g., `v1.4.2`) and confirm the deployed artifact digest.
2. Create hotfix branch from the production tag: `git checkout -b hotfix/1.4.3-INC12345 v1.4.2`
3. Implement the minimal fix; include the incident/ticket reference in commits/PR metadata.
4. Open PR; obtain expedited `CODEOWNERS` approval; run required checks.
5. CI creates signed tag `v1.4.3` and publishes the new immutable **Artifact**; record artifact digest and SBOM/provenance.
6. Promote the same artifact to production via the pipeline; Release Manager approves the production gate.
7. Forward-port:
    - `cherry-pick` into `main` via PR
    - `cherry-pick` into each active `release/<major>.<minor>` via PR

8. Close incident; file follow-ups and attach evidence links (PRs, tags, approvals, digest).

1.16.6.2 Required evidence checklist (attach to incident/change record)

- [ ] Incident/ticket ID and severity classification (Hotfix vs Emergency change)
- [ ] PR links (hotfix + forward-port PRs) and review approvals (`CODEOWNERS`)
- [ ] CI run IDs and gate status (tests, SAST, dependency/license, SBOM; DAST when applicable)
- [ ] Release tag v`MAJOR.MINOR.PATCH` and commit SHA
- [ ] Artifact identifier(s) + digest and registry/repo location
- [ ] Promotion approvals and timestamps (including production gate approver)
- [ ] Rollback plan and outcome (executed or not)
- [ ] Post-incident summary and preventative follow-ups



1.17 sec-11 — Feature flags and progressive delivery

1.17.1 Objective

You **MUST** enable trunk-first development while preserving production stability and SOX-like auditability by controlling exposure separately from deploy/release, and by standardizing rollout and rollback patterns.

1.17.2 Scope

This section applies to any in-scope repository that can affect production outcomes (services, infrastructure, and deployable applications) and any supporting tooling that configures feature flags or progressive delivery.

1.17.3 Policy

1.17.3.1 Feature flag governance

- You **MUST** implement features behind a **Feature flag** when the change is not safe to expose immediately upon deploy, or when rollout risk cannot be reduced to acceptable levels via testing alone.
- You **MUST** define an owner for each flag (team and on-call rotation) and record it in a discoverable location (e.g., `flags/registry.yaml` or `docs/flags.md`).
- You **MUST** ensure feature flag changes are auditable (who/what/when/why) and retained per your compliance retention period.
- You **MUST** treat flag configuration as production change when it affects production behavior (subject to production windows, approval, and separation of duties).
- You **SHOULD** use default-off flags at merge time (“merge darkly”) unless the feature is proven backward compatible and low-risk.
- You **MUST** define a lifecycle for each flag: create → default-off → progressive rollout → full rollout → cleanup (remove flag and dead code).
- You **MUST** remove stale flags and associated dead code within a defined timebox (e.g., by the next `MINOR` release or within 30–90 days); deviations **MUST** be recorded with rationale and a removal date.

  Guardrail
  `main` **MUST** remain releasable; feature flags are the primary mechanism to merge incomplete work without destabilizing releases.

1.17.3.2 Progressive delivery standards

- You **MUST** promote an **Immutable artifact** across environments without rebuilding; progressive delivery adjusts exposure, not the artifact.
- You **MUST** include explicit verification steps (automated and/or manual) between rollout stages.
- You **MUST** define objective health criteria and alerting that can trigger rollback (error budget burn, SLO violations, elevated 5xx, latency regression, business KPI drop).

- You **SHOULD** prefer progressive delivery methods in this order when applicable:
    1. Feature flag ramp (fastest rollback, least disruptive)
    2. Canary release (small % of traffic/users)
    3. Ring-based rollout (internal → beta → general)
    4. Blue/green (when infrastructure supports clean cutover)
- You **MUST** document the chosen strategy per service in `docs/progressive-delivery.md` (or equivalent), including rollback triggers and responsibilities.

1.17.3.3 Rollback requirements

- You **MUST** prefer rollback via flag configuration (turn flag off) when safe and sufficient.
- You **MUST** support artifact rollback (redeploy prior known-good artifact digest) when flag rollback is insufficient (schema incompatibility, non-flagged behavior change, security incident).
- You **MUST** ensure database changes follow **expand/contract** so that turning a flag off or redeploying a prior artifact remains viable during the rollout window.

    Pitfall
    “Release” and “deploy” are not the same. You **SHOULD** deploy frequently and control exposure with configuration and **Feature flags**.

    Exception
    A feature may ship without a flag only if the change is demonstrably low-risk and fully reversible via artifact rollback without data loss; the exception **MUST** be approved and recorded (ticket + approver + rationale).

1.17.3.4 Required checks (minimum)

Control area	Requirement
Ownership	Flag owner and escalation contact **MUST** be recorded
Auditability	Flag change logs **MUST** capture actor, timestamp, environment, before/after, and ticket/reference

Control area	Requirement
Separation of duties	Production flag changes **MUST** require approval by a role distinct from the change author (where tooling permits)
Immutability	Artifact digest/version **MUST** remain constant across rollout stages
Verification	Each ramp step **MUST** include verification evidence (dashboards, test results, or signed checklist)
Rollback	Runbook **MUST** define flag-off and artifact rollback procedures

1.17.4 Rationale

Feature flags and progressive delivery reduce the blast radius of change, preserve the requirement that `main` remains releasable, and improve auditability by making exposure changes explicit, reviewable, and reversible. A “config-first rollback” minimizes MTTR and avoids unnecessary rebuilds, aligning with immutable artifact promotion and production window controls.

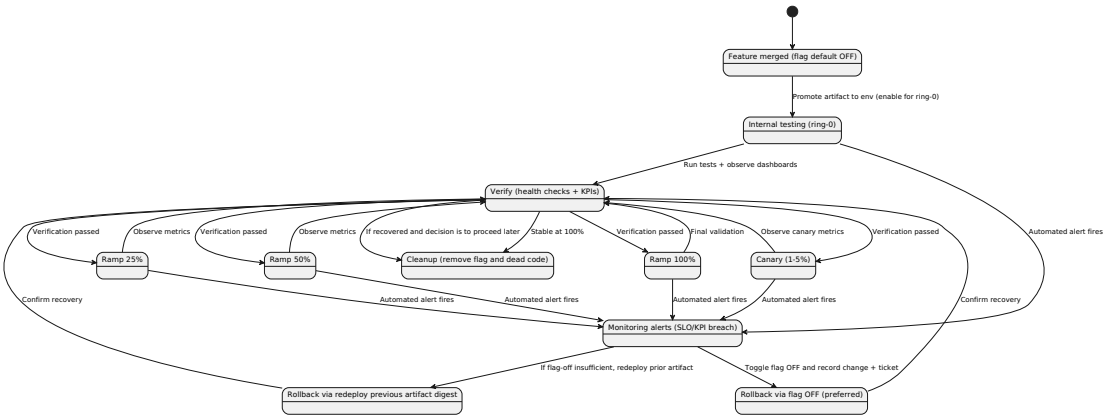
1.17.5 Examples/Checklist

1.17.5.1 Flag lifecycle checklist

1. Create flag with a stable key (e.g., `feature.checkout_v2_enabled`) and default off.
2. Record metadata (owner, purpose, expected removal date, linked ticket) in `flags/registry.yaml`.
3. Merge to `main` with flag off; artifact is built once and promoted.
4. Internal testing: enable flag only for internal users/ring and verify metrics.
5. Canary: ramp to small % (e.g., 1–5%) and verify.
6. Ramp: increase to `25%` → `50%` → `100%` with explicit verification between stages.
7. Cleanup: remove flag and dead code; delete configuration entries; update registry.

1.17.5.2 Rollout/rollback operational checklist (production)

1. Confirm release artifact digest to be promoted is immutable and already verified in staging.
2. Confirm production window and approvals are in place (or execute **Emergency change** with required records).
3. Enable flag for internal/ring-0; verify.
4. Ramp canary and each subsequent stage only after verification.
5. If alerts fire, roll back by turning flag off; if insufficient, redeploy prior artifact digest and follow the incident process.



1.17.5.3 Decision record (optional variant): rollout cadence

You **MAY** use either on-demand rollout or a scheduled rollout train for feature exposure changes. You **MUST** document the choice and criteria in `docs/decision-records/branching-and-release.md`.

Option	When it fits	Criteria you MUST record
On-demand progressive rollout	High autonomy, fast iteration, small/ independent services	Risk classification, monitoring readiness, approver model, rollback SLO/MTTR target
Scheduled rollout train	Coordinated releases, shared dependencies, constrained production windows	Cutoff rules, go/no-go attendees, batching rationale, exception handling and audit artifacts

1.18 sec-12 Dependencies, monorepos, and multi-repo coordination

1.18.1 Objective

You **MUST** reduce cross-repo release friction and breakages while preserving SOX-like auditability and repeatable promotion of **immutable artifacts**. You **MUST** define compatibility and upgrade policies so that dependent repos can coordinate changes without rebuilding artifacts during **Promotion**.

1.18.2 Scope

This section applies to:

- Any repo that publishes an **Artifact** consumed by another repo (libraries, base images, shared pipelines).
- Any repo whose production deploy depends on artifacts from other repos (services, apps).
- Monorepos containing multiple release units (per taxonomy requirements).

1.18.3 Policy

1.18.3.1 1) Dependency update cadence and reproducibility

- You **MUST** use versioned, immutable dependency references for all production-impacting dependencies (e.g., container digests, package versions, Git tags).
- You **MUST** commit and enforce lockfiles (when supported) to ensure reproducible builds.
- You **MUST** prevent “floating” dependency upgrades in CI/CD for release/promotion workflows (e.g., `latest` tags, unpinned branches).
- You **SHOULD** define a standard dependency update cadence per repo type:
  - Services: at least monthly dependency refresh, plus urgent security updates as needed.
  - Libraries: continuous releases acceptable, but consumers must have a defined upgrade path (see compatibility policy).
- You **SHOULD** run dependency/security scanning on both direct and transitive dependencies and store SBOM outputs as release artifacts.

Guardrail
You **MUST** promote the same immutable artifact digest/version across environments without rebuilding. Dependency updates that change build output **MUST** be represented as a new version and a new release record.

Pitfall
Unpinned base images, “auto-update” build steps, or CI fetching dependencies without lockfile enforcement will break traceability and invalidate promotion guarantees.

1.18.3.2 2) API compatibility and deprecation windows

- Producers (libraries and services exposing APIs) **MUST** publish a compatibility contract:
  - Supported major versions.
  - Backward compatibility guarantees within a major version (SemVer).
  - Deprecation policy with timelines.
- Breaking changes **MUST** increment SemVer MAJOR and **MUST** include:
  - Migration guidance.
  - A deprecation window for consumers unless an **Emergency change** is approved and recorded.
- Consumers **MUST** not adopt breaking changes without an explicit coordination plan (see “Coordinated releases” below).
- You **SHOULD** use **Feature flags** to decouple deploy from exposure when coordinating multi-service behavior changes.

Exception
Emergency change may shorten deprecation windows, but you **MUST** record reason, approver, scope, rollback plan, and follow-ups, and you **MUST** complete post-facto review.

1.18.3.3 3) Coordinated releases (multi-repo): manifests, pinned versions, integration testing

- For any release involving multiple repos, you **MUST** use a version-pinned “release manifest” stored in a controlled location (repo or release system) that records:
  - Each repo name.
  - Release tag `vMAJOR.MINOR.PATCH`.
  - Commit SHA.
  - Artifact identifier(s) (e.g., image digest).

