Improving Development Process Discipline within Cloud Platform Engineering

Fixing Flaccid Scrum! Restoring Engineering Excellence, Delivery Confidence, and Cross-Team Alignment

| **Version** | **Date** | **Author** | **Change Description** |
| --- | --- | --- | --- |
| 1.0 | 2025-03-27 | Christian Turner | Initial draft |
| 1.1 | 2025-04-01 | Christian Turner | Added sections on security & compliance |
| 1.2 | 2025-04-15 | Christian Turner | Updated implementation roadmap |

Table of Contents

[1. Executive Summary 2](#_Toc194002890)

[1.1 Purpose of the document 2](#_Toc194002891)

[1.2 Summary of the current state 2](#_Toc194002892)

[1.3 What is being proposed 3](#_Toc194002893)

[1.4 Expected business and technical benefits 4](#_Toc194002894)

[2. Current State: Key Challenges 6](#_Toc194002895)

[2.1 Lack of Shared Understanding 6](#_Toc194002896)

[2.2 Poor Work Definition & Execution Discipline 6](#_Toc194002897)

[2.3 Ad Hoc Design and Development 7](#_Toc194002898)

[2.4 Lack of Security & Compliance Integration 8](#_Toc194002899)

[2.5 Documentation & Knowledge Gaps 8](#_Toc194002900)

[2.6 Consequences: Risk, Inefficiency, Rework 9](#_Toc194002901)

[3. Proposed Development Lifecycle 10](#_Toc194002902)

[3.1 End-to-End Workflow Overview (Design → Validation) 10](#_Toc194002903)

[3.2 Lifecycle Stages Explained 11](#_Toc194002904)

[3.2.1 Design + Team Review 11](#_Toc194002905)

[3.2.2 Proof of Concept (Optional) 12](#_Toc194002906)

[3.2.3 Development & Testing 13](#_Toc194002907)

[3.2.4 Deployment, Documentation, Security & Compliance 14](#_Toc194002908)

[3.2.5 Final Validation & Success Metrics 15](#_Toc194002909)

[3.3 Time Planning, Progress Tracking, and Reporting 16](#_Toc194002910)

[4. Engineering Principles and Governance 19](#_Toc194002911)

[4.1 Definition of Ready / Done 19](#_Toc194002912)

[4.2 Peer Review Expectations 20](#_Toc194002913)

[4.3 Use of Design Documents and Success Criteria 21](#_Toc194002914)

[4.4 Documenting for Reusability and Onboarding 22](#_Toc194002915)

[4.5 Security & Compliance Checkpoints 23](#_Toc194002916)

[4.6 Exceptions & Flexibility 24](#_Toc194002917)

[4.7 Continuous Improvement 25](#_Toc194002918)

[5. Strategic Alignment with Cloud Engineering Principles 27](#_Toc194002919)

[5.1 Pillar Overview 27](#_Toc194002920)

[5.2 Mapping Pillars to Development Lifecycle 28](#_Toc194002921)

[5.3 Ensuring Deliverables Align with the Six Pillars 29](#_Toc194002922)

[6. Implementation Roadmap 32](#_Toc194002923)

[6.1 Implementation Objectives 32](#_Toc194002924)

[6.2 Phased Rollout Plan 32](#_Toc194002925)

[6.3 Feedback & Continuous Governance 34](#_Toc194002926)

[6.4 Risks & Mitigations 34](#_Toc194002927)

[6.5 Success Measurement (KPIs, Adoption Metrics) 36](#_Toc194002928)

# 1. Executive Summary

## 1.1 Purpose of the document

This document outlines a proposal to improve the development process discipline within the Cloud Platform Engineering (CPE) team. The intent is to introduce a structured, repeatable, and team-aligned development lifecycle that enhances quality, reduces ambiguity, and ensures that platform deliverables are secure, compliant, and production-ready. The proposed process is designed to complement existing automation and GitOps tooling while restoring engineering excellence and delivery predictability across the department.

The document serves both as a strategic guide and a tactical reference for evolving the way platform engineering work is planned, designed, built, reviewed, and released. It is targeted at management stakeholders and engineering contributors alike and forms the basis for phased implementation and organizational alignment.

## 1.2 Summary of the current state

The current development process within the CPE team lacks consistent structure, shared understanding, and engineering discipline. While Agile ceremonies are in place (e.g., standups, grooming, sprint planning), they primarily function as status update meetings rather than collaborative planning or design validation sessions. There is no unified approach to how platform features are designed, reviewed, developed, or validated.

**Key issues include:**

* Inconsistent understanding of what has been implemented, how it works, and what is currently being worked on.
* Work items are often task-oriented rather than outcome-driven, with little to no success criteria or validation mechanisms.
* Design work is typically created and approved in isolation, without peer review or traceable documentation.
* Security and compliance are not integrated early in the development cycle, leading to rework or missed controls.
* Documentation is frequently lacking, leading to knowledge silos and poor onboarding for new team members.

These gaps contribute to inefficiencies, rework, and inconsistent platform quality—and undermine our ability to scale and support a production-grade multi-tenant AWS landing zone environment.

## 1.3 What is being proposed

This document proposes the adoption of a structured, repeatable development process tailored specifically to the needs of the Cloud Platform Engineering (CPE) team. The proposed approach is not a toolset or technology change—it is a **discipline and methodology** for how we design, build, document, validate, and hand off our engineering work.

The development lifecycle follows a simple, consistent flow:

Each phase is designed to:

* Reinforce shared understanding before work begins
* Embed peer review, design validation, and cross-team alignment
* Integrate security, compliance, and operational requirements early
* Ensure all deliverables are testable, documented, and supportable

In addition to the lifecycle, the proposal includes:

* Clear expectations for design documentation, success criteria, and review participation
* Lightweight governance practices to prevent rework, siloed decision-making, or incomplete delivery
* Alignment with strategic engineering principles—such as automation, observability, and secure design
* A phased rollout approach that emphasizes practical adoption over idealized process

By introducing this model, we aim to improve delivery consistency, reduce engineering risk, and enable the CPE team to scale platform capabilities confidently and sustainably.

