NYCPS TMS: Hyper-Detailed Prescriptive DevSecOps Strategy

I. Introduction: Philosophy, Goals & Scope

This document mandates the comprehensive, highly detailed, end-to-end DevSecOps strategy and implementation plan for the NYCPS Transportation Management System (TMS) project (RFP R1804). This prescriptive approach integrates Development, Security, Quality Assurance (QA), Site Reliability Engineering (SRE), and Operations into a unified, automated, and collaborative workflow. It leverages **GitLab** as the central DevOps platform and targets deployment onto **AWS GovCloud (US)** infrastructure provisioned via **Terraform**.

The overarching methodology is **Agile (Scrum)**, executed in short iterations (Sprints) to deliver value incrementally and adapt to feedback. Security and Quality are not phases but continuous activities embedded throughout the entire lifecycle ("Shift Left").

This strategy is designed for a **globally distributed team of 50-60 engineers** working across diverse technology stacks (Frontend: React/React Native/TypeScript; Backend: Python/FastAPI, Node.js/Express, Java/Spring Boot; Infrastructure: Terraform/AWS; Databases: PostgreSQL/PostGIS, DynamoDB; Messaging: Kinesis/MSK, SQS/SNS).

Mandatory Goals:

1. **Elite Developer Experience & DORA Metrics:** Provide a seamless, efficient local development setup, rapid CI feedback (<10-15 mins typical), clear processes, and minimized friction. Target elite DORA performance: Daily+ Deployment Frequency, <1 Day Lead Time for Changes, <15% Change Failure Rate, <1 Hour Time to Restore Service.
2. **Ultra-Fast, Secure, Reliable CI/CD Pipeline:** Implement highly optimized GitLab CI/CD pipelines with extensive caching, parallelization, and robust error handling. Pipelines must enforce all quality and security gates automatically.
3. **Impenetrable Automated Quality & Security Gating:** Mandate automated checks at every stage: pre-commit, CI (MRs, main branches), post-deployment. Implement zero tolerance for critical/high security vulnerabilities and regressions in core functionality. Developer-owned testing is foundational.

4. **Hyper-Automation:** Automate infrastructure provisioning (IaC), configuration management, builds, all test levels (Unit, Integration, API, E2E, Performance, Security, Accessibility), deployments (Blue/Green, Canary), monitoring setup, alerting, and rollback triggers.

II. Visualizing the End-to-End DevSecOps Flow (Textual Representation)

The following describes the journey of a typical code change (new feature, bug fix, enhancement) from a developer's machine through to production, highlighting the automated gates and feedback loops.

```
1. [Developer] -> Checkout `develop`, Create `feature/TMS-XXX` branch.
2. [Developer] -> Code Feature/Fix + Write Unit/Integration Tests (TDD).
3. [Developer] -> Run tests locally + Format/Lint Code.
4. [Developer] -> `git commit`
   [Pre-Commit Hook: Lint/Format/Secrets Scan -> PASS]
5. [Developer] -> `git push` to feature branch.
6. [Developer] -> Create GitLab Merge Request (MR) -> `develop`.
```

7. [GitLab CI] -> Trigger MR Pipeline: Build **->** Unit Test **->** SAST **->** SCA **->** Coverage Check **[Quality Gate: All Checks PASS]**

8. [Reviewer(s)] -> Peer Code Review (Focus: Logic, Security, Testing, Standards). **[Quality Gate: Approval(s) Required]**

9. [Developer/Integrator] -> Merge MR (Squash) -> `develop`.

10. [GitLab CI] -> Trigger `develop` Branch Pipeline: Build **->** Test **->** Scan **->** Package Artifacts (Docker Image/Lambda Zip). **[Quality Gate: All Checks PASS]**

11. [GitLab CD] -> Auto-Deploy Artifact -> **[DEV Environment]**.

12. [GitLab CI] -> Auto-Run Post-DEV Deploy Tests (API Tests, Basic Integration Tests). **[Quality Gate: PASS]** -> Notify Dev Team.

13. [GitLab CD] -> Manual Trigger/Schedule -> Deploy Artifact -> **[QA Environment]**.