    - SBOM/provenance reference(s).
- The release manifest **MUST** be the single source of truth for the version set promoted across environments.
- You **MUST** execute an integration test suite against the exact manifest version set before production **Promotion**.
- Promotions **MUST** reference the manifest version set and approvals/timestamps for each environment gate.

Guardrail
If the manifest changes, you **MUST** treat it as a new release candidate: rerun required checks and record a new go/no-go decision.

1.18.3.4 4) Handling breaking changes across multiple services

- If a shared library introduces a breaking change, the library release **MUST** occur before dependent service releases.
- Dependent services **MUST** upgrade, pass CI/security gates, and pass integration tests on the pinned manifest set prior to production **Promotion**.
- You **MUST** define rollback strategy at the manifest level:
    - Rolling back **MUST** mean promoting a prior known-good manifest set (or prior artifact versions), not rebuilding.
- You **SHOULD** use expand/contract for shared database/schema dependencies and coordinate the contract phase only after all consumers are confirmed migrated.

1.18.4 Rationale

Cross-repo dependencies amplify risk: an uncoordinated dependency change can break multiple services, compromise auditability, or force rebuilds that violate immutable artifact promotion. A pinned release manifest plus integration testing provides traceability from change request → code → review → build → test → **Promotion** → production, while preserving separation of duties and controlled windows.

1.18.5 Examples/Checklist

1.18.5.1 Branch and versioning checklist (multi-repo change)

1. You create changes on `feature/<ticket>-<slug>` in each affected repo.

2. Producers (e.g., shared library) cut a release tag `vMAJOR.MINOR.PATCH` via CI, producing an immutable **Artifact** and SBOM/provenance.
3. Consumers update dependency references to the new producer version (lockfile updated as needed).
4. You generate or update the release manifest pinning all versions.
5. CI runs unit + integration + security gates using the manifest set.
6. Release Manager approves go/no-go and production promotion within the change window (or records **Emergency change**).

1.18.5.2 Required checks for coordinated releases (minimum)

Check category	Minimum requirement
Traceability	Ticket/CR linked in PR and release record; manifest includes tag + SHA + artifact identifiers
Code review	**MUST** satisfy `CODEOWNERS` approvals per repo
Reproducibility	Lockfiles enforced; dependencies pinned; no floating versions
Security	SAST + dependency/license scan + SBOM generation (**DAST** when applicable)
Integration	Manifest-based integration test suite passes
Promotion governance	Manual, auditable approval gate with separation of duties

1.18.5.3 Decision record: Release trains vs on-demand coordination

Variant A — On-demand coordinated releases

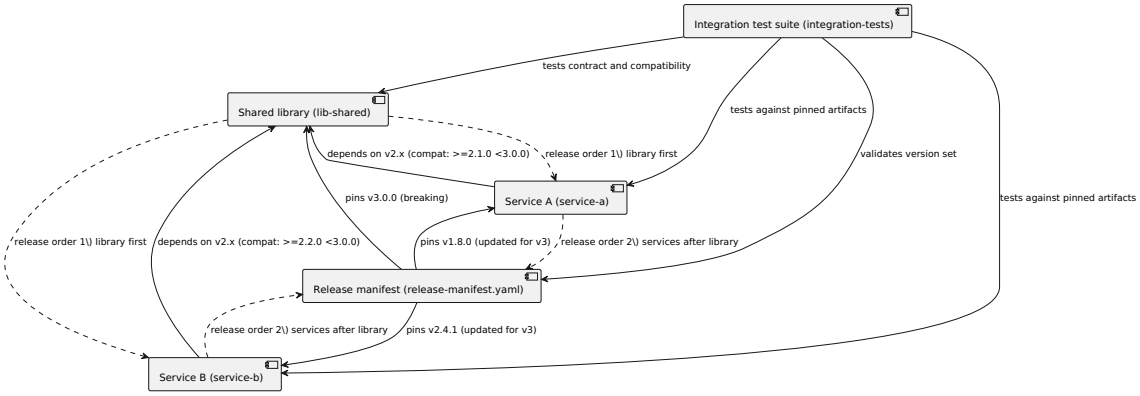
- Criteria: you have low coupling, fast CI, and can coordinate changes per feature/incident.
- You **SHOULD** use this when cross-repo breaking changes are infrequent.

Variant B — Release train (scheduled coordination)

- Criteria: you have high coupling, frequent dependency churn, or repeated “version skew” incidents.
- You **SHOULD** adopt a train when you need predictable cutoffs, stabilization, and shared go/no-go.
- You **MUST** define and publish:
  - Train cadence and cutoff.
  - Stabilization rules.
  - Ownership for manifest publication and approvals.

> **Pitfall**
> Release trains without strict manifest pinning and integration testing will create “it worked on my branch set” failures and undermine auditability.

1.18.6 Diagram: Coordinated multi-repo release dependency graph



1.19 Environment strategy and configuration management {#sec-13}

1.19.1 Objective

You **MUST** ensure promotions of **immutable artifacts** across environments are consistent, auditable, and repeatable, while separating code deployment from runtime configuration when doing so reduces risk.

1.19.2 Scope

This section applies to any repository or pipeline that builds, publishes, deploys, or promotes a production **Artifact**, and to configuration and data changes that can affect production behavior.

1.19.3 Policy

1.19.3.1 Environment definitions and parity

You **MUST** define a standard set of environments and treat them as promotion targets for the same **Artifact** digest/version.

Environment	Purpose	Allowed change types	Promotion source	Notes
`dev`	Fast feedback and integration	Frequent deploys, feature flags, experiments	Build pipeline output	Data may be synthetic; controls lighter but still logged
`staging`	Release candidate validation	Full verification, performance, security, release sign-off	Promoted from `dev` (same digest)	**SHOULD** mirror `prod` topology and critical integrations
`prod`	Customer-facing runtime	Controlled deploys only	Promoted from `staging` (same digest)	Production change windows **MUST** be enforced

You **MUST** document environment-specific deltas (e.g., external endpoints, scaling) as configuration, not code branches. You **SHOULD** maintain “parity” such that failures reproduce in `staging` without requiring rebuilds.

> [!Guardrail] You **MUST** “build once, promote many”: the `prod` deploy **MUST** use the same **Artifact** digest proven in `staging`; rebuilds for the same version are prohibited.

1.19.3.2 Configuration as code

You **MUST** manage runtime configuration as versioned, reviewable code (e.g., `config/`, `helm/`, `k8s/`, `terraform/`, or a dedicated config repo) and enforce **Branch protection** and `CODEOWNERS` approvals on production-affecting configuration.

You **MUST** ensure every configuration change is traceable to:

- a change request/ticket,
- an approved PR (with required checks),
- an auditable deploy/promotion record (who/when/what).

You **SHOULD** decouple configuration rollout from code rollout via **Feature flags** or environment configuration, provided audit logging is maintained.

> [!Pitfall] Treating config edits in a UI as “not code” breaks auditability. If you use a config UI (feature flag console, secrets manager UI), you **MUST** export logs and link changes to the change request.

1.19.3.3 Secrets management

You **MUST NOT** store secrets in Git (including encrypted blobs without approved key management). You **MUST** use an approved secrets manager and retrieve secrets at runtime or deploy-time using short-lived credentials where feasible.

You **MUST** define and enforce:

- rotation policy and owners,
- access controls (least privilege),
- audit logging for read/write operations,
- break-glass process for production.

> [!Exception] If a legacy system cannot integrate with the approved secrets manager, an exception **MUST** be recorded with compensating controls (restricted access, rotation procedure, monitoring) and a remediation timeline.

1.19.3.4 Production configuration change control

Production configuration changes **MUST** follow the same governance as production deploys:

- occur only in defined change windows, except **Emergency change**,

- include a rollback plan (e.g., revert config version, disable flag),
- include separation of duties (maker/checker) via required approvals.

Feature flag changes in `prod` **MUST** be treated as production changes and **MUST** be auditable (flag name, old/new value, actor, timestamp, linked ticket).

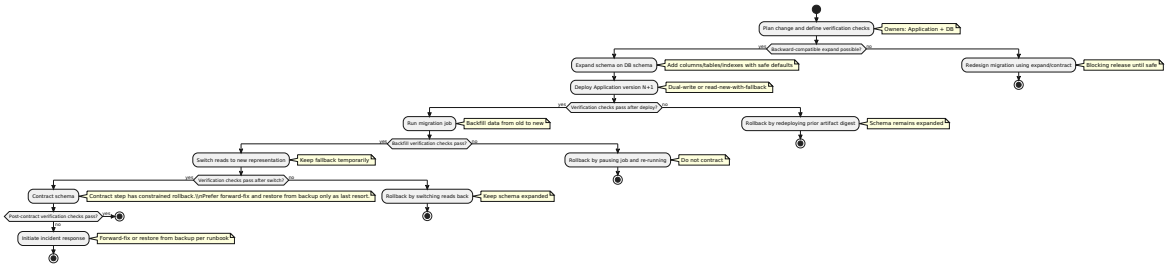
1.19.3.5 Database migration policy (expand/contract) and compatibility

You **MUST** design database changes to be safe under rolling deployments and mixed-version operation.

You **MUST** use **Expand/contract** for schema changes that would otherwise break backward compatibility:

- Expand steps **MUST** be backward-compatible with Application version N (current).
- Code enabling new schema usage **MUST** be deployable while old code may still be running.
- Contract steps (dropping/removing old schema) **MUST** occur only after verification and **MUST** be treated as higher-risk because rollback is constrained.

You **SHOULD** require explicit verification checkpoints between each phase, and you **MUST** document rollback constraints (especially for contract).



1.19.3.6 Observability requirements per environment

You **MUST** implement baseline observability in all environments to support promotion decisions and audits:

- logs, metrics, and traces **MUST** include the deployed **Artifact** version/digest and commit SHA metadata,
- alerts **SHOULD** be enabled in `staging` for release validation (at least smoke/health SLOs),
- `prod` dashboards and alerting **MUST** support go/no-go and rollback decisions.

You **SHOULD** standardize a minimum verification set used at each promotion gate (health checks, key business metrics, error rates, latency).

1.19.4 Rationale

- Promoting the same **immutable artifact** across `dev` → `staging` → `prod` provides reproducibility and audit integrity.
- Config-as-code and secrets governance enable traceability and separation of duties under SOX-like controls.
- **Expand/contract** minimizes downtime and prevents breakage during mixed-version rolling deployments.
- Environment parity and observability reduce change failure rate and improve MTTR by making issues detectable and diagnosable before `prod`.

1.19.5 Examples/Checklist

1.19.5.1 Promotion and configuration checklist

1. You **MUST** confirm the `staging` artifact digest equals the intended `prod` artifact digest.
2. You **MUST** confirm production change window eligibility or file an **Emergency change** record.
3. You **MUST** confirm configuration changes are reviewed, approved, and linked to the ticket.
4. You **MUST** confirm secrets are referenced via the secrets manager and not present in the repo.
5. You **MUST** confirm DB changes follow **Expand/contract** and contract is not scheduled until verification is complete.
6. You **SHOULD** confirm dashboards/alerts show stable health post-deploy before proceeding to wider rollout.

   [!Pitfall] Contracting schema immediately after switching reads often fails when delayed jobs or older instances still access old columns. You **SHOULD** enforce a “cooldown period” and validate no remaining reads/writes to old schema before contract.

1.19.5.2 Decision record: optional variants (release trains vs on-demand)

Decision: Choose whether `staging` promotions to `prod` occur on a fixed **Release train** cadence or on-demand.