## 1.4 Expected business and technical benefits

The proposed development lifecycle and supporting practices offer tangible, immediate, and long-term benefits across multiple dimensions of engineering, collaboration, and platform governance.

**Improved Clarity and Shared Understanding**

* Aligns the team on what is being built, why it matters, and what “done” looks like
* Reduces ambiguity during development and limits last-minute changes or scope drift
* Enables team members to pick up or support each other’s work with minimal handover effort

**Greater Engineering Discipline and Delivery Confidence**

* Enforces consistent use of design, success criteria, testing, and documentation
* Prevents the delivery of incomplete, insecure, or unsupported solutions
* Establishes a professional and repeatable standard of work across all feature sets

**Early Security and Compliance Integration**

* Embeds SecOps and Compliance into the earliest lifecycle stages, reducing late-stage review delays
* Improves our ability to generate and track ITSG-33 compliance evidence
* Builds a security-aware engineering culture without creating bottlenecks

**Operational Readiness by Default**

* Ensures that all deliverables are designed and documented for support, not just delivery
* Reduces friction during handoff to Operations, minimizes post-deployment rework
* Makes runbooks, procedures, and platform behaviors clear and available

**Better Collaboration Across Functions**

* Creates clear interaction points with Security, Operations, and Compliance during the lifecycle
* Improves communication, reduces surprises, and builds cross-functional trust

**Alignment with Platform Engineering Principles**

* Reinforces core strategic pillars such as automation, observability, and cost efficiency
* Makes every deliverable traceable, supportable, and aligned to broader platform goals

This approach delivers more than a process—it builds a foundation for sustainable, scalable, and secure engineering outcomes that align with both technical and organizational priorities.

# 2. Current State: Key Challenges

## 2.1 Lack of Shared Understanding

There is currently no consistent or reliable mechanism for ensuring shared understanding across the development team. Team members often have different interpretations of:

* How platform components are implemented
* What functionality currently exists in the environment
* What others are working on, and how it relates to their own tasks

This lack of alignment leads to duplicated effort, inconsistent implementation approaches, and miscommunication about responsibilities. Development work becomes reactive rather than strategic, and decisions made by individuals may conflict with broader architectural patterns or operational expectations.

An additional and critical consequence is the inability for team members to pick up or continue another person’s work in the event of absence or reallocation. Without adequate shared understanding or documentation, task continuity is compromised, often resulting in delays, incomplete work, or rework when the original contributor is unavailable.

While the risk may appear technical in nature, the business impact is real: delays, poor visibility into delivery progress, and an inability to scale engineering outcomes across the organization. This also undermines the trust and effectiveness of our collaboration with Operations, SecOps, and Compliance partners.

## 2.2 Poor Work Definition & Execution Discipline

Much of the current development work is captured in the form of loosely defined tasks rather than well-scoped, outcome-driven deliverables. These work items often lack:

* Clear objectives or success criteria
* Defined inputs, outputs, or dependencies
* Validation requirements or test expectations

This results in a workstream that is difficult to plan, estimate, or track meaningfully. Developers are left to interpret scope based on personal understanding, which can vary widely—especially in a multi-tenant, highly regulated environment like ours.

In many cases, the actual nature of the work evolves during the sprint itself, as team members encounter ambiguity or discover underlying dependencies that were not previously accounted for. This leads to:

* Inconsistent implementation quality
* Last-minute redesign or rework
* Incomplete or unverified delivery of technical components

Furthermore, without consistent discipline around planning, estimation, and progress tracking, it becomes challenging to report accurately on what is being delivered—or to forecast what can be achieved within a given sprint or planning cycle.

From a broader perspective, this erodes confidence in our ability to deliver predictably, which in turn affects how Operations, SecOps, and Compliance teams engage with and depend on engineering output.

## 2.3 Ad Hoc Design and Development

Design and implementation decisions are often made independently by individual developers without consistent peer review, documentation, or cross-team validation. In many cases, designs are:

* Created informally and not documented
* Reviewed only verbally, typically with a single Tech Lead
* Not shared with the broader team for feedback or alignment
* Not converted into actionable requirements or traceable technical objectives

This approach leads to siloed decisions, divergent implementation patterns, and missed opportunities for reuse, standardization, or early detection of issues. Without shared visibility into design intent, downstream team members—including reviewers, testers, and operations staff—are left guessing about why something was built a certain way, or how it is supposed to behave.

Moreover, the lack of structured design validation makes it difficult to assess whether a given deliverable meets its original intent. When things go wrong—or simply behave unexpectedly—there is often no baseline to refer back to, and no defined set of design assumptions to validate against.

This ad hoc approach also hinders onboarding, scalability, and long-term maintainability of the platform. As complexity grows, the absence of formalized design thinking becomes a significant bottleneck to sustainable delivery.

## 2.4 Lack of Security & Compliance Integration

Security and compliance considerations are not consistently integrated into the early phases of development. In the current process, security is often treated as a post-development checkpoint—if it is addressed at all. Common patterns include:

* Developers assuming that their solutions are secure by default
* Minimal or no engagement with the SecOps team during the design phase
* Compliance considerations (e.g., ITSG-33 controls) being addressed retroactively or informally
* No structured security threat modeling or evidence collection aligned to compliance requirements

This introduces avoidable risk, both technical and organizational. Designs may fail to incorporate key security controls or unintentionally violate compliance expectations due to lack of awareness or oversight. By the time these issues are discovered—if at all—they often require rework, introduce delays, or limit the ability to deploy confidently to the Production Tenant.

From a governance perspective, this gap also weakens the credibility of the development process. Security and compliance teams are placed in a reactive role, forced to “check” deliverables they had no input into, which can strain cross-team relationships and reduce trust in engineering outcomes.

Proactive integration of SecOps and Compliance into the design and planning phases is not just a regulatory requirement—it is a foundational aspect of developing production-ready, enterprise-grade platforms.