14. [GitLab CI/QA Team] -> Run Post-QA Deploy Tests (Full Integration, E2E Tests, DAST Scan, Accessibility Scan). QA performs Exploratory Testing. **[Quality Gate: All Automated Tests PASS, No Blocking Manual Bugs]** -> Notify QA/Dev Team.

15. [Release Manager/Lead] -> Create `release/vX.Y.Z` branch from `develop`.

16. [GitLab CD] -> Deploy `release/` branch -> **[Staging/UAT Environment]**.

17. [NYCPS Stakeholders/QA] -> Conduct UAT.

18. [GitLab CI/Perf Team] -> Deploy `release/` branch -> **[Perf Test Environment]** -> Run Automated Load/Performance Tests.

19. [Security Team/3rd Party] -> Conduct Penetration Testing against Staging. **[Quality Gates: UAT Sign-off, Perf NFRs Met, No Critical/High PenTest Findings]**

20. [Release Manager] -> Merge `release/` -> `main`, Tag `vX.Y.Z`, Merge `release/` -> `develop`.

21. [GitLab CD] -> Trigger Prod Deploy from Tag -> **[Manual Approval Gate: CCB/Release Manager]** -> Execute Blue/Green or Canary Deploy -> **[PROD Environment]**.

22. [GitLab CI/SRE] -> Run Post-Prod Smoke Tests. Intense Monitoring Activated. **[Quality Gate: Smoke Tests PASS, Key Metrics Stable]** -> Complete Traffic Shift (if B/G or Canary).

23. [SRE/Ops/Dev] -> Ongoing Monitoring, Alerting, Incident Response (using CloudWatch, APM, PagerDuty).

```
24. [All Teams] -> Feedback Loop (Monitoring Data, Incident RCAs, User
Feedback, Retrospectives) -> Input to Backlog (Step 1).
```

III. Detailed Lifecycle Phase Implementation

This section provides exhaustive detail on the activities, tools, responsibilities, quality gates, and implementation steps for each phase of the DevSecOps lifecycle.

A. Phase: Planning & Requirements Definition

1. Activity: Project Initiation & Governance Setup

Implementation Steps:

1. Schedule and conduct the official project kick-off meeting involving all key vendor and NYCPS stakeholders (Exec

Sponsors, PMs, Tech Leads, Security Liaisons, OPT SMEs).

2. Document and distribute kick-off meeting minutes, including agreed-upon vision, high-level goals, key contacts, and communication plan (meeting cadence, status reporting format/frequency, escalation paths).
3. Formally define and document the Governance Structure (Steering Committee charter, Technical Review Board scope, Change Control Board process aligning with NYCPS standards). Obtain sign-off on the governance plan.
4. Create shared project calendars and distribution lists.
5. Review the high-level 12-month roadmap; identify immediate dependencies and high-level risks. Populate initial Risk Register in Jira/Confluence.

Tools: Confluence/SharePoint (Documentation), Jira/Azure DevOps (Risk Register), Calendar Tool, Email/Slack/Teams.

Responsibility: Project Managers (Vendor & NYCPS), Technical Leads, Executive Sponsors.

2. Activity: Detailed Requirements Elicitation (Iterative)

Implementation Steps:

1. Identify key SME groups for initial phases (e.g., OPT Routers for core routing, DIIT Security for auth/compliance, Pilot School Admins for UI).
2. Schedule and facilitate requirements workshops using techniques like:

   - User Story Mapping: Visually map user journeys and identify epics/features.
   - Interviews: One-on-one discussions with SMEs.
   - Process Modeling: Documenting AS-IS and TO-BE workflows using BPMN or similar.
   - Prototyping: Creating low-fidelity mockups (e.g., Balsamiq, Figma) to elicit UI/UX feedback early.
3. Focus elicitation on the scope of the upcoming 1-2 project phases/releases.
4. Document NFRs explicitly: Performance (latency, throughput targets), Scalability (user/bus numbers, data volume growth), Availability (SLAs per component), Accessibility (WCAG 2.0 AA), Security (specific GovCloud/NYCPS controls, encryption standards), Data Retention (7 years).
5. Document detailed data mapping requirements for integrations with existing NYCPS systems (ATS, NPSIS, etc.). Specify data fields, formats, frequency, direction.
6. Document compliance requirements check-list (FERPA, NY Ed Law 2-d, CIPA, HIPAA implications in GovCloud, NYC3 policies).