Variant	You choose this when…	Additional requirements
Release train	You need predictable batch windows, coordinated comms, or heavy compliance scheduling	You **MUST** define cutoff, stabilization rules, and go/no-go criteria; you **SHOULD** publish a calendar
On-demand	You optimize for lead time and can validate continuously	You **MUST** still respect production windows; you **SHOULD** use feature flags and progressive delivery for risk

You **MUST** record the chosen variant per service (or platform) and the criteria that justify it (risk profile, deploy frequency, incident history, regulatory constraints).

1.20 Release documentation: notes, changelogs, and artifacts {#sec-14}

1.20.1 Objective

You **MUST** produce consistent, auditable release documentation that links each **Release** (signed tag) to its **Artifact**(s), checks, approvals, and **Promotion** history, so a third party can reconstruct “what changed, who approved it, what was deployed, and when”.

1.20.2 Scope

This section applies to any in-scope repository that:

- builds or publishes a production **Artifact**, or
- participates in CI/CD that can deploy/promote to production, or
- materially affects production outcomes (tooling repos **SHOULD** comply).

1.20.3 Policy

1.20.3.1 Release record (system of record)

You **MUST** create a release record for every production-bound release tag `vMAJOR.MINOR.PATCH`.

The release record **MUST** include, at minimum:

Field	Requirement
Release tag	**MUST** be `vMAJOR.MINOR.PATCH`, annotated, CI-created, and signed where supported/mandated
Commit SHA	**MUST** be the exact commit the tag points to
Change request / ticket	**MUST** link to the authoritative change record(s)
PR references	**MUST** link PR(s) included; backports **MUST** link original PR/SHAs
Artifact identifiers	**MUST** include immutable artifact version(s) and digest(es) (e.g., image `@sha256:...`)
Build provenance	**MUST** include build run ID and provenance attestation reference (or documented equivalent)
SBOM	**MUST** include SBOM reference/ location where required by policy or regulation
Required checks	**MUST** record pass/fail evidence for required gates at release time
Promotion log	**MUST** record environment-by-environment **Promotion** approvals and timestamps
Rollback plan	**MUST** specify the rollback method and last known-good version

Field	Requirement
Known issues / risks	**MUST** capture known issues, mitigations, and monitoring notes
Production window	**MUST** record the planned window; exceptions **MUST** follow Emergency change policy

> **Guardrail:** You **MUST NOT** rebuild or “republish” the same version. If content changes, you **MUST** cut a new tag/version and produce a new immutable **Artifact**.

1.20.3.2 Release notes (human-facing communication)

You **MUST** publish release notes for every release. Release notes **MUST** be generated from a standard template and stored in a durable location (e.g., release entry in the VCS platform plus a repository file).

Release notes **MUST** include:

- Summary/highlights
- Customer-impacting changes (behavioral changes, migrations, deprecations)
- Fixes (bug/security)
- Operational notes (runbook updates, feature flags, configs)
- Known issues
- Risks and mitigations
- Rollback instructions (including target version/artifact digest)

Release notes **SHOULD** be consumable by both engineering and operations audiences and **SHOULD** explicitly state whether changes are behind a **Feature flag**.

> **Pitfall:** “Changelog-only” releases without operational notes frequently fail audits because they omit approvals, rollback, and promotion evidence. You **MUST** publish both release notes and a release record.

1.20.3.3 Changelog generation (machine-assisted, policy-aligned)

You **MUST** maintain a changelog that is derivable from commits/PR metadata and supports audit traceability.

Changelog entries **MUST** be generated from:

- Conventional Commits (or documented equivalent), and/or
- PR labels that map to change categories (e.g., `type:feature`, `type:fix`, `type:security`, `type:breaking`)

Changelog generation **MUST**:

1. Reference PR number(s) and associated ticket/change request ID(s).
2. Distinguish **Backport** entries (and link original PR/SHA).
3. Highlight breaking changes and migrations.

Changelog generation **SHOULD** be automated in CI and **SHOULD** fail the release workflow if required metadata is missing (e.g., no ticket link).

> **Exception:** If you cannot use Conventional Commits, you **MUST** document the alternative scheme and **MUST** demonstrate equivalent traceability (ticket → PR → commit → build → artifact → promotion).

1.20.3.4 Artifact repository structure and retention

You **MUST** store published **Artifacts** in an approved registry/repository with immutable addressing (version plus digest) and access logging.

Artifact storage **MUST** support:

- immutable retention for at least the audit-required period (org-defined; **MUST** be documented),
- the ability to retrieve the exact artifact for any released tag,
- separation of environments via **Promotion** (same digest promoted, not rebuilt).

Recommended artifact coordinates **SHOULD** be standardized:

- Container: `registry.example.com/<org>/<service>:vMAJOR.MINOR.PATCH` plus immutable digest `@sha256:...`
- Packages: `<name>-<version>.tgz` (or ecosystem standard) with checksum
- SBOM/provenance: stored alongside the artifact and referenced by the release record

> **Guardrail:** Environment-specific builds (e.g., “prod build”) are prohibited. You **MUST** promote the same immutable artifact digest across environments.

1.20.3.5 SBOM generation and publication

If required by policy/regulation or if the repo produces production **Artifacts**, you **MUST** generate an SBOM as part of the build/release pipeline and publish it to an approved location.

SBOM handling **MUST**:

- be tied to the exact artifact digest/version,
- be immutable once published,
- be referenced by the release record.

SBOM handling **SHOULD** include license and vulnerability metadata where tooling supports it.

1.20.4 Rationale

- Auditability requires a deterministic chain: change request → PR/review → build/test → signed release tag → immutable artifact → promotion approvals → production deploy.
- Standardized release notes reduce operational risk (clear rollback, known issues, and monitoring guidance).
- Automated changelog generation reduces human error and increases completeness, especially for **Backport** and hotfix scenarios.
- SBOM and provenance support supply-chain integrity and incident response (what is running, and what it contains).

1.20.5 Examples/Checklist

1.20.5.1 Release documentation checklist (per `vMAJOR.MINOR.PATCH`)

1. You **MUST** create (or validate) signed tag `vMAJOR.MINOR.PATCH` pointing to the intended commit SHA.
2. You **MUST** ensure the CI build produces immutable **Artifacts** and captures digest(es).
3. You **MUST** publish release notes using the standard template (see Appendix).
4. You **MUST** publish/update `CHANGELOG.md` (or equivalent) with entries linked to PRs/tickets.
5. You **MUST** attach or link SBOM/provenance artifacts (where required).

6. Release Manager **MUST** ensure the release record includes approval evidence and promotion timestamps.
7. You **MUST** verify rollback instructions reference a known-good version/digest.

1.20.5.2 Minimal release notes template (required sections)

- `Summary`
- `Highlights`
- `Fixes`
- `Breaking changes / migrations`
- `Known issues`
- `Risks and mitigations`
- `Rollback plan`
- `Artifacts` (include digest identifiers)
- `Approvals and promotion status` (links to pipeline runs/approvals)

  Pitfall: Omitting artifact digests makes it impossible to prove that staging and production ran the same immutable **Artifact**. You **MUST** record digests.

1.20.6 Decision record (optional variants)

1.20.6.1 Variant A: Release notes stored only in VCS “Releases”

- **MAY** be used if the platform provides immutable history, access control, and export capability for audits.
- You **MUST** ensure links remain valid and that release notes include all required fields.
- **Criteria:** choose this when your audit process accepts the VCS platform as a durable system of record and your retention/export controls are proven.

1.20.6.2 Variant B: Release notes stored in-repo under `docs/releases/`

- **MAY** be used to guarantee long-term portability and reviewability via PRs.
- You **MUST** protect the directory via `CODEOWNERS` and ensure edits are traceable.
- **Criteria:** choose this when you require repository-native retention and PR-reviewed corrections/errata.

1.21 sec-15: Access control, protections, and compliance

1.21.1 Objective

You **MUST** implement access controls and evidence capture that provide end-to-end traceability (change request → code/review → build/test → Release → Promotion/Deploy → verification) and enforce separation of duties for production changes in SOX-like regulated contexts.

1.21.2 Scope

This section applies to any repository and CI/CD system that:

1. Publishes or promotes a production **Artifact**, or
2. Can deploy/promote to production, or
3. Can otherwise affect production behavior (internal tooling repos **SHOULD** comply when practical).

   [!Guardrail] `main` **MUST** be treated as releasable at all times unless an approved, recorded freeze exists.

1.21.3 Policy

1.21.3.1 Branch protection baseline (protected branches)

You **MUST** protect `main` and any `release/<major>.<minor>` and `hotfix/<major>.<minor>.<patch>-<ticket>` branches with branch protection that enforces:

Control	Requirement (auditable)	Notes/Evidence
Direct pushes	**MUST** be blocked	Git platform branch protection config snapshot
Required reviews	**MUST** require at least 2 approvals or `CODEOWNERS`-defined	PR record with reviewers

Control	Requirement (auditable)	Notes/Evidence
	approvals (whichever is stricter)	
Status checks	**MUST** require passing CI checks (build/test + security gates) before merge	CI status checks on PR
Linear history	**MUST** enforce linear history on protected branches	Merge policy (e.g., squash-only)
Signed changes	Signed tags **MUST** be used for Releases; signed commits **SHOULD** be required where supported	Signed tag verification, commit signature report
Merge method	PRs into `main` **MUST** use squash merge	Git settings + PR merge commit metadata
Shared branches	Shared branches **MUST NOT** be rebased	Operational control; enforce via education and tooling

> [!Pitfall] If you treat “Release” and “Deploy” as the same event, you will lose traceability and weaken rollback safety. You **SHOULD** decouple deploy from exposure using configuration or feature flags.

1.21.3.2 Separation of duties (SoD) for production

You **MUST** separate responsibilities such that no single individual can both approve code and unilaterally promote/deploy it to production without independent review/approval.

- A PR approver (code review) **MUST NOT** be the sole approver for the production deployment gate.
- Production deployment approval **MUST** be a distinct, auditable action in the deploy system (or CI/CD environment protection).

- Least privilege **MUST** be enforced: write access to protected branches and access to production deploy controls **MUST** be granted only to designated roles.

1.21.3.3 Release integrity and immutability controls

You **MUST** ensure that a Release is a signed, immutable reference that links code → artifact → deployments:

- Release tags **MUST** be annotated and signed (e.g., `vMAJOR.MINOR.PATCH`) and created by CI using protected credentials.
- Artifacts **MUST** be immutable and promoted across environments by digest/version without rebuilding.
- Provenance attestation and SBOM **MUST** be generated and stored for each released artifact version.
- Deployments **MUST** reference the artifact digest and the release tag/version, and capture the approver identity and timestamp.

1.21.3.4 Audit logging, retention, and evidence collection

You **MUST** retain evidence sufficient to reconstruct "who changed what, why, and what ran in production":

Evidence item	System of record	Minimum retention	Link key(s)
PR record (approvals, discussion, metadata)	Git platform	**MUST** meet regulatory retention policy (e.g., ≥ 7 years)	PR ID, commit SHA
CI logs + required checks	CI system	**MUST** meet retention policy	workflow run ID, commit SHA, tag
Signed tag verification	Git platform	**MUST** meet retention policy	tag `vMAJOR.MINOR.PATCH`, commit SHA
Provenance attestation (SLSA-like)	Artifact registry or attestation store	**MUST** meet retention policy	tag, artifact digest

Evidence item	System of record	Minimum retention	Link key(s)
SBOM	Artifact registry or SBOM store	**MUST** meet retention policy	artifact digest, version
Deployment record + approvals	Deploy system	**MUST** meet retention policy	artifact digest, environment, change window
Monitoring verification evidence	Observability platform	**SHOULD** meet retention policy; **MUST** retain incident links	release version, deployment ID

You **SHOULD** automate evidence collection into an “evidence bundle” per Release (or per production deployment), referencing immutable identifiers (tag/version and digest).