## 2.5 Documentation & Knowledge Gaps

Documentation across the development process is inconsistent, often informal, and in many cases entirely absent. Critical information—such as design decisions, implementation details, operational behaviors, or known limitations—is frequently stored in individual knowledge, conveyed verbally, or captured in ad hoc chat threads.

This results in:

* A high dependency on tribal knowledge
* Inability for other team members to understand, support, or extend a component they did not build
* Onboarding friction for new team members
* Risk of information loss when contributors leave or shift roles

Without structured and accessible documentation, continuity suffers. Even basic questions such as “What does this module do?”, “Why was it implemented this way?”, or “How is this secured?” become difficult to answer without going directly to the original developer.

The lack of documentation also limits the ability of Operations, SecOps, and Compliance teams to assess, validate, or support deliverables—creating unnecessary bottlenecks and rework when the burden of understanding is shifted to downstream consumers.

Well-documented work is not a luxury—it is a necessary foundation for long-term maintainability, operational readiness, and platform trustworthiness.

## 2.6 Consequences: Risk, Inefficiency, Rework

The combined impact of the issues identified in the preceding sections manifests in a cycle of inefficiency, uncertainty, and rework. Specifically, the lack of shared understanding, poor work definition, ad hoc design, insufficient security integration, and weak documentation practices contribute to:

* Frequent rework and technical corrections due to unclear or misunderstood requirements
* Delayed delivery timelines caused by mid-sprint scope changes, missing design clarity, or unplanned security and compliance fixes
* Low confidence in deliverable completeness and correctness, both within the team and from stakeholders
* Increased dependency on specific individuals who hold critical undocumented knowledge
* Bottlenecks during integration, release, or audit activities, as downstream teams must reverse-engineer intent and context

These outcomes undermine the team’s ability to operate predictably, scale platform capabilities efficiently, or maintain consistent quality standards. They also erode trust and alignment with dependent functions—namely Operations, SecOps, and Compliance—who require confidence in the artifacts delivered by the development team.

If left unaddressed, these consequences will continue to impair the platform’s ability to support a stable, secure, and compliant enterprise-grade Landing Zone.

# 3. Proposed Development Lifecycle

## 3.1 End-to-End Workflow Overview (Design → Validation)

To address the challenges identified in the current development process, we propose adopting a structured, repeatable workflow that emphasizes clarity, alignment, and accountability at every stage of delivery. This workflow forms the foundation for consistent engineering discipline, team-wide understanding, and production-ready outcomes.

The proposed development process is built around a simple, consistent lifecycle

The diagram below visualizes this process in a clear, stepwise format. Each stage transitions logically to the next, alternating between planning, execution, and validation to maximize quality and reduce rework.

Each phase is deliberately placed to:

* Encourage collaboration early
* Validate feasibility and security before full development begins
* Ensure all deliverables are tested, documented, and aligned with operational, security, and compliance expectations

This model balances flexibility with discipline. It accommodates both small, well-understood features and larger, exploratory workstreams by incorporating an optional Proof of Concept (PoC) phase.

Importantly, security and compliance are embedded as early contributors—not late-stage gatekeepers—supporting proactive integration rather than reactive review and reducing final stage SA&A process work.

## 3.2 Lifecycle Stages Explained

## 3.2.1 Design + Team Review

The design phase initiates all development work by clearly defining the problem, outlining the proposed solution, and identifying any known constraints, dependencies, or risks. This phase establishes the **shared understanding** required to ensure that engineering work is aligned, intentional, and testable.

Designs may range from simple architectural overviews for small enhancements to detailed flow diagrams, component interactions, or infrastructure layout documents for more complex deliverables. Regardless of scope, the objective is the same: to make the intent and implementation strategy transparent before development begins.

**Key Design Outputs**

* Problem statement and rationale
* High-level and/or low-level design (as appropriate)
* Key decisions and trade-offs
* Success criteria and validation approach
* Known limitations or open questions
* Security and compliance considerations

Once the initial design is authored, it proceeds to a **Team Review**. This is a collaborative checkpoint involving the full development team and representatives from SecOps (and other stakeholders as needed). The intent is not to “approve” the design in a top-down fashion, but to:

* Validate that the design is well understood by the team
* Identify risks, oversights, or alternative approaches early
* Ensure the proposed approach aligns with platform standards and strategic goals
* Surface any security, compliance, or operational considerations before development begins

Designs are evaluated through a simple **Red/Green consensus model**:

* **Green** indicates the team is aligned, and development can proceed
* **Red** indicates concerns that require further revision, clarification, or validation (e.g., via a Proof of Concept)

This phase reinforces collaborative accountability and avoids isolated decision-making. It also minimizes the risk of costly mid-sprint course corrections by ensuring that what we build is based on what we collectively understand and agree on.

## 3.2.2 Proof of Concept (Optional)

In cases where a proposed design contains significant unknowns, novel integrations, or unproven approaches, a **Proof of Concept (PoC)** phase may be introduced to reduce uncertainty before committing to full-scale development.

The PoC is a **lightweight, time-boxed technical exercise** used to:

* Validate core assumptions or hypotheses within the design
* Explore feasibility of integration points, APIs, or platform behaviors
* Identify technical blockers or gaps early
* Generate focused feedback from stakeholders such as SecOps or Operations

The goal is not to produce production-ready code, but rather to **confirm that the proposed direction is viable**—or, if not, to pivot quickly and with minimal sunk cost. A successful PoC may take the form of a spike module, a scripted workflow, a sandbox deployment, or even a partial configuration applied in a non-critical environment.

**When to Use a PoC**

* New platform capabilities or patterns are being introduced
* There is uncertainty around tool compatibility or integration
* There are strong security, operational, or compliance concerns
* The team explicitly flagged open risks during the design review

The outcome of the PoC is not just technical—it provides **input into team decision-making**. It informs the POC Review, where the approach is validated and either greenlit for development or revised based on findings.

**POC Review**

Following the completion of a Proof of Concept, a formal POC Review is conducted to assess the findings, validate assumptions, and determine whether the proposed approach is ready for full development. This step acts as a quality gate, ensuring that decisions made during the exploratory phase are sound, scalable, and aligned with broader platform and organizational requirements.