Tools: Miro/Lucidchart (Mapping), Figma/Balsamiq (Prototyping), Confluence (Documentation), Jira/ADO (Requirement Tracking).

Responsibility: Business Analysts, UX Designers, Technical Leads, NYCPS SMEs, Product Owner.

3. Activity: Requirements Analysis & Documentation

Implementation Steps:

1. Translate workshop outputs into formal User Stories in Jira/ADO.
2. Each User Story must include:
    - Clear Title (User Role, Action, Goal).
    - Detailed Description/Narrative.
    - Specific, Measurable, Achievable, Relevant, Time-bound (SMART) Acceptance Criteria (using Gherkin syntax - Given/When/Then - is highly recommended for BDD).
    - Link to related Epics/Features.
    - Initial priority and estimated effort (Story Points).
    - Relevant NFRs or Compliance constraints noted.
3. Refine process models (BPMN) and conceptual data models based on analysis.
4. Update Data Dictionary with newly identified attributes.
5. Create initial test scenario outlines linked to user stories.

Tools: Jira/Azure DevOps, Confluence, Gherkin syntax validators.

Responsibility: Business Analysts, Product Owner, Technical Leads, QA Lead.

4. Activity: Requirements Validation, Prioritization & Backlog Management

Implementation Steps:

1. Conduct formal review sessions for documented user stories and acceptance criteria with NYCPS Product Owner and SMEs. Obtain documented approval/sign-off per story or feature set.
2. Prioritize the Product Backlog collaboratively with the Product Owner based on business value, dependencies, risk, and alignment with phase goals.
3. Define and document the Definition of Ready (DoR): Checklist criteria a User Story must meet before it can be accepted into Sprint Planning (e.g., Clear Acceptance Criteria, Estimated, Dependencies Identified, UX Mockup Approved if applicable).
4. Conduct regular Backlog Grooming sessions (at least once per sprint) where the development team reviews upcoming stories, asks clarifying questions, breaks down large stories, re-estimates effort, and ensures stories meet the DoR.

Tools: Jira/Azure DevOps (Backlog, Priorities, DoR field), Confluence (Meeting Minutes).

Responsibility: Product Owner, Business Analysts, Scrum Master, Development Team Lead, Development Team.

Quality Gate & Sign-off: Formal sign-off by NYCPS Product Owner(s) on the prioritized and groomed backlog for the upcoming

phase/release, confirming requirements are understood and meet DoR.

B. Phase: Architecture & Design

1. Activity: Detailed Architecture Definition

Implementation Steps:

1. Based on refined NFRs, finalize choices for AWS GovCloud services (e.g., Kinesis vs. MSK, Fargate vs. EKS, specific RDS instance sizes per environment). Justify choices in Architecture Decision Records (ADRs) stored in Confluence/Git.
2. Create detailed network diagrams (using diagramming tools, export as images/PDF stored in Confluence/Git) showing VPCs, subnets, routing, security zones, load balancers, VPN/Direct Connect, and key traffic flows for each environment.
3. Define and document the multi-AZ HA strategy for each critical component (e.g., RDS Multi-AZ, ECS service desired count > 1 across AZs, ELB cross-zone balancing).
4. Define and document the DR strategy (e.g., Pilot Light in secondary GovCloud region), including data replication mechanisms (S3 CRR, RDS cross-region snapshots/replicas), IaC deployment process for DR, and RPO/RTO targets per service.

Tools: Lucidchart/Draw.io, Confluence, Git (for ADRs).

Responsibility: Cloud Architect, Tech Leads, Security Architect, Ops Lead.

Quality Gate & Sign-off: Formal approval of the detailed Architecture Document and HA/DR plan by the Technical Review Board.