1.21.3.5 Controlled production change windows

- Production changes **MUST** occur only during defined windows.
- CI/CD **MUST** enforce window policy (e.g., time-based controls or deployment approval checks) and record the window used.

  [!Exception] Production changes outside defined windows **MUST** use the **Emergency change** process and record reason, approver, scope, rollback plan, and follow-up actions.

1.21.3.6 Break-glass (emergency access) controls

You **MUST** implement a break-glass mechanism that is auditable and time-bounded:

- Break-glass access **MUST** require a separate approval path (e.g., on-call manager) and be logged.
- Break-glass credentials **MUST** be rotated after use and access revoked when the emergency ends.
- A post-incident review **MUST** be completed, linking the emergency deployment record to PR/commits/tags and documenting follow-up remediation.

1.21.4 Rationale

These controls establish:

- Traceability across systems using stable identifiers (commit SHA, release tag, artifact digest).
- Strong change integrity via signed tags and immutable artifacts promoted without rebuilds.
- Separation of duties to reduce fraud/accidental risk under SOX-like auditability requirements.
- Reliable investigations and MTTR improvement through complete deployment and monitoring records.

1.21.5 Examples/Checklist

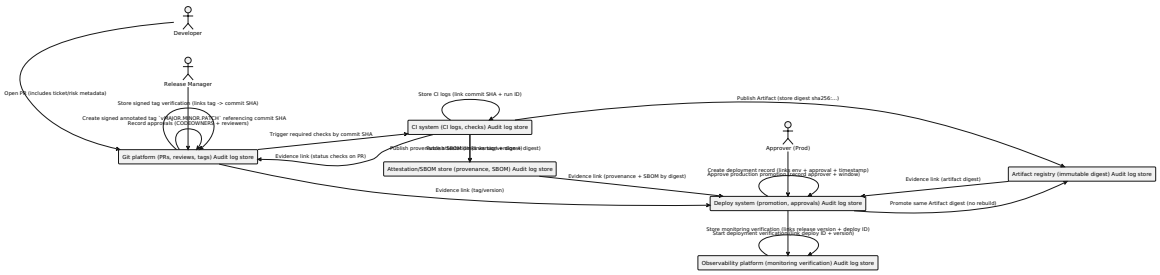
1.21.5.1 Configuration checklist (minimum viable compliance)

1. You **MUST** enable branch protection on `main`, `release/*`, and `hotfix/*` with required reviews, required checks, and blocked direct pushes.
2. You **MUST** enforce squash merges to `main` and linear history on protected branches.
3. You **MUST** configure CI to:
    1. Build once and publish an immutable artifact.
    2. Generate SBOM and provenance attestation.
    3. Create a signed annotated tag `vMAJOR.MINOR.PATCH` for Releases.
4. You **MUST** configure promotion so staging → production uses the same artifact digest (no rebuild).
5. You **MUST** require a manual, auditable production approval gate with separation of duties.
6. You **MUST** ensure deploy records include artifact digest, release tag, approver, timestamp, and environment.
7. You **MUST** define and enforce production change windows; outside-window deploys **MUST** follow Emergency change.

1.21.5.2 Decision record: optional “release train” variant

- **Decision**: Adopt a scheduled **Release train** (e.g., weekly) versus continuous releases from `main`.
- **Criteria**:
    - Choose release trains if you need predictable change windows, consolidated go/no-go, or coordinated multi-service releases.

    - Choose continuous releases if you can meet controls with frequent small changes and have robust automated verification.
- **Record**: If you deviate from the default approach, you **SHOULD** document rationale, cadence, cutoff rules, and evidence impacts in a tracked decision (e.g., `docs/decisions/release-train.md`).



1.22 sec-16 — Incident response integration and rollback strategy

1.22.1 Objective

You MUST integrate releases with incident response so restoration actions are fast, least-risk, and audit-ready. You MUST standardize rollback triggers, decisioning, execution, and post-incident feedback into CI/CD gates and runbooks.

1.22.2 Scope

This section applies to any service/library that ships a production **Artifact**, and to any CI/CD workflow that can **Deploy** or **Promote** an Artifact to production, including feature-flag/config systems that can change production behavior.

1.22.3 Policy

1.22.3.1 Incident–release linkage and traceability

- You **MUST** be able to correlate each production incident to the exact release tag `vMAJOR.MINOR.PATCH`, commit SHA, and Artifact digest deployed at the time of impact.
- You **MUST** record, in the incident ticket and the deployment record, the rollback mechanism used (feature flag/config, redeploy prior Artifact, or hotfix) and the approver(s) where required by environment controls (see sec-10/sec-11 dependencies).

- Production changes during an incident **MUST** follow the defined change window policy; if out-of-window, you **MUST** execute an **Emergency change** with documented reason, scope, approver, rollback plan, and follow-ups.

  Guardrail
  You MUST treat `main` as releasable during incident remediation; if a freeze is required, it MUST be approved and recorded, and the freeze scope MUST be explicit (what merges are blocked and why).

1.22.3.2 Rollback triggers and decision criteria

- You **MUST** define rollback triggers in operational terms (e.g., SLO burn, error rate, latency, data integrity signals) and keep them consistent with monitoring alerts used for go/no-go.
- You **MUST** select the rollback mechanism that restores service with the least additional risk, in this preference order when feasible:
  1. Feature flag/config rollback (fast, low blast-radius change)
  2. Redeploy a prior known-good Artifact digest (promote/deploy without rebuild)
  3. Hotfix (last resort when rollback cannot mitigate)
- You **MUST** explicitly consider database constraints before any rollback decision; destructive migrations and non-expand/contract changes constrain rollback options and may force hotfix-forward.

  Pitfall
  If you “roll back” by rebuilding an older version, you break immutability and auditability. You MUST redeploy the prior Artifact digest that was already published.

1.22.3.3 Rollback mechanisms and constraints

- Feature flag/config changes affecting production behavior **MUST** be treated as production changes (approvals, change window compliance, and auditable logs of who/what/when).
- Artifact rollback **MUST** redeploy a previously published immutable Artifact digest via the Deployment system; you **MUST NOT** rebuild the same version.
- Database rollback:
  - You **SHOULD** design schema changes using **expand/contract** to preserve rollback feasibility.
  - You **MUST** document “no safe rollback” conditions for DB changes in the PR (e.g., destructive migration), including forward-only remediation steps.

Exception
If a safe rollback is not technically possible (e.g., irreversible data migration), you MAY proceed with a forward-fix approach, but you **MUST** document the constraint, mitigation plan, and verification criteria in the incident record and the hotfix/release notes.

1.22.3.4 Communications and recovery declaration

- The On-call **MUST** communicate rollback intent, selected mechanism, and expected user impact to the incident channel before execution (or immediately after if time-critical).
- Recovery **MUST** be declared only after verification gates pass (monitoring stabilizes and functional checks confirm), and the incident timeline **MUST** include the exact time of change and time of verified recovery.
- You **MUST** hand off release-context to incident responders during active releases: current release tag, rollout stage, feature flags involved, and any known risky components.

1.22.3.5 Post-incident feedback loop into delivery controls

- You **MUST** create retro actions that map to concrete control improvements (tests, monitors, runbooks, gates, feature-flag defaults, rollout policies).
- You **SHOULD** update gating in CI/CD (sec-13) when incidents reveal missing coverage (e.g., add regression test, add DAST where applicable, strengthen canary verification).
- You **MUST** track incident-related delivery metrics (change failure rate, MTTR) and tie remediation work to measurable reductions over subsequent monthly reporting.

1.22.4 Rationale

Rollback is a controlled operational activity that must optimize for time-to-restore while preserving SOX-like auditability. Standardizing rollback decisioning reduces risk during high-pressure incidents, ensures separation of duties is respected, and reinforces the “build once, promote many” model by preventing unsafe rebuilds and undocumented production behavior changes.

1.22.5 Examples/Checklist

1.22.5.1 Rollback decision checklist (during incident)

1. Confirm incident scope: severity, blast radius, and whether customer impact is ongoing.
2. Identify current production release: tag `vMAJOR.MINOR.PATCH`, commit SHA, Artifact digest.
3. Determine feasibility of feature flag/config rollback:
    - Is the behavior behind a flag/config?
    - Is the flag change auditable and within change window (or Emergency change)?
    - Execute change; verify recovery signals.
4. If not feasible/effective, determine feasibility of redeploy prior Artifact:
    - Select last known-good Artifact digest (already published).
    - Promote/redeploy without rebuilding; verify recovery signals.
5. If rollback cannot mitigate, initiate hotfix:
    - Cut `hotfix/<major>.<minor>.<patch>-<ticket>` from the production tag.
    - Apply minimal fix; approvals and gates remain enforced (expedited, not bypassed).
    - Release via signed tag and promote Artifact through environments as required.
6. Declare recovery only after verification gates pass; record timeline and decisions.

1.22.5.2 Required incident record fields (audit-ready)

Field	Requirement
Incident ID and severity	**MUST**
Impact window (start/end)	**MUST**
Production release tag `vX.Y.Z`	**MUST**
Commit SHA + Artifact digest	**MUST**
Rollback mechanism used	**MUST**
Approver(s) and change window compliance	**MUST**

Field	Requirement
Verification evidence (dashboards/ checks)	**MUST**
Follow-up actions mapped to controls/ tests/gates	**MUST**