The review includes participation from:

* Development Team (presenting the findings and recommending a path forward)
* SecOps (to assess any security risks or concerns identified)
* Operations (to provide feedback on observability, deployment impact, or operational supportability)

POC Review Objectives

* Confirm that the original uncertainties have been addressed
* Evaluate the success criteria defined during design
* Identify any new technical, security, or compliance risks
* Determine if the implementation plan needs revision before development proceeds

The review concludes with a clear decision:

* Proceed with implementation as designed
* Revise the design or implementation strategy based on findings
* Reject the current approach and return to design for an alternate solution

This step reinforces discipline and prevents premature commitment to flawed or unproven approaches, protecting both sprint capacity and platform integrity.

By intentionally separating exploration from execution, this phase protects sprint velocity, reduces risk, and supports smarter architectural decisions.

## 3.2.3 Development & Testing

Following an approved design or validated Proof of Concept, work progresses into the **development and testing phase**. This is the execution stage in which the solution is implemented in accordance with the agreed design, and verified for correctness, quality, and maintainability.

This phase includes:

* Implementation of the planned solution using appropriate tools and standards
* Incremental testing throughout the build process
* Peer feedback and collaborative problem-solving
* Ongoing validation against the original success criteria

Because the nature of development work can vary—from infrastructure as code to platform automation, security hardening, or documentation tooling—the exact testing and validation mechanisms may differ. However, the following expectations apply to all deliverables:

* **The solution must be verifiable**: There must be a clear way to confirm that the work performs as intended.
* **The solution must be reviewable**; Team members should be able to inspect and understand the implementation through version-controlled artifacts and human-readable design or test notes.
* **The solution must be maintainable**: Naming, structure, and documentation should support future modifications and reduce the burden on Operations, Security, or other support teams.

Where applicable, test cases, configuration validation, automated checks, or peer walkthroughs may be used to confirm correctness and completeness. This discipline ensures that development work results not just in code, but in **reliable, supportable platform capabilities** that meet the team’s standards and align with broader delivery expectations.

## 3.2.4 Deployment, Documentation, Security & Compliance

Once development and testing are complete, the next phase focuses on preparing the deliverable for production readiness. This stage ensures that the solution is not only functional, but also deployable, supportable, and compliant with operational, security, and governance expectations.

**Deployment**

Deployment activities may vary depending on the nature of the solution and its criticality. This includes:

* Release scheduling and coordination
* Application of GitOps-based deployment mechanisms where applicable
* Use of deployment environments for staging, integration, or validation
* Ensuring rollback or recovery plans are understood and documented

This phase must include verification that the deployment approach is sound and does not create operational risk for the Production Tenant.

**Documentation**

To support downstream teams and future maintenance, every deliverable must include **fit-for-purpose documentation**. This may include:

* Operational documentation for support teams (e.g., runbooks, standard operating procedures)
* Technical documentation for developers and engineers (e.g., architectural notes, configuration references)
* User-facing guides or onboarding materials, when applicable

Documentation must be accessible, stored in version control or a recognized knowledge base, and updated as the work evolves.

**Security & Compliance**

In parallel, deliverables must be reviewed for adherence to internal security and compliance requirements. This includes:

* Validation against known security practices and organizational controls
* Early integration of SecOps and compliance expectations (not post-delivery review)
* Preparation of documentation or artifacts required for ITSG-33 evidence, where applicable

Rather than treat security and compliance as final sign-offs, this phase ensures that they are embedded in the development process and verified as part of normal delivery operations.

## 3.2.5 Final Validation & Success Metrics

The final phase of the development lifecycle is dedicated to validating the completeness, effectiveness, and readiness of the delivered work. This is not a technical test phase, but rather a structured close-out process that ensures the solution meets its original intent and is ready for operational ownership.

This phase includes three key activities:

**Final Checks**

A review of the completed work against its original design, scope, and success criteria. The goal is to confirm:

* All intended outcomes have been achieved
* The solution behaves as expected under real conditions
* Any deviations from the original plan are documented and justified

This step may involve light operational testing, team walkthroughs, or retrospective analysis of issues encountered and resolved during development.

**Post-Implementation Review**

A short-form internal review used to gather observations from the development team, stakeholders, and reviewers. This is not a retrospective in the Agile sense, but a focused checkpoint that asks:

* Did the work deliver value as intended?
* Were there avoidable blockers or delays?
* Are there any follow-up tasks, enhancements, or risks to capture?

This feedback is used to improve future planning and reinforce continuous improvement.

**Hand-Over for Operationalization**

The final outcome is a formal or informal hand-over of the deliverable to the relevant owning or consuming teams. This includes:

* Operational readiness confirmations
* Documentation and support links
* Clarification of ownership, support boundaries, and future responsibilities

This stage closes the loop on delivery and enables the team to move forward with confidence that the work is not only “done,” **but ready to be used, supported, and built upon**.

## 3.3 Time Planning, Progress Tracking, and Reporting

To support predictability and transparency across the delivery lifecycle, the proposed development process incorporates a lightweight but structured approach to time planning, progress tracking, and reporting.

This is not intended to add administrative burden but to ensure that work can be forecasted, measured, and communicated effectively—both within the team and to stakeholders.

**Time Planning**

At the outset of each development effort—whether a small enhancement or a multi-sprint initiative—basic time expectations should be defined. This includes:

* Estimated effort (e.g., hours or days) for each stage of the lifecycle
* Anticipated timeline or sprint alignment
* Identification of risks, dependencies, or blockers that may affect timing

Estimates are not rigid commitments; they are intended to support sprint planning, resource allocation, and delivery forecasting.

**Progress Tracking**

Progress is tracked **not by ticket closure alone**, but by tangible movement through the lifecycle. Each stage provides a natural checkpoint for verifying that work is moving forward and that quality standards are being met.

Checkpoints may include:

* Design review completed and greenlit
* PoC delivered and reviewed
* Code implementation completed and verified
* Documentation submitted
* Deployment completed and validated
* Handover finalized

It’s also important to recognize that some development work—particularly larger platform capabilities or cross-cutting features—may span multiple sprints. In these cases, tracking progress by lifecycle stage offers a more accurate and meaningful representation of delivery maturity than task-level status alone. It allows the team to communicate where the work truly is, not just how much effort has been expended.