2. Activity: Microservice & API Design

Implementation Steps:

1. Finalize microservice boundaries and responsibilities using DDD workshops/event storming if needed.
2. Define RESTful API contracts using OpenAPI Specification (OAS) v3.x for *all* inter-service communication and public-facing endpoints. Store OAS files in Git repository, ideally alongside the service code.
    - Specify paths, HTTP methods, parameters (path, query, header, cookie), request bodies, response schemas (using JSON Schema), status codes, and security schemes (e.g., JWT Bearer, API Key).
    - Use tools like Swagger Editor/UI or Stoplight for design and validation.
    - Implement API versioning strategy (e.g., URI path versioning `/v1/`).
3. Define asynchronous event schemas (e.g., using JSON Schema or Avro) for messages published to Kinesis/MSK/SNS/SQS.

Document event producers and consumers.

4. Design internal communication patterns (sync vs. async) based on coupling needs and resilience requirements. Design retry/circuit breaker logic for synchronous calls.

Tools: OpenAPI/Swagger Editor, Stoplight, Avro tools, Confluence, Git.

Responsibility: Backend Tech Leads, Backend Developers, Cloud Architect.

Quality Gate & Sign-off: Peer review and approval of API/Event schemas by Tech Leads and consuming teams.

3. Activity: Data & Security Design

Implementation Steps:

1. Create detailed Logical and Physical Data Models (ERDs for RDS, document structure for DynamoDB) including data types, constraints, relationships, indexing strategies.
2. Perform Threat Modeling (STRIDE) on key workflows (e.g., user login, PII access, route modification, GPS data flow). Document threats and planned mitigations.
3. Design Authentication flow integrating with NYCPS IdP (SAML) or Cognito. Specify token handling (JWT validation, refresh tokens).
4. Design Authorization model (RBAC). Define roles, permissions, and how they map to API endpoints and data access. Specify how permissions are checked (e.g., custom Lambda authorizer, middleware in service).

5. Define data encryption strategy: KMS keys (CMKs vs. AWS-managed), specific services requiring encryption at rest (S3, EBS, RDS, DynamoDB, SQS, SNS), TLS enforcement for transit.
6. Design secure logging/auditing strategy: What specific security events must be logged, log format, log storage/retention/access control.
7. Define specific Security Group rules per service/tier (e.g., App tier SG allows ingress from LB SG on port X, DB tier SG allows ingress from App tier SG on port Y). Define WAF rules (Managed rulesets + custom rules).

Tools: ERD tools (draw.io, ERWin), Threat modeling tools/templates, Confluence.

Responsibility: Data Architects, Security Architects, Tech Leads, Compliance Officer.

Quality Gate & Sign-off: Formal approval of Data Models, Threat Models, and Security Control designs by Technical Review Board and Security Review Board/NYC3.

4. Activity: UI/UX Design Finalization

Implementation Steps:

1. Refine wireframes and mockups into high-fidelity, pixel-perfect designs based on previous feedback.
2. Create clickable prototypes for key user flows.
3. Conduct final usability testing rounds.

4. Finalize component library and ensure alignment with NYCPS style guide and WCAG 2.0 AA requirements (color contrast, focus states, ARIA attributes planned).

Tools: Figma/Sketch/Adobe XD, InVision/Marvel (Prototyping), Accessibility testing browser plugins.

Responsibility: UX/UI Designers, Frontend Lead, Product Owner, Accessibility Specialist.

Quality Gate & Sign-off: Formal approval of final UI designs and prototypes by Product Owner and key stakeholders.

5. Activity: Test & Operations Planning

Implementation Steps:

1. Develop Master Test Plan detailing overall strategy, scope, environments, tools, roles, entry/exit criteria for each test level.
2. Develop initial Performance Test Plan outlining key scenarios, load profiles, NFR targets, and tools.
3. Develop initial Security Test Plan outlining scope for DAST, SAST, SCA, and penetration testing coordination.
4. Develop initial Monitoring & Alerting plan identifying key metrics (Golden Signals) per service, dashboard requirements, and alerting strategy.
5. Draft initial operational runbooks for common tasks and incident response.

Tools: Confluence, Test management tools (e.g., Jira Zephyr/Xray).

Responsibility: QA Lead, Performance Engineer, Security Lead, SRE/Ops Lead, Tech Leads.