1.22.6 Decision record (optional variants)

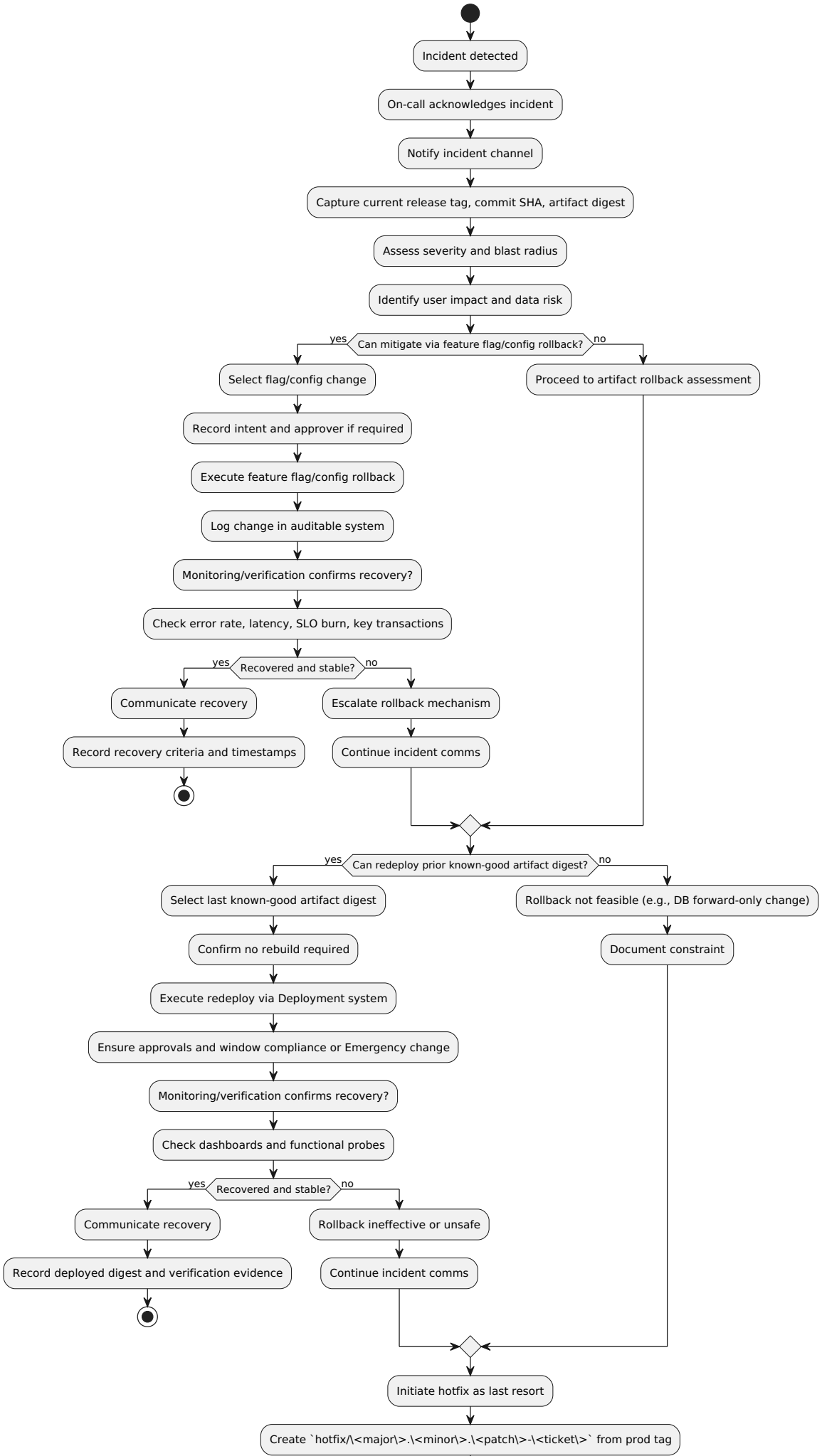
1.22.6.1 Variant: “Rollback-first” vs “Hotfix-first” posture

Decision: Teams MAY adopt a “rollback-first” default posture, with a documented exception to “hotfix-first” only when rollback is unsafe or ineffective.

Criteria	Rollback-first (recommended)	Hotfix-first (exceptional)
Artifact rollback available	Prefer redeploy prior digest	Only if prior digest also broken/unsafe
Feature flag/config available	Prefer flag/config rollback	Only if flag/config not present or too risky
DB changes reversible	Favor rollback	Favor hotfix-forward when irreversible
Regulatory/audit constraints	Strong fit (minimize change)	Requires stronger documentation and approvals

> **Guardrail**
> Even under “hotfix-first,” you **MUST** preserve immutability (new version, new Artifact) and you **MUST** forward-port fixes to `main` and active `release/<major>.<minor>` branches.

1.22.7 Diagram — Rollback decision flow during incidents


Incident detected
On-call acknowledges incident
Notify incident channel
Capture current release tag, commit SHA, artifact digest
Assess severity and blast radius
Identify user impact and data risk
Can mitigate via feature flag/config rollback?
yes
no
Select flag/config change
Record intent and approver if required
Execute feature flag/config rollback
Log change in auditable system
Monitoring/verification confirms recovery?
Check error rate, latency, SLO burn, key transactions
Recovered and stable?
yes
no
Communicate recovery
Record recovery criteria and timestamps
Escalate rollback mechanism
Continue incident comms
Proceed to artifact rollback assessment
Can redeploy prior known-good artifact digest?
yes
no
Select last known-good artifact digest
Confirm no rebuild required
Execute redeploy via Deployment system
Ensure approvals and window compliance or Emergency change
Monitoring/verification confirms recovery?
Check dashboards and functional probes
Recovered and stable?
yes
no
Communicate recovery
Record deployed digest and verification evidence
Rollback ineffective or unsafe
Continue incident comms
Rollback not feasible (e.g., DB forward-only change)
Document constraint
Initiate hotfix as last resort
Create `hotfix/\<major\>.\<minor\>.\<patch\>-\<ticket\>` from prod tag

1.23 sec-17 — Common workflows (cookbook)

1.23.1 Objective

You standardize day-to-day Git and release operations so that every production change is traceable (change request → code → review → build/test → **Promotion** → deploy), auditable (SOX-like), and repeatable while preserving the “build once, promote many” model for **immutable artifacts**.

1.23.2 Scope

Applies to all repositories in scope that build/publish/promote a production **Artifact** or participate in CI/CD that can affect production outcomes. Tooling repositories **SHOULD** follow these workflows when they influence production outcomes.

1.23.3 Policy

- You **MUST** use protected branches and PR-based merges for `main` and any `release/*` or `hotfix/*` branches; direct pushes are prohibited.
- You **MUST** ensure `main` is releasable at all times unless a freeze is approved and recorded (scope, dates, exit criteria).
- You **MUST** build an **immutable artifact** once per version and **MUST** promote the same artifact digest/version across environments without rebuilding.
- You **MUST** use mandatory code reviews, required checks, and `CODEOWNERS` approvals for protected branches.
- You **MUST** create signed releases (signed annotated tags) for `vMAJOR.MINOR.PATCH` and ensure the tag maps to the commit that produced the promoted artifact and provenance/SBOM.
- You **MUST** deploy to production only during defined change windows unless executing an **Emergency change** with documented reason, approver, scope, rollback plan, and follow-up actions.
- You **MUST** use squash merge for feature PRs to `main`.
- You **MUST** backport via `git cherry-pick` and record provenance (original PR/link/SHAs) and labeling.

1.23.4 Rationale

Cookbook workflows reduce variance, prevent rebuild-based drift between environments, and provide consistent evidence for audit (separation of duties, approvals, signed releases, and traceable promotions). They also minimize

operational risk by enforcing change windows and providing clear emergency and rollback paths.

1.24 Feature → PR → merge → deploy (normal change)

1.24.1 Objective

You deliver changes safely by integrating frequently into `main`, producing a single immutable artifact, and promoting it across environments with auditable approvals.

1.24.2 Policy

- You **MUST** branch from `main` using `feature/<ticket>-<slug>`.
- You **MUST** open a PR and obtain required `CODEOWNERS` approvals and passing checks before merge.
- You **MUST** squash-merge feature PRs into `main`.
- You **MUST** publish an immutable artifact from CI and promote the same artifact to staging and production (no rebuilds).
- You **MUST** comply with production change windows (or use the **Emergency change** path).

1.24.3 Examples/Checklist

1.24.3.1 Procedure (copy/paste)

1. Create branch from `main`:
    1. `git checkout main`
    2. `git pull --ff-only`
    3. `git checkout -b feature/<ticket>-<slug>`
2. Develop and commit using Conventional Commits (or documented equivalent):
    1. `git add -A`
    2. `git commit -m "feat: <summary> (<ticket>)"`
3. Push and open PR:
    1. `git push -u origin feature/<ticket>-<slug>`
    2. Open PR → target `main` → include ticket link, risk notes, test evidence.
4. Ensure PR gates pass (required checks + approvals).
5. Merge using squash:
    1. Merge method: “Squash and merge”

6. CI builds and publishes artifact:
    - Record: commit SHA, artifact version, artifact digest, SBOM/provenance link.
7. Promote artifact:
    1. Deploy/promote to dev/test (automated)
    2. Deploy/promote to staging (automated + verification)
    3. Approve production (manual, auditable, separation of duties)
    4. Deploy/promote to production (within change window)

Guardrail
You **MUST** promote the *same* artifact digest to staging and production. If the digest changes between environments, the deployment is non-compliant (rebuild drift) and **MUST** be blocked.

Pitfall
Rebasing shared branches or force-pushing a PR branch can break review/audit traceability. You **MUST NOT** rebase shared branches; prefer additional commits or close/reopen PR if necessary.

1.25 Cut release branch → stabilize → tag → promote (standard release)

1.25.1 Objective

You stabilize a planned release without blocking ongoing work on `main`, while keeping traceability and immutability guarantees.

1.25.2 Policy

- You **MUST** cut `release/<major>.<minor>` from `main` at an explicit cutoff point (time/commit) and record it.
- After cutoff, the release branch **MUST** accept only stabilization fixes; no new features.
- Fixes **MUST** land on `main` first and then be backported to `release/<major>.<minor>` via `git cherry-pick` (unless an exception is approved and recorded).
- The release **MUST** be created as a signed annotated tag `vMAJOR.MINOR.PATCH` by CI and tied to artifact provenance and SBOM.
- Promotion to production **MUST** occur within change windows (or follow **Emergency change** controls).

1.25.3 Decision record (optional variants)

1.25.3.1 Variant: Release train cadence

- You **MAY** use a release train (scheduled cadence) if:
  - The organization needs predictable bundling and coordinated go/no-go.
  - Stabilization overhead is acceptable and measured (lead time impact documented).
- You **SHOULD** prefer trunk-based continuous release when:
  - Feature flags and progressive delivery are mature.
  - Stabilization branches increase lead time or change failure rate.
- Record for each train:
  - Cutoff timestamp/commit, included PRs, excluded PRs, and go/no-go approvers.

1.25.4 Examples/Checklist

1.25.4.1 Procedure (copy/paste)

1. Cut release branch:
    1. `git checkout main`
    2. `git pull --ff-only`
    3. `git checkout -b release/<major>.<minor>`
    4. `git push -u origin release/<major>.<minor>`
2. Enforce branch protection on `release/<major>.<minor>` (PR-only + required checks + `CODEOWNERS`).
3. Stabilization workflow (repeat as needed):
    1. Fix on `main` via normal PR workflow.
    2. Backport to release branch:
        1. `git checkout release/<major>.<minor>`
        2. `git pull --ff-only`
        3. `git cherry-pick -x <commit_sha>`
        4. `git push`
        5. Open PR → target `release/<major>.<minor>` → include original PR link/SHAs.
4. Create release candidate build (CI):
    - CI builds artifact once; store artifact digest/version; attach SBOM/provenance.

5. Tag release (CI-only):
    - `git tag -a v<major>.<minor>.<patch> -m "Release v<major>.<minor>.<patch>"`
    - Tag **MUST** be signed where supported/mandated and **MUST NOT** be moved.
6. Promote the tagged artifact dev → staging → production with approval gates.

Exception
A fix **MAY** be applied directly to `release/<major>.<minor>` only if approved and recorded (reason, risk, approver), and it **MUST** be forward-ported to `main` immediately after (same day unless incident constraints prevent it).

Pitfall
Cutting a release branch while `main` is unstable violates the releasable-`main` guardrail. If instability requires a freeze, it **MUST** be approved and recorded with exit criteria.

1.26 Hotfix from production → deploy → forward-port (expedited patch)

1.26.1 Objective

You mitigate a production issue with minimal scope, preserve auditability, and ensure the fix is not lost by forward-porting.

1.26.2 Policy

- You **MUST** cut `hotfix/<major>.<minor>.<patch>-<ticket>` from the production release tag (or the exact commit deployed).
- You **MUST** produce a new signed patch tag `vMAJOR.MINOR.PATCH+1` (or next patch) tied to the hotfix artifact.
- You **MUST** follow expedited approvals appropriate for hotfixes, maintaining separation of duties and documentation.
- You **MUST** forward-port the fix to `main` (and to any active `release/<major>.<minor>` branch) via PR after production stabilization.