**Reporting**

Work status should be reported in terms that reflect **delivery confidence**:

* What phase is the work in?
* Are success criteria at risk?
* Is further input needed from Operations, SecOps, or other stakeholders?

Sprint updates, demos, and planning meetings should reinforce this model—focusing not just on what was “done,” but whether the work met its intended goal and is ready for operational use.

This reinforces accountability, supports better planning, and helps all teams maintain shared visibility into what is being delivered, how, and when.

# 4. Engineering Principles and Governance

This section defines the guardrails and shared expectations that support the development lifecycle outlined in Section 3. While the lifecycle describes what we do at each stage of delivery, this section focuses on how we work together to ensure consistency, quality, and alignment.

These principles are not intended to add bureaucracy—they exist to ensure that our development work remains verifiable, maintainable, and ready for operational ownership. They also provide clarity for downstream teams and leadership, reinforcing transparency and accountability across the delivery process.

## 4.1 Definition of Ready / Done

To ensure that development work begins with a clear understanding and ends with shared confidence in completion, we establish two complementary standards: the **Definition of Ready (DoR)** and the **Definition of Done (DoD)**. These definitions are not just procedural—they are the foundation for predictability, accountability, and quality across the team.

**Definition of Ready (DoR)**

Work is considered “ready” to enter development when it meets the following baseline criteria:

* The problem statement and scope are clearly defined
* A design document (appropriate to the size and complexity of the work) is complete
* Success criteria are articulated and agreed upon
* Known risks, constraints, and dependencies are documented
* A Red/Green Team Review has been completed and approved
* Input has been received from SecOps and Operations, if relevant

This ensures that development starts only when the team is aligned and the work is feasible, understood, and scoped appropriately.

**Definition of Done (DoD)**

Work is considered “done” when the following outcomes are in place:

* The solution is implemented and validated against the success criteria
* Required documentation (technical, operational, or user-facing) is complete
* Security and compliance reviews have been completed
* The work has been handed off or made ready for operationalization
* Any relevant metrics, outputs, or support requirements have been captured
* The team has agreed that the work is complete and requires no further changes or clarification

Applying a consistent Definition of Done across all types of development work reinforces delivery integrity and minimizes the risk of rework, missed expectations, or operational surprise.

## 4.2 Peer Review Expectations

Peer review is a core practice in maintaining quality, fostering shared understanding, and reducing delivery risk. It is not a formality—it is a collaborative discipline that improves outcomes through constructive feedback, early detection of issues, and reinforcement of common standards.

All development work, regardless of type or size, is expected to undergo peer review at appropriate stages. This includes not only code, but also design artifacts, documentation, and infrastructure definitions.

**What Should Be Reviewed**

* **Design Documents**: Reviewed during team design review, with feedback captured and revisions made as needed
* **Implementation Artifacts**: Code, templates, pipelines, or scripts must be reviewed before merge or deployment
* **Documentation**: Operational, user-facing, or technical documentation must be reviewed for completeness and clarity

**Review Expectations**

* Reviews are expected to be timely, respectful, and focused on the deliverable, not the individual
* Reviewers are responsible for validating alignment to the design, clarity of implementation, and maintainability
* Contributors are expected to respond constructively to feedback, explain design decisions when challenged, and revise work where appropriate
* Where security, compliance, or operational impact exists, relevant stakeholders (e.g., SecOps, Operations) should be engaged as part of the review cycle

Peer review is both a quality control mechanism and a knowledge-sharing practice. It ensures that work is never isolated, and that technical debt, security concerns, or hidden complexity are surfaced early—when they are cheapest to resolve.

## 4.3 Use of Design Documents and Success Criteria

Design documents and clearly stated success criteria are foundational elements of disciplined, intentional engineering. They ensure that the work is guided by a shared understanding of what needs to be built—and what success looks like when it’s done.

This section establishes minimum expectations for when and how these artifacts are used.

**Design Documents**

Every substantial piece of work—whether it relates to infrastructure, automation, compliance tooling, or security controls—must begin with a documented design. The level of detail should match the complexity and impact of the work, but all designs must address:

* What problem is being solved and why
* How the solution will be implemented
* What risks, constraints, and trade-offs are being considered
* How the work aligns with platform standards or broader architectural patterns
* Any specific concerns for SecOps, Compliance, or Operations

Designs do not need to be overly formal, but they must be concrete, reviewable, and version-controlled.

**Success Criteria**

Success criteria define the intended outcome in objective, measurable terms. They serve as the baseline for validation and are used throughout the lifecycle to:

* Determine whether the work can begin (Definition of Ready)
* Guide development and testing activities
* Confirm completion (Definition of Done)
* Communicate value and progress during reporting

Success criteria should be defined early (ideally during design) and reviewed during the Red/Green team review. They should describe not just what will be built, but how its success will be verified—whether through functional output, performance metrics, or compliance alignment.

## 4.4 Documenting for Reusability and Onboarding

Documentation is not a closing activity—it is a parallel discipline that supports clarity, continuity, and reusability across the entire development lifecycle. It ensures that what is built can be understood, supported, extended, and reused by others—not just the original contributor.

This principle is especially important in a platform engineering context, where components are reused across environments, owned by multiple teams, and subject to compliance and operational scrutiny.

**What Should Be Documented**

At a minimum, the following types of documentation should accompany any completed deliverable:

* **Technical Documentation**: Architecture decisions, component behaviors, dependencies, limitations, and integration details
* **Operational Documentation**: Deployment instructions, runbooks, rollback/recovery steps, configuration options, monitoring hooks
* **Support and Onboarding Documentation**: Usage guidance for Operations, Security, and other consumers of the platform capability

The documentation depth should reflect the complexity and criticality of the work. A minor configuration change may require a brief update to an existing doc, while a new automation module may require a full onboarding guide and support reference.