C. Phase: Development (Implementation)

1. Activity: Sprint Execution (Iterative)

Implementation Steps:

1. **Sprint Planning:** Team commits to User Stories meeting DoR. Tasks created/assigned in Jira. Sprint Goal agreed.
2. **Feature Branching:** Developers create `feature/TMS-XXX` branches from `develop`.
3. **Coding (TDD/BDD):**
    - Write failing unit/integration test based on acceptance criteria.
    - Write minimal code to make test pass.
    - Refactor code and tests for clarity and efficiency.
    - Strictly adhere to secure coding standards (input validation, output encoding, auth checks, proper error handling, least

privilege). Use framework security features correctly.

   - Implement structured logging with correlation IDs.

4. **Local Testing & Verification:**
   - Run all relevant tests locally (`npm test`, `pytest`, `mvn verify`).
   - Use Docker Compose/Dev Containers to test against local dependencies.
   - Manually test functionality in local environment.
5. **Committing & Pre-commit Checks:**
   - Commit frequently with Conventional Commit messages linking to Jira issues.
   - Pre-commit hooks *must* pass (linting, formatting, secret scanning).
6. **Infrastructure Code (IaC):** Develop/update Terraform modules/configurations for resources required by the feature. Test using `terraform validate` and `terraform plan`. Commit to the *same* feature branch or a linked branch.
7. **Documentation:** Update code comments, READMEs, API specifications (OpenAPI) as needed.

Responsibility: Developers, DevOps Engineers (for IaC).

2. Activity: Code Review & Merge Request (MR) Process

Implementation Steps:

1. Developer creates MR targeting `develop`, linking issues, providing detailed description and testing notes.
2. **Automated Checks (GitLab CI on MR):**
    - Build verification.
    - Linter/Formatter checks.
    - Unit test execution & code coverage check (against threshold).
    - SAST scan (e.g., SonarQube/GitLab SAST).
    - SCA scan (e.g., GitLab Dependency Scanning).
3. **Quality Gate:** MR pipeline must pass ALL checks. Merge blocked otherwise.
4. **Peer Review:** 1-2 reviewers assigned. Reviewers perform thorough check against documented checklist (functionality, security, testing, readability, performance, standards). Constructive feedback provided via GitLab comments.
5. **Quality Gate:** Required number of reviewer approvals obtained. All comments resolved.
6. **Merge:** Developer/Lead merges using Squash option, ensuring commit message links MR/ticket. Feature branch deleted.

Responsibility: Developers, Reviewers, Tech Leads.

3. Activity: Continuous Integration on `develop`

Implementation Steps:

1. Merge to `develop` triggers `develop` branch pipeline in GitLab CI.
2. Pipeline Stages: Build -> Static Analysis -> Unit Test -> Package (Build & Push Docker image to ECR / Zip to S3).
3. **Quality Gate:** Pipeline must pass all stages. Failure sends immediate notifications (Slack/Email).
4. Successful pipeline run makes the artifact available for deployment to DEV environment.

Responsibility: CI/CD System.

D. Phase: Continuous Testing & Quality Assurance

1. Activity: Automated Test Execution (Integrated with CI/CD)

Implementation Steps:

1. **Unit & Component Tests:** Run automatically as part of CI pipeline on every commit/MR (triggered by GitLab CI). Results

reported back to MR/pipeline status.

2. **Integration & API Tests:** Run automatically *after* successful deployment to DEV and QA environments (triggered by GitLab CD). Results reported back to pipeline status. Failures block promotion.
3. **E2E Tests:** Run automatically against QA and Staging environments (e.g., nightly, post-deployment). Results reported to QA team/dashboards. Critical failures block promotion/trigger alerts.
4. **Security Tests (DAST, Container Scan):** Run automatically against QA/Staging environments (post-deployment, nightly). Results fed into defect tracking/security dashboards. Critical findings block promotion.
5. **Accessibility Tests:** Run automatically as part of E2E suite against QA/Staging. Results reported for review/remediation.
6. **Performance Tests:** Run manually or scheduled against dedicated Perf environment before major releases or significant changes. Results analyzed against NFRs.