1.26.3 Examples/Checklist

1.26.3.1 Procedure (copy/paste)

1. Identify current production tag and artifact digest:
    - Example tag: `v1.4.2`
2. Create hotfix branch from the production tag:
    1. `git fetch --tags`
    2. `git checkout -b hotfix/1.4.3-<ticket> v1.4.2`
3. Implement minimal fix; commit:
    1. `git add -A`
    2. `git commit -m "fix: <summary> (<ticket>)"`
4. Push and open PR targeting `hotfix/1.4.3-<ticket>` integration target (repo-specific) or directly target `main` only if policy allows; typically:
    1. `git push -u origin hotfix/1.4.3-<ticket>`
    2. Open PR → target `hotfix/1.4.3-<ticket>` (or `release/<major>.<minor>` if your hotfixes are applied there) → include incident reference and rollback plan.
5. CI builds and publishes artifact once; verify in staging; obtain hotfix production approval (auditable).
6. Tag the hotfix release (CI-only):
    - `git tag -a v1.4.3 -m "Hotfix v1.4.3 (<ticket>)"`
7. Promote tagged artifact to production within a window; if out-of-window, use **Emergency change** controls.
8. Forward-port:
    1. Cherry-pick into `main`:
        - `git checkout main`
        - `git pull --ff-only`
        - `git cherry-pick -x <hotfix_commit_sha>`
        - `git push`
        - Open PR → target `main`
    2. If `release/1.4` exists and is active, cherry-pick there as well.

Guardrail
The hotfix branch **MUST** start from the production tag/commit to ensure the patch applies exactly to what is running in production.

Pitfall
Applying a hotfix only on `main` and then deploying it as a “patch” breaks traceability to the production baseline. Hotfixes **MUST** be cut from the production tag/commit and then forward-ported.

1.27 Backport a fix (main → release branch)

1.27.1 Objective

You ship a targeted fix in an older supported release line without introducing unreviewed divergence.

1.27.2 Policy

- You **MUST** land the fix on `main` first (unless an approved exception exists).
- You **MUST** use `git cherry-pick -x` to backport and preserve provenance.
- You **MUST** link the backport PR to the original PR/commit SHAs and apply backport labeling.

1.27.3 Examples/Checklist

1.27.3.1 Procedure (copy/paste)

1. Ensure fix is merged to `main` and identify the commit SHA to backport.
2. Create a backport branch off the release branch:
    1. `git checkout release/<major>.<minor>`
    2. `git pull --ff-only`
    3. `git checkout -b backport/<ticket>-to-<major>.<minor>`
3. Cherry-pick with provenance:
    1. `git cherry-pick -x <commit_sha>`
4. Push and open PR:
    1. `git push -u origin backport/<ticket>-to-<major>.<minor>`
    2. PR target: `release/<major>.<minor>`; include original PR link, commit SHAs, test evidence.
5. Complete required checks/approvals; merge via the repo's protected-branch merge policy.
6. Ensure release notes include the backport reference and ticket.

Exception
If the fix cannot land on `main` first due to incompatibility, you **MAY** backport first only with recorded rationale and an explicit follow-up item to implement an equivalent fix on `main` (or document why it is not applicable).

1.28 Revert a change (safe rollback via Git)

1.28.1 Objective

You remove a problematic change while preserving history and auditability.

1.28.2 Policy

- You **MUST** prefer `git revert` over rewriting history on protected branches.
- Reverts **MUST** follow the same PR/review/check requirements as any change to protected branches.
- If reverting a production deployment, you **MUST** coordinate with artifact promotion rules (redeploy prior artifact digest where feasible) and record incident/rollback evidence.

1.28.3 Examples/Checklist

1.28.3.1 Procedure (copy/paste)

1. Identify the merge commit (or offending commit) on the target branch (usually `main`).
2. Create a revert branch:
    1. `git checkout main`
    2. `git pull --ff-only`
    3. `git checkout -b feature/<ticket>-revert-<slug>`
3. Revert:
    - For a merge commit: `git revert -m 1 <merge_commit_sha>`
    - For a single commit: `git revert <commit_sha>`
4. Push and open PR:
    1. `git push -u origin feature/<ticket>-revert-<slug>`
    2. PR includes: reason, impact, verification plan, rollback notes.
5. Merge after approvals/checks; CI produces a new artifact version; promote per gates.

    Pitfall
    Reverting code without reverting configuration/feature flags can leave production behavior unchanged. You **SHOULD** explicitly address flags/config in the revert plan and verification steps.

1.29 Handle flaky tests (do not “green by chance”)

1.29.1 Objective

You prevent unreliable tests from eroding confidence in CI gates while maintaining delivery flow with auditable controls.

1.29.2 Policy

- You **MUST NOT** merge by repeatedly re-running CI to “get a green run” without addressing flakiness.
- You **MUST** track flaky tests with a ticket and link it in the PR when flakiness is detected.
- Disabling or quarantining tests **MUST** be time-bound and approved; the rationale and restoration plan **MUST** be recorded.

1.29.3 Examples/Checklist

1.29.3.1 Checklist

- Identify the flaky test signature (test name, suite, failure mode, run URL).
- Confirm reproducibility or nondeterminism (timing, order dependence, external dependency).
- Apply one of the allowed actions:
    1. Fix determinism (preferred).
    2. Add retries only with bounded retries and metrics.
    3. Quarantine test into a non-blocking job with explicit approval and a time-bound follow-up.

Guardrail
If a required gate is flaky, you **MUST** treat it as a release risk and either fix it or apply an approved quarantine with a restoration deadline.

Exception
In a time-critical stabilization or hotfix context, you **MAY** quarantine a flaky non-security test only with documented approval and a committed follow-up item; security gates (SAST/dependency/SBOM; DAST when applicable) **MUST NOT** be bypassed without an approved emergency exception record.

1.30 Quick-reference tables

1.30.1 Objective

You provide operators and engineers with consistent, glanceable references aligned to policy.

1.30.2 Policy

These tables define expected conventions; deviations **SHOULD** be documented with rationale and approvals where required.

1.30.3 Examples/Checklist

1.30.3.1 Branch types

Branch type	Pattern	Source	Lifetime	Merge target	Notes
Main/ Trunk	`main`	N/A	Long-lived	N/A	**MUST** remain releasable ; protected
Feature	`feature/<ticket>-<slug>`	`main`	Short-lived	`main`	**MUST** PR + squash merge
Release branch	`release/<major>.<minor>`	`main`	Temporary	N/A (stabilization)	Post-cutoff stabilization only; backports via `cherry-pick`
Hotfix branch	`hotfix/<major>.<minor`	prod tag	Temporary	`main` + active	Cut from prod tag; forward-

Branch type	Pattern	Source	Lifetime	Merge target	Notes
	`>.<patch>-<ticket>`			`release/*`	port required
Backport helper	`backport/<ticket>-to-<major>.<minor>`	`release/*`	Short-lived	`release/*`	Tracks a single backport scope

1.30.3.2 Environments (promotion model)

Environment	Purpose	Artifact rule	Typical gate
dev/test	Fast feedback	Same artifact version promoted	Automated checks
staging	Pre-prod verification	Same artifact digest/version promoted	Integration/ DAST when applicable
production	Customer impact	Same artifact digest/version promoted	Manual auditable approval + change window

1.30.3.3 Required checks (minimum)

Gate	Applies to	Requirement
Code review + `CODEOWNERS`	Protected branches	**MUST**
CI (lint/unit/integration as defined)	Protected branches	**MUST**
SAST	Protected branches	**MUST**

Gate	Applies to	Requirement
Dependency/license scan	Protected branches	**MUST**
SBOM generation	Releases/artifacts	**MUST**
DAST	Running services where applicable	**MUST** (or explicitly N/A with rationale)
Signed release tag `vMAJOR.MINOR.PATCH`	Releases	**MUST**
Manual production approval	Production promotion	**MUST** (separation of duties)

1.31 sec-18: Tooling standards and reference implementations

1.31.1 Objective

You MUST standardize CI/CD, security, and release tooling so that every production **Artifact** is built once, is traceable to a commit, and is promoted across environments with SOX-like auditability, signed releases, and enforced separation of duties.

1.31.2 Scope

This section applies to any repository that:

- builds/publishes/promotes a production **Artifact**, or
- participates in CI/CD that deploys/promotes to production.

Internal tooling repositories **SHOULD** comply when they can affect production outcomes (e.g., reusable pipelines, deployment tooling, policy-as-code).

1.31.3 Policy

1.31.3.1 1) Required repository files

You **MUST** include the following files at the repository root, and you **MUST** keep them current:

- `CODEOWNERS` (required approvers by path)
- `SECURITY.md` (reporting and handling security issues)
- `RELEASE.md` (release/promotion process, versioning, and gates)

You **SHOULD** also include:

- `CHANGELOG.md` (generated/curated; see tooling below)
- `docs/audit/` (links to controls evidence locations, if applicable)

> **Guardrail**: If `CODEOWNERS` is missing or empty for production-impacting paths, merges to protected branches **MUST** be blocked.

1.31.3.2 2) CI/CD implementation standards (templates + reuse)

You **MUST** implement CI/CD using reusable, versioned templates (e.g., GitHub reusable workflows, GitLab CI includes, Jenkins shared libraries) and you **MUST** pin them to immutable references (tag/commit SHA).

You **SHOULD** publish centrally managed “golden pipeline” templates that provide:

- consistent stage names and emitted evidence (logs, test reports, SBOM, provenance)
- standardized gates aligned to required checks and environments
- documented escape hatches with auditable approvals

> **Pitfall**: Referencing pipeline templates by a moving branch (e.g., `@main`) breaks auditability because past runs cannot be reconstituted.

1.31.3.3 3) Policy-as-code for enforcement

You **MUST** use policy-as-code to enforce:

- artifact immutability (no rebuild-on-promotion)
- required evidence presence (SBOM, scan results, provenance/attestations)
- environment promotion rules (prod window, approvals, separation of duties)

You **SHOULD** use OPA/conftest (or a documented equivalent) for:

- validating pipeline definitions and deployment manifests
- validating release metadata (SemVer, Conventional Commits-derived changelog entries)
- validating branch protection expectations (where platform APIs allow)

  Exception: If the platform cannot support policy-as-code enforcement for a control, you **MUST** document compensating controls in `RELEASE.md` and record periodic verification evidence.

1.31.3.4 4) Release automation and versioning

You **MUST** publish releases via automation (CI) that:

- creates a signed, annotated tag `vMAJOR.MINOR.PATCH`
- links the tag to the exact commit SHA used to build the **Artifact**
- attaches or references: SBOM, scan summaries, and provenance/attestations
- produces release notes derived from Conventional Commits (or the documented equivalent)

You **SHOULD** use:

- `semantic-release` (or equivalent) for SemVer calculation + release notes
- `conventional-changelog` tooling (or equivalent) for standardized changelogs

  Guardrail: Human-created/moved tags **MUST** be disallowed for release tags (enforce via permissions and/or pipeline checks).

1.31.3.5 5) SBOM, provenance, and signing standards

You **MUST** generate an SBOM for every published **Artifact** and store it alongside the artifact or in a registry-supported attachment mechanism.

You **MUST** produce and store build provenance/attestations sufficient to support supply-chain integrity expectations (SLSA-aligned where feasible).

You **SHOULD** sign:

- container images (e.g., Sigstore/cosign)
- release tags (Git signed tags)

- provenance attestations (keyless where supported, otherwise managed keys with rotation)

You **MUST** verify signatures during promotion to staging and production.