**Where Documentation Lives**

Documentation must be:

* Version-controlled alongside the codebase or IaC when it is developer-focused or tightly coupled to the code itself
* Published to the central document repository (e.g., SharePoint) when it is intended for operational, security, compliance, or cross-team consumption
* Accessible to all relevant stakeholders, including Operations, SecOps, Compliance, and Management
* Kept current throughout the development process and finalized during the closeout phase

This dual-path approach ensures that documentation is both **maintainable by the team** and **discoverable by the organization**—bridging the gap between technical authorship and broad usability.

**Why It Matters**

Good documentation accelerates onboarding, enables faster troubleshooting, reduces rework, and increases confidence across teams. It also ensures that knowledge is preserved and distributed—not held in silos or lost during transitions.

As a team principle: **write documentation as though the next person to read it will need to support it without your help.**

## 4.5 Security & Compliance Checkpoints

Security and compliance are not optional concerns to be addressed after development—they are foundational aspects of platform engineering, particularly within a regulated and multi-tenant environment. This section defines when and how **SecOps** and **Compliance** must be integrated into the development lifecycle to ensure all work is secure, reviewable, and ready for audit or operational acceptance.

**Early Engagement is Expected**

Security and compliance input must be included during the design phase, not left until deployment or final review. This allows:

* Proactive threat modeling or identification of control gaps
* Alignment with existing security standards or frameworks (e.g., ITSG-33)
* Timely clarification of ownership, responsibilities, or evidence needs

SecOps should be included in the Team Review, and both SecOps and Compliance should be consulted during the POC Review and again during the Deployment & Documentation phase, depending on the scope and risk profile of the work.

**Roles and Responsibilities**

* **Development Team**: Responsible for initiating security and compliance engagement, integrating feedback, and collecting necessary artifacts or evidence
* **SecOps**: Reviews technical aspects of the solution to identify vulnerabilities, assess risk posture, and provide guidance on secure implementation. This includes input on identity management, encryption, logging, and network exposure
* **Compliance Team (SMG)**: Ensures that the solution aligns with the applicable controls under ITSG-33 and that sufficient evidence is generated and stored for audit or assessment activities

**Evidence and Traceability**

Where required, development efforts must produce or update artifacts that contribute to system security assurance and audit readiness. These may include:

* Design documents referencing security considerations and decisions
* Implementation notes documenting how key security behaviors are achieved
* Compliance traceability matrices or security posture summaries
* Evidence artifacts for traceability to specific ITSG-33 controls

Proactively addressing security and compliance in this structured way reduces the risk of rework, accelerates acceptance processes, and ensures the platform remains trustworthy and auditable as it evolves.

## 4.6 Exceptions & Flexibility

While the development lifecycle and governance model outlined in this document provides structure and consistency, it is not intended to be inflexible or overly prescriptive. We recognize that not all work requires the same depth of process, and that some deliverables—particularly small, low-risk changes—may not warrant every stage of the lifecycle.

This section establishes a clear principle:

**Any lifecycle stage may be streamlined or skipped—but only through conscious agreement, not default behavior.**

**When Exceptions May Be Appropriate**

* Small, isolated changes with no external impact (e.g., variable renaming, internal comment updates)
* Low-risk internal optimizations with no security, operational, or compliance implications
* Emergency or hotfix work that requires rapid delivery and limited procedural overhead (with retroactive validation as needed)

**How Exceptions Are Handled**

* The team must agree that a reduced lifecycle is appropriate
* The rationale for skipping a stage should be noted (e.g., in ticket notes, pull request comments, or sprint documentation)
* Security and Compliance teams must still be informed if their areas are impacted—even in streamlined cases
* Even when formal stages are skipped, the **outcomes of readiness, verification, and communication must still be met**

This flexible approach maintains quality and accountability without introducing unnecessary overhead. It allows the team to adapt the process to the nature of the work—while ensuring that guardrails are respected, and that critical stages are not silently bypassed.

## 4.7 Continuous Improvement

Process discipline is not static—it must evolve as the team grows, the platform matures, and the nature of our work changes. To ensure our development lifecycle remains effective, relevant, and practical, we commit to treating the process itself as a living system—subject to feedback, reflection, and refinement.

**Team Feedback Loops**

Regular opportunities for feedback should be embedded in the team’s operating rhythm, including:

* **Sprint Retrospectives**: Identify pain points in the lifecycle, tooling, or handoffs that require adjustment
* **Peer Reviews**: Surface trends in documentation quality, readiness, or testing that can inform process tuning
* **Architecture, Design & POC Reviews**: Capture recurring friction, misunderstandings, or success patterns to inform future design guidance

**Adapting the Process**

The team is encouraged to suggest improvements and raise challenges when:

* A lifecycle stage is routinely adding overhead without value
* New tools or practices offer a more efficient path to the same outcome
* The nature of the work has changed and requires a different approach

Process updates should be evaluated collaboratively and agreed upon as a team. Changes may be piloted informally before being adopted more broadly.

**Institutional Learning**

Where possible, process improvements and lessons learned should be captured in documentation, onboarding materials, or internal playbooks. This ensures that institutional knowledge is retained and shared across teams, not just individuals.

Continuous improvement reinforces a culture of self-accountability, adaptability, and excellence—making the process not just sustainable, but resilient.

# 5. Strategic Alignment with Cloud Engineering Principles

## 5.1 Pillar Overview

The proposed development lifecycle is designed to do more than structure engineering work—it also reinforces the strategic principles that underpin modern platform engineering. These principles are summarized in the Six Pillars of Cloud Engineering, which guide the way we build, deliver, and support scalable, secure, and resilient platforms.

The Six Pillars are:

* **Automation:** Reducing manual effort, human error, and deployment drift through infrastructure as code, pipelines, and repeatable workflows.
* **Observability & Traceability:** Ensuring system behaviors, design decisions, and delivery outcomes are transparent, auditable, and well-documented.
* **Security:** Embedding security posture and risk reduction early in the lifecycle, not just at release.
* **Site Reliability Engineering (SRE):** Building stable, supportable, and scalable platforms with a focus on operational excellence and incident prevention.
* **Cloud-native Services:** Leveraging platform-aligned, cloud-native capabilities that reduce overhead and support agility without compromising control.
* **FinOps:** Promoting cost awareness, resource efficiency, and financial accountability as part of technical decision-making.