Responsibility: CI/CD System, Developers (fixing test failures), QA Team (managing E2E/Perf/Accessibility suites), Security Team (managing DAST/Container scans).

2. Activity: Manual & Exploratory Testing

Implementation Steps:

1. QA Engineers perform exploratory testing based on user stories and risk areas in the QA environment throughout the

sprint.

2. QA verifies bug fixes deployed to QA environment.
3. QA performs targeted manual regression testing for high-risk areas before releases.
4. QA facilitates and supports UAT sessions in the Staging environment, providing test scenarios and guidance to NYCPS stakeholders.
5. Accessibility Specialist performs manual checks (keyboard, screen reader) on key flows in Staging.

Responsibility: QA Team, Accessibility Specialist, NYCPS UAT Participants.

3. Activity: Defect Triage & Management

Implementation Steps:

1. All failed automated tests automatically create or link to defects in Jira/ADO. Manual defects logged by QA/UAT participants.
2. Daily defect triage meeting ("Bug Scrub") involving QA Lead, Dev Lead, Product Owner (or rep) to review new defects, prioritize, and assign them for fixing within the current or next sprint.
3. Track defect resolution status and verification in Jira/ADO. Maintain metrics on defect density, resolution time, etc.

Responsibility: QA Lead, Dev Lead, Product Owner, Developers, QA Engineers.

Quality Gates: Defined pass/fail criteria for each automated test suite integrated into CI/CD pipelines. Formal UAT sign-off. Go/No-Go decision based on outstanding defect severity/count before production release. Performance NFRs met. No unresolved critical/high security vulnerabilities.

E. Phase: Deployment & Release Management

1. Activity: Automated Deployment to Environments

Implementation Steps:

1. Configure GitLab CI/CD pipelines (`.gitlab-ci.yml`) with distinct stages and jobs for deploying to DEV, QA, Staging, Perf, and PROD environments.
2. Use environment-specific GitLab CI/CD variables for configurations (endpoints, feature flags, resource names). Securely inject secrets needed for deployment (e.g., AWS credentials via OIDC or Vault integration).
3. **DEV Deployment:** Trigger automatically on merge to `develop` after successful CI build.
4. **QA Deployment:** Trigger manually or scheduled (e.g., nightly) from successful `develop` builds.

5. **Staging/UAT Deployment:** Trigger manually from a `release/` branch build.
6. **Perf Deployment:** Trigger manually from a `release/` branch build to the dedicated performance testing environment.
7. **PROD Deployment:** Trigger manually *only* from a tagged commit on the `main` branch. **Mandatory:** Include a `when: manual` step in the GitLab CI job requiring explicit action by authorized personnel (Release Manager, Ops Lead) in the GitLab UI.**
8. Implement Blue/Green or Canary deployment strategies using AWS CodeDeploy integrated with ECS/Fargate or via ALB/Route 53 weighting managed by deployment scripts/Terraform.
9. Integrate database migration script execution (Flyway/Liquibase) as a controlled step within the deployment pipeline *before* application traffic is shifted to the new version. Implement rollback procedures for migrations.

Tools: GitLab CI/CD, AWS CodeDeploy, Terraform, Flyway/Liquibase, Docker, ECR, S3, AWS Systems Manager Parameter Store/Secrets Manager.

Responsibility: DevOps Team, CI/CD System.

2. Activity: Infrastructure Provisioning (Terraform via CI/CD)

Implementation Steps:

1. Create dedicated GitLab CI jobs for `terraform plan` and `terraform apply` for each environment.
2. Configure jobs to run in the correct `environments/` directory using appropriate `.tfvars`.
3. **Plan Job:** Runs on MRs targeting `develop`/`release`/`main` branches and on merges. Saves plan file (`tfplan.out`) as an artifact.
4. **Apply Job (DEV/QA):** Runs automatically after Plan job succeeds on merges to `develop`. Applies the saved plan artifact.
5. **Apply Job (Staging/PROD):** Includes `when: manual` trigger. Requires authorized user to click 'play' in GitLab UI after reviewing the plan artifact from the preceding job. Applies the saved plan artifact.
6. Securely provide AWS credentials to the runner executing Terraform (