1.31.3.6 6) ChatOps for release operations (with guardrails)

You **MAY** enable ChatOps for routine release operations (e.g., “promote artifact X to staging”).

If you use ChatOps, you **MUST** implement:

- strong authentication (SSO), authorization (role-based), and audit logging
- command allowlists (no arbitrary script execution)
- separation of duties for production actions (e.g., requestor vs approver)
- enforcement of production windows and **Emergency change** flows

  Pitfall: ChatOps without environment-aware policy checks turns chat into an ungoverned production backdoor.

1.31.3.7 7) Reference pipeline pattern (build once, promote many)

You **MUST** structure pipelines so that the **Artifact** is built once, published, and then promoted across environments without rebuilding.

Recommended logical stages:

1. Validate (lint/unit)
2. Test (integration)
3. Secure (SAST, dependency/license, DAST when applicable or explicitly N/A)
4. Build and Publish immutable **Artifact** (store digest, SBOM, provenance)
5. Deploy/Verify dev (using the published digest)
6. Promote to staging (same digest)
7. Manual go/no-go approval (auditable; separation of duties)
8. Promote to prod (same digest; within window unless **Emergency change**)

1.31.4 Rationale

Standardized tooling and reference implementations reduce variance, prevent bypass of required controls, and improve auditability. Building once and promoting the same immutable digest ensures that “what was tested is what is deployed,” enabling reliable traceability from change request → commit → review → build → **Promotion** → production.

1.31.5 Examples/Checklist

1.31.5.1 Minimum tooling checklist (per repo)

- [ ] `CODEOWNERS`, `SECURITY.md`, `RELEASE.md` present at repo root
- [ ] Protected branches enforce PRs, required checks, and `CODEOWNERS` approvals
- [ ] Pipeline uses pinned, versioned templates/includes
- [ ] Conventional Commits enforced (lint/check)
- [ ] SBOM generated and stored per **Artifact**
- [ ] SAST + dependency/license scan executed per change
- [ ] Provenance/attestation produced and stored per **Artifact**
- [ ] Release tags v`MAJOR.MINOR.PATCH` created by CI and signed
- [ ] Promotions verify signatures and reuse the same digest (no rebuild)
- [ ] Production approval gate is manual, auditable, and enforces separation of duties
- [ ] Production deploys occur only during defined windows, except **Emergency change**

1.31.5.2 Required checks table (minimum baseline)

Check category	Minimum check	Evidence artifact	Enforcement point
Code quality	Lint + unit tests	test report	PR to protected branches
Integration	Integration tests	test report	PR to protected branches
Security (static)	SAST	SARIF/report	PR to protected branches
Supply chain	Dependency + license scan	report	PR to protected branches
SBOM	SBOM generation	SBOM file/ attachment	build/publish stage
Provenance	Build provenance/ attestation	attestation link/ file	build/publish stage

Check category	Minimum check	Evidence artifact	Enforcement point
Signing	Signed release tag + artifact signature	signature verification logs	promotion to staging/prod
Governance	Manual prod approval + window check	approval record	before prod promotion

1.31.5.3 Environments table (promotion-focused)

Environment	Deployment input	Allowed action	Minimum gate
dev	artifact digest	deploy/verify	automated checks only
staging	same artifact digest	promote + verify	signature + evidence verification
prod	same artifact digest	promote + verify	manual approval, window enforcement, signature + evidence verification

1.31.5.4 Decision record: Release automation variants (optional)

Variant A: Continuous release (tag on every merge to `main`)

- Criteria: high deployment frequency, strong test suite, low change failure rate
- You **MUST** keep `main` releasable and rely on feature flags for incomplete work

Variant B: Release trains (scheduled tags from `release/<major>.<minor>`)

- Criteria: regulated coordination, multi-team batching needs, fixed windows

- You **MUST** define cutoff, stabilization rules, and backport approach in `RELEASE.md`

Record your chosen variant in `RELEASE.md` with:

- selection criteria (why this variant fits)
- required exceptions (if any) and compensating controls

  Exception: If a repo cannot adopt automated tagging immediately, you **MUST** time-box the deviation, record the rationale, and implement a migration plan with an owner and date in `RELEASE.md`.

1.32 sec-19 — Governance: exceptions, reviews, and policy evolution

1.32.1 Objective

You **MUST** ensure deviations from this branching/release/promotion policy are explicitly requested, risk-assessed, approved, time-bounded where possible, and fully auditable; and you **MUST** ensure the policy itself remains current through controlled change management, periodic review, and communicated deprecations.

1.32.2 Scope

This section applies to all repositories in scope (publish/promote/deploy a production **Artifact**, or can affect production behavior via CI/CD). Repositories out of scope **MAY** adopt a lighter-weight variant, but **SHOULD** still follow review/signing controls where feasible.

1.32.3 Policy

1.32.3.1 1) Exception governance (deviations from policy)

- You **MUST** treat any deviation from a **MUST** requirement as an exception requiring documented approval.
- You **SHOULD** treat deviations from **SHOULD** guidance as exceptions with recorded rationale when they affect auditability, separation of duties, or production risk.
- Exception requests **MUST** be recorded in a durable system of record (e.g., ticket) and linked to:
  - the change request,

    - relevant pull request(s),
    - relevant release tag(s) (for releases),
    - and deployment/promotion records (for production impact).

Exception types

- **Temporary exception** (preferred): you **MUST** set an explicit expiry (date or condition) and a remediation plan to return to compliance.
- **Permanent exception**: you **MUST** document why the control is not feasible, what compensating controls exist, and when it will be reconsidered (at minimum at the next audit/maturity review).

Minimum required exception record fields (audit baseline) You **MUST** capture, at minimum:

Field	Requirement
Exception ID	**MUST** be unique and immutable once issued
Owner	**MUST** name a role (e.g., Engineering Manager) accountable for closure
Type	**MUST** be `temporary` or `permanent`
Policy reference	**MUST** cite section + requirement being waived
Scope	**MUST** list repo(s), branch(es) (e.g., `main`), pipeline(s), and environment(s) impacted
Risk assessment	**MUST** include impact, likelihood, and customer/financial/compliance considerations
Compensating controls	**MUST** be explicit (e.g., extra approvals, narrowed change scope)
Approvers	**MUST** include separation of duties (author cannot be sole approver)
Expiry & review date	**MUST** exist for temporary; **SHOULD** exist for permanent
Rollback/containment	

Field	Requirement
	MUST exist when production risk is present
Evidence links	**MUST** link to PRs, builds, SBOM/ provenance, promotions, deployments, and incident tickets if relevant

Guardrail
You **MUST** not approve an exception that makes `main` non-releasable unless a freeze is explicitly declared, recorded, and time-bounded with a re-enable plan.

Pitfall
Granting “one-time” exceptions without expiry and follow-up tasks becomes de facto permanent non-compliance and undermines SOX-like auditability.

1.32.3.2 2) Emergency changes (outside defined production windows)

- Production changes outside defined windows **MUST** follow the **Emergency change** process and include reason, approver, scope, rollback plan, and follow-up actions (including post-facto review).
- Emergency change records **MUST** be linked to the resulting code change, release (signed tag), **Artifact** version, and promotion/deploy evidence.
- You **MUST** perform a post-facto review within the agreed SLA (e.g., next business day) and capture corrective actions.

Exception
“Emergency change” is not a standing permission. Each instance **MUST** be individually approved and documented.

1.32.3.3 3) Reviews, audits, and maturity checkpoints

- You **MUST** run periodic compliance reviews covering:
  - branch protections and required checks on protected branches,
  - code review adherence and approver independence,
  - signed release tags and release provenance evidence,
  - promotion without rebuilding (immutability),
  - production change window adherence (including emergency exception usage).

- Findings **MUST** be tracked to closure with owners and due dates.
- Teams **SHOULD** maintain operational metrics to inform policy evolution (e.g., lead time for changes, change failure rate, MTTR).

1.32.3.4 4) Policy change management (versioning and evolution)

- The policy document **MUST** be versioned in source control and changes **MUST** be reviewed and approved (same review controls as other production-impacting changes).
- Each policy change **MUST** include:
  - a summary of changes,
  - rationale,
  - effective date,
  - migration guidance,
  - and deprecation timelines for superseded processes.
- You **MUST** announce policy changes to affected teams before the effective date, except for urgent security/compliance fixes (which **MAY** be effective immediately with after-the-fact broadcast).

Deprecations

- Deprecated procedures **MUST** specify:
  - last date for new adoption,
  - last supported date,
  - and required migration steps.
- Exceptions relying on deprecated procedures **MUST** be re-evaluated before the deprecation end date.

1.32.3.5 Decision record: Release train governance (optional variant)

Some teams may operate on a **Release train** cadence.

Option	You MAY choose when	Additional requirements
Continuous (no train)	risk is low; strong feature flagging; high automation	`main` **MUST** remain releasable; releases **MUST** be signed and promoted as immutable artifacts
Fixed release train	many teams share production windows;	train cutoffs/freezes **MUST** be declared; go/

Option	You MAY choose when	Additional requirements
	coordinated go/no-go needed	no-go approvers **MUST** be documented; backports **MUST** follow the same review/ signing controls

1.32.4 Rationale

Exception governance preserves traceability and separation of duties while enabling pragmatic delivery under constraints (e.g., legacy systems, tool gaps). Time-bounded exceptions prevent control erosion. Policy versioning and audits ensure SOX-like auditability, reproducibility via **immutable artifacts**, and consistent enforcement of controlled production windows, while allowing measured evolution based on operational feedback.

1.32.5 Examples/Checklist

1.32.5.1 Exception request checklist (temporary)

1. Create an exception ticket titled `Exception: <policy-ref> for <repo>`.
2. Include scope: `repo`, impacted branches (e.g., `release/1.2`), and environments (e.g., `prod`).
3. Provide risk assessment and compensating controls.
4. Set expiry date and list remediation tasks with owners.
5. Obtain approvals with separation of duties.
6. Link the ticket in the PR description and, if relevant, the release notes.

1.32.5.2 Permanent exception checklist

1. Document why compliance is infeasible and what alternative controls exist.
2. Define objective criteria for reconsideration (tooling milestone, architecture change, vendor roadmap).
3. Schedule re-review at the next maturity review (or earlier if risk changes).

1.32.5.3 Policy update checklist

1. Update the policy in version control and open a PR.

2. Ensure reviewers include governance owners (e.g., Release Manager, Security).
3. Add “effective date”, “migration guidance”, and “deprecation timeline” if applicable.
4. Announce changes and track migration work where needed.

2. Appendices: templates and checklists (sec-20)

2.1 Objective

You provide standardized, auditable templates and operational checklists that enforce consistent branching, review, release, and promotion practices for immutable **Artifacts**.

2.2 Scope

These appendices apply to any repository in scope (publishes/promotes a production **Artifact** or can affect production outcomes). Tooling repositories that affect production outcomes **SHOULD** adopt these templates.

2.3 Policy

- You **MUST** use standardized PR and release records for all changes that can reach production.
- You **MUST** record evidence of review, testing, security checks, and promotion approvals to meet SOX-like auditability.
- You **MUST** treat `main` as releasable unless a freeze is approved and recorded.
- You **MUST** release via an annotated, signed tag `vMAJOR.MINOR.PATCH` created by CI; tags **MUST NOT** be moved.
- You **MUST** promote the same immutable **Artifact** digest/version across environments without rebuilding.

2.4 Rationale

Templates reduce variance, ensure required compliance evidence is captured at the point of work, and make audits repeatable (who approved what, when, with which tests and which artifact digest).