These pillars are not aspirational—they reflect practical needs and operational realities in our environment. The development lifecycle introduced in this document was intentionally structured to reinforce these principles, ensuring that platform delivery is not only efficient, but strategic, sustainable, and aligned with the enterprise mission.

## 5.2 Mapping Pillars to Development Lifecycle

The development process defined in this document is directly aligned to the Six Pillars of Cloud Engineering. Each stage of the lifecycle reinforces one or more of these principles, ensuring that platform delivery is not only methodical—but also strategic.

**Automation:**

* Use of GitOps workflows for deployment, documentation versioning, and design artifacts supports reproducibility and consistency
* Defined lifecycle checkpoints reduce reliance on manual oversight or unstructured validation
* Peer reviews, lifecycle tracking, and handoff readiness contribute to smoother, automated platform integration

**Observability & Traceability**

* Design documents and success criteria ensure work is traceable from intent to outcome
* Documentation and peer reviews make design and implementation decisions visible to all stakeholders
* Lifecycle-based reporting replaces ambiguous ticket status with outcome-based progress

**Security**

* Security engagement during Design and POC phases promotes proactive threat mitigation
* SecOps participation in Team Review and POC Review ensures risk awareness before development begins
* Lifecycle deliverables produce artifacts that support ITSG-33 security posture and evidence gathering

**Site Reliability Engineering (SRE)**

* Testing and validation requirements reduce the likelihood of operational issues or downstream instability
* Documentation and runbook expectations ensure that new capabilities can be monitored, supported, and recovered predictably
* Final validation and closeout processes improve the integrity of operational handoffs

**Cloud-native Services**

* The Design + Review process encourages thoughtful use of native cloud capabilities aligned with platform direction
* POC phases allow new technologies or integrations to be tested without exposing production environments to risk
* Platform-aligned standards are reinforced through collaborative review and design transparency

**FinOps**

* Design phase tradeoff discussions encourage teams to consider cost implications early
* Success criteria can include efficiency or usage metrics tied to cloud resources
* Reusability and standardization reduce redundant infrastructure and duplicate service patterns

This alignment ensures that every piece of development work, no matter how small, contributes to a broader set of platform engineering outcomes that scale beyond individual tasks—and support a secure, observable, cost-aware, and reliable Landing Zone.

## 5.3 Ensuring Deliverables Align with the Six Pillars

In addition to aligning the overall development process to the Six Pillars of Cloud Engineering, it is equally important that **each individual deliverable**—regardless of size or scope—demonstrates alignment with these principles in its design, implementation, and documentation.

This is not a theoretical exercise. These principles provide the foundation for building secure, scalable, maintainable, and cost-aware platform components that meet the needs of both internal teams and external consumers.

Deliverables are expected to demonstrate pillar alignment through thoughtful engineering, clearly recorded decisions, and reviewable outputs.

**Automation**

* Deliverables should be implemented in a way that supports repeatable deployment through automation
* Avoid manual configuration or state drift; leverage GitOps or pipeline-based execution wherever possible

**Observability & Traceability**

* Include basic observability hooks (e.g., logging, metrics, traceable workflows) where applicable
* Ensure documentation includes intent, behavior, and outcomes—so future readers can understand the “why” as well as the “what”
* All changes must be traceable to a defined success criteria or engineering objective

**Security**

* Address security considerations directly in the design
* Avoid default-permit configurations, exposed endpoints, or hardcoded secrets
* Engage SecOps during design and development, especially for components with elevated privilege, external access, or sensitive data

**Site Reliability Engineering (SRE)**

* Validate that the work can be operated, monitored, and recovered with minimal intervention
* Provide runbooks or support notes when applicable
* Minimize complexity and brittle dependencies that increase operational risk

**Cloud-native Services**

* Leverage native cloud or platform-aligned services where they reduce maintenance burden and improve agility
* Avoid introducing unnecessary tools or frameworks when a supported service already exists within the platform ecosystem

**FinOps**

* Consider cost implications of the design and implementation (e.g., data transfer, idle compute, scale-out behaviors)
* Use sizing and configuration defaults that reflect expected use—not maximum capacity
* Consolidate resources, when possible, to reduce redundancy and overhead

This alignment should be considered during **design planning, peer review**, and **final validation**. Reviewers are encouraged to ask:

“Does this work support the Six Pillars, or does it create risk, complexity, or cost that undermines them?”

By embedding these principles into everyday engineering practice—not just documentation—we ensure that everything we build strengthens the platform’s foundation.

# 6. Implementation Roadmap

## 6.1 Implementation Objectives

The intent of this implementation is to transition from inconsistent, ad hoc engineering practices to a **structured and sustainable development process** that improves delivery quality, clarity, and accountability across the Cloud Platform Engineering team.

This is not about introducing formality for its own sake. The goal is to:

* **Improve clarity and predictability** in how development work is scoped, delivered, and validated
* **Strengthen cross-functional alignment** with Security, Compliance, and Operations
* **Reinforce team-wide visibility** into what is being built, how it is built, and when it is ready for use
* **Reduce delivery risk and technical debt** by embedding design, review, and documentation into normal workflows
* **Establish best practices and shared expectations** to support long-term platform scalability

Importantly, this implementation will build on existing tools, platforms, and workflows—such as GitOps pipelines, ArgoCD, and ADO Repos. There is no intent to disrupt these foundations. Instead, the focus is on improving **how we work within them**.

The implementation approach is designed to be **incremental and team-led**, with early emphasis on the highest-impact changes (such as design discipline, success criteria, and handoff readiness). Feedback will be gathered continuously and used to adjust the model as adoption proceeds.

## 6.2 Phased Rollout Plan

To support adoption without introducing disruption or fatigue, the proposed development process will be rolled out in three practical phases. Each phase focuses on foundational practices first, then builds toward maturity and strategic alignment.

This phased approach allows the team to build confidence, observe early benefits, and shape the process collaboratively based on lived experience.

**Phase 1: Foundation – Establishing Core Discipline**

Focus: **Start small, build consistency**

* Begin using structured **design documentation** for all new work (Level of detail matched to scope and complexity)
* Introduce the **Red/Green Team Review** as a mandatory checkpoint before development
* Define and capture success criteria for all development work
* Track work progress by **lifecycle stage**, not just ticket status
* Apply peer review to **both design and code**, not just implementation artifacts

This phase sets the cultural tone: design comes first, success is defined, and review is collaborative.

**Phase 2: Integration – Embedding Process into Broader Practice**

Focus: **Improve quality and handoff readiness**

* Use the **Proof of Concept** and **POC Review** model for exploratory or high-risk work
* Expand **documentation expectations** and link final artifacts to SharePoint and Git as appropriate
* Integrate SecOps and Compliance earlier in the lifecycle and reviews
* Begin surfacing lifecycle alignment in sprint planning and standups

This phase strengthens delivery discipline and brings Security, Operations, and Compliance into alignment with development cadence.

**Phase 3: Maturity – Reinforcing Strategic Engineering**

Focus: **Drive alignment to platform and business goals**

* Incorporate **pillar alignment** checks into design reviews and code validation
* Use success criteria and lifecycle checkpoints to inform **progress reporting**
* Apply findings from sprint retrospectives to refine process stages and eliminate friction
* Codify updated guidance and examples into team documentation or onboarding materials

This phase completes the transition from ad hoc engineering to a fully traceable, accountable, and reusable development model.

## 6.3 Feedback & Continuous Governance

The successful adoption of this development process depends on regular feedback and continuous refinement. Rather than treating this model as a static policy, the team will treat it as a living framework, adjusting it over time based on experience, outcomes, and stakeholder needs.

**Feedback Channels**

Feedback should be collected through regular team activities, including:

* **Sprint Retrospectives**: Identify where the process added value, caused friction, or could be improved
* **Design and Review Sessions**: Capture repeat blockers, misunderstandings, or process gaps
* **Peer Feedback**: Encourage team members to surface concerns or suggestions informally between ceremonies

**Process Governance**

* **Initial Adoption Review**: After the first 2–3 sprints using the new process, the team should conduct a focused review to assess early effectiveness and needed adjustments.
* **Quarterly Checkpoints**: Set aside time once per quarter (or release cycle) to review the process and confirm whether any updates to documentation, expectations, or review steps are needed.
* **Shared Ownership**: All team members, not just leadership, are responsible for improving the process and holding one another accountable to it.

This governance model ensures that the process remains practical, scalable, and relevant—and reinforces a team culture of continuous improvement.

## 6.4 Risks & Mitigations

Any process change carries the potential for friction, misunderstanding, or unintended side effects. While the proposed development lifecycle is intentionally lightweight and aligned to existing workflows, it is important to acknowledge the risks associated with its adoption—and how we plan to address them.

**Risk: Resistance to Change**

Team members may be hesitant to adopt new documentation, design, or review practices.

**Mitigation:**

Start with clear, simple expectations. Emphasize that structure is being added to support the team, not burden it. Reinforce benefits through early wins and peer support, not policy enforcement.

**Risk: Process Overload**

The lifecycle may be seen as too heavy for smaller or fast-moving work items.

**Mitigation:**

Apply the Exceptions & Flexibility guidance defined in Section 4.6. Allow teams to tailor the process by context, as long as accountability and traceability are preserved.

**Risk: Misalignment with Agile Ceremonies**

Teams may struggle to integrate lifecycle stages into sprint planning, standups, or reviews.

**Mitigation:**

Map lifecycle stages directly to Agile practices (e.g., designs prepared before sprint start, lifecycle state used in status updates). Treat the lifecycle as a complement to Agile, not a replacement.

**Risk: Incomplete Adoption Across Teams**

Some developers or reviewers may not consistently apply the process, creating delivery gaps.

**Mitigation:**

Use peer review and lifecycle-aligned reporting to surface gaps naturally. Encourage team members to hold each other accountable constructively. Reinforce expectations through retrospectives and design reviews.

**Risk: Documentation Fatigue**

Increased emphasis on documentation may lead to delays or shallow compliance.

**Mitigation:**

Provide guidance and templates that scale with complexity. Emphasize documentation value in terms of operational handoff, security review, and reusability—not as a checkbox.

This section acknowledges that adoption is a process—not an instant shift—and reinforces that the team is empowered to shape it through feedback, reflection, and shared ownership.

## 6.5 Success Measurement (KPIs, Adoption Metrics)

To ensure the development lifecycle is delivering its intended benefits—and to support continuous improvement—success must be measured not just anecdotally, but through observable outcomes. These measurements provide visibility for the team, stakeholders, and leadership into how well the process is working and where further refinement may be needed.

**Adoption Metrics**

Track whether the process is being followed and where uptake may be inconsistent:

* Percentage of work items with a defined design and success criteria
* Percentage of features reviewed by both peers and SecOps
* Frequency of POC usage for exploratory or high-risk features
* Volume of documentation linked to SharePoint and Git

These metrics help identify adoption gaps early and guide support or coaching needs.

**Performance Indicators (Team-Level)**

Assess how the new process improves team delivery over time:

* Reduction in post-sprint rework or last-minute redesigns
* Improvement in delivery predictability (commit vs. complete)
* Cycle time from design to validated deliverable
* Feedback from Operations or Compliance on deliverable quality and readiness

These indicators reflect the lifecycle’s impact on engineering discipline, handoff quality, and stakeholder trust.

**Strategic Alignment Indicators**

Evaluate whether engineering outputs support broader platform goals:

* Evidence artifacts generated to support ITSG-33 compliance reviews
* Proportion of deliverables demonstrating alignment with one or more of the Six Pillars
* Team self-assessment of confidence in platform readiness, reusability, and documentation

Success is not defined by 100% adherence to process—but by measurable improvements in platform quality, collaboration, and confidence.