> **Guardrail**
> If information is missing from a required template field, the PR/release **MUST NOT** be approved or promoted until the record is complete.

> **Pitfall**
> “We’ll add release notes later” creates an audit gap; you **MUST** capture release intent, risk, and rollback before production promotion.

2.5 PR template (pull request)

2.5.1 Objective

You standardize PR metadata so reviewers and approvers can assess risk, rollout, and evidence quickly.

2.5.2 Policy

- You **MUST** use this PR template (or an equivalent that captures all required fields).
- You **MUST** link to the tracking ticket and include rollout and rollback guidance.
- You **MUST** provide test evidence and explicitly mark security testing as performed or N/A with rationale.
- You **MUST** declare whether database changes follow expand/contract.

2.5.3 Examples/Checklist

Create `.github/pull_request_template.md`:

```
## Summary
- What changed:
- Why:

## Ticket / Change record
- Ticket: <link>
```

```
- Related incidents / problems: <link(s) or N/A>

## Risk assessment
- User impact if wrong:
- Blast radius:
- Change type: feature | fix | refactor | infra | security
- Backward compatibility: yes | no | N/A
- Data migration: none | expand/contract | other (describe)

## Rollout plan (deploy decoupled from release where possible)
- Feature flag(s): `flag_name` (default state):
- Config changes required: yes | no (details):
- Rollout steps:
  1.
  2.
- Canary/progressive delivery: yes | no | N/A (details):

## Rollback plan
- Primary rollback: redeploy prior Artifact version `vX.Y.Z` / digest `<
- Data rollback considerations:
- Manual steps (if any):

## Testing evidence
- Unit tests: pass (link to CI run)
- Integration tests: pass (link)
- E2E/DAST (if applicable): pass | N/A (rationale)
- Repro steps (for reviewers):
  1.
  2.

## Security and compliance
- SAST: pass (link)
- Dependency/license scan: pass (link)
- SBOM generated: yes (link/artifact ref)
- Secrets check: pass (link) / N/A (rationale)

## Review requirements
- CODEOWNERS paths impacted: yes | no
- Required approvers: @team or @role
- Notes for Release Manager (if any):
```

```
## Release notes (customer-facing if applicable)
- Added:
- Changed:
- Fixed:
- Security:
```

> **Exception**
> If a repository cannot support `.github/pull_request_template.md`, an equivalent template **MAY** be enforced via repository settings or a PR checklist bot, but the same fields **MUST** be captured.

2.6 Release notes template + go/no-go checklist

2.6.1 Objective

You standardize release records and formalize the go/no-go decision with required evidence and approvals.

2.6.2 Policy

- A release record **MUST** include: version, commit/tag, **Artifact** digest, provenance/SBOM references, included changes, and approvals.
- Production promotion **MUST** include a go/no-go decision by the Release Manager (or delegated approver per RACI) during a defined window, unless it is an **Emergency change**.

2.6.3 Examples/Checklist

2.6.3.1 Release notes template (per version)

Store in `docs/releases/vMAJOR.MINOR.PATCH.md` (or your release system equivalent):

```
# Release vMAJOR.MINOR.PATCH

## Release metadata
- Tag: `vMAJOR.MINOR.PATCH` (annotated, signed)
- Source branch: `main` | `release/X.Y` | `hotfix/X.Y.Z-<ticket>`
- Commit SHA: `<sha>`
- Artifact:
```

```
  - Type: container | package | binary
  - Registry URL: `<ref>`
  - Digest: `<sha256:...>` (immutable)
- Build provenance (SLSA attestation): `<link>`
- SBOM: `<link>`

## Scope
- Features:
- Fixes:
- Security:
- Breaking changes: yes | no (details)

## Risk and rollout
- Risk level: low | medium | high (rationale)
- Feature flags/config gates:
- Progressive delivery/canary: yes | no (details)
- Monitoring/alerts to watch:

## Validation evidence
- CI pipeline run: `<link>`
- Required checks summary:
  - Lint/unit: pass
  - Integration: pass
  - SAST: pass
  - Dependency/license: pass
  - DAST: pass | N/A (rationale)
- Manual verification (if any): `<steps and evidence>`

## Promotion record
- Dev deployed: yes (timestamp, link)
- Staging deployed: yes (timestamp, link)
- Production window: `<date/time window>`
- Go/no-go decision:
  - Decision: go | no-go
  - Approver (Release Manager): `<name>`
  - Timestamp: `<time>`
  - Notes:

## Rollback
- Rollback Artifact version/digest: `vX.Y.Z` / `<sha256:...>`
- Rollback steps:
```

1.
2.

2.6.3.2 Go/no-go checklist (copy into the release record)

- [ ] Tag `vMAJOR.MINOR.PATCH` created by CI and signature verified
- [ ] **Artifact** digest recorded and matches the promoted artifact in staging
- [ ] All required CI checks passed (or approved exception recorded)
- [ ] SBOM and provenance links attached
- [ ] CODEOWNERS approvals recorded; separation-of-duties satisfied
- [ ] Production change scheduled within defined window (or Emergency change record created)
- [ ] Rollback plan validated and last known-good version identified
- [ ] Monitoring plan confirmed; on-call notified if required

---

2.7 Hotfix checklist + post-incident checklist

2.7.1 Objective

You provide an expedited but controlled path for hotfixes while preserving auditability and forward-port discipline.

2.7.2 Policy

- Hotfix branches **MUST** be cut from the production tag being patched (e.g., `v1.2.3`).
- Hotfix changes **MUST** be minimal scope and include a rollback plan.
- The hotfix commit(s) **MUST** be forward-ported to `main` (and any active `release/X.Y`) after production stabilization.
- Out-of-window hotfix deployment **MUST** follow **Emergency change** controls.

2.7.3 Examples/Checklist

2.7.3.1 Hotfix execution checklist

1. Create branch from the production tag: `git checkout -b hotfix/MAJOR.MINOR.PATCH- vMAJOR.MINOR.PATCH``.
2. Implement minimal fix; update tests as needed.
3. Open PR to the active release line (or to `main` if no release branch):
    - Include incident link, impact, rollback, and validation evidence.

4. Ensure required checks pass (lint/unit, integration, SAST, dependency/license, SBOM; DAST if applicable).
5. Create release tag via CI: `vMAJOR.MINOR.(PATCH+1)` and record **Artifact** digest.
6. Promote the same immutable **Artifact** through environments; obtain manual approve-production.
7. Forward-port:
    - Cherry-pick hotfix commit(s) to `main` (and `release/X.Y` if applicable) with linkage to original PR/SHAs.

Pitfall
Applying “the fix” directly to `main` and deploying a different rebuild to production violates immutability; you **MUST** promote the same **Artifact** that was released.

2.7.3.2 Post-incident checklist (after hotfix)

- [ ] Root cause documented and linked to PR/release record
- [ ] Follow-up actions created with owners and due dates
- [ ] Metrics updated (MTTR, change failure rate, lead time)
- [ ] Preventive guardrails added (tests, alerts, branch protection, pipeline gates)
- [ ] Forward-port verified and linked

2.8 Branch naming regex and examples

2.8.1 Objective

You standardize branch naming for traceability and automation (CI routing, policy checks, and reporting).

2.8.2 Policy

- Branches **MUST** follow the approved patterns and include a ticket/reference.
- Shared branches **MUST NOT** be rebased; backports **MUST** use cherry-pick and preserve linkage.

2.8.3 Examples/Checklist

2.8.3.1 Approved branch types (table)

Branch type	Pattern (regex)	Example	Notes
Feature	`^feature\/[A-Z]+-[0-9]+-[a-z0-9-]+$`	`feature/ABC-123-add-timeout`	Short-lived; squash merge into `main`
Release	`^release\/[0-9]+\.[0-9]+$`	`release/1.4`	Stabilization only after cutoff
Hotfix	`^hotfix\/[0-9]+\.[0-9]+\.[0-9]+-[A-Z]+-[0-9]+$`	`hotfix/1.4.2-ABC-987`	Cut from prod tag; forward-port required
Chore (optional)	`^chore\/[A-Z]+-[0-9]+-[a-z0-9-]+$`	`chore/ABC-555-dep-bump`	Still requires PR + checks

> **Exception**
> If a repository uses a different ticket scheme (e.g., numeric only), the regex **MAY** be adapted, but the branch name **MUST** remain uniquely traceable to a work item and the deviation **SHOULD** be recorded.

2.9 Required checks quick-reference (appendix table)

2.9.1 Objective

You provide a minimal auditable checklist for protected branches and production promotion.

2.9.2 Policy

Protected branches (`main`, `release/*`) **MUST** require these checks unless an approved, recorded exception exists.

Check category	Minimum requirement	Evidence location
Code review	CODEOWNERS approval(s) + required reviewers	PR approvals
CI quality	Lint + unit tests	CI run link
Build	Immutable **Artifact** built once	Build logs + registry digest
Integration	Integration tests	CI run link
Security	SAST + dependency/ license scan	CI run link
Supply chain	SBOM generated + provenance attached	Release record links
DAST	When applicable; else explicit N/A	CI run link or release record
Governance	Manual approve-production + window compliance	Deployment approval record

---

2.10 Decision record (optional variant): release trains

2.10.1 Objective

You define when to use a release train vs. continuous release while preserving the same controls.

2.10.2 Policy

Teams **MAY** adopt release trains, but they **MUST** keep `main` releasable, use immutable artifact promotion, and maintain signed, versioned releases.

2.10.3 Decision record

- **Option A: Continuous release from `main`**
  - Criteria: low coupling, strong automated testing, frequent small changes, high confidence in progressive delivery.
- **Option B: Release train using `release/X.Y` stabilization**
  - Criteria: coordinated changes across components, regulatory timing needs, planned cutoffs and stabilization capacity.

Record the chosen option in `docs/release-strategy.md` with:

- Selected option, rationale, cadence/cutoff rules, and how freezes are approved/recorded.

2.11 RACI mapping (appendix table)

2.11.1 Objective

You clarify responsibilities for reviews, releases, and promotions under separation-of-duties.

2.11.2 Policy

Promotion to production **MUST** be approved by an authorized role distinct from the change author where required by policy.

Activity	Responsible	Accountable	Consulted	Informed
Implement change on `feature/*`	Developer	Engineering Manager	Security (as needed)	Team
Approve PR to `main`/ `release/*`	CODEOWNERS/ Reviewers	Engineering Manager	QA/Platform	Team
Create signed tag `vMAJOR.MINOR.PATCH` (via CI)	CI system	Release Manager	Security	Team

Activity	Responsible	Accountable	Consulted	Informed
Promote **Artifact** to staging	Platform/CI	Release Manager	Service Owner	Team
Go/no-go for production	Release Manager	Release Manager	Service Owner, Security	Stakeholders
Deploy/ promote to production (within window)	Platform/CI	Release Manager	On-call/SRE	Stakeholders
Emergency change approval (out-of-window)	Platform/CI	Incident Commander / Release Manager (per policy)	Security	Stakeholders
Backport/ cherry-pick to release branch	Developer	Release Manager	CODEOWNERS	Team

---

2.12 Appendix: quick copy/paste command snippets (non-normative)

2.12.1 Objective

You reduce operator error by standardizing common git actions.

2.12.2 Examples

1. Create feature branch: `git checkout -b feature/ABC-123-short-slug``
2. Create release branch (when approved): `git checkout -b release/1.4 main``
3. Create hotfix from tag: `git checkout -b hotfix/1.4.2-ABC-987 v1.4.2``

4. Backport via cherry-pick (record linkage): `git cherry-pick -x <commit-sha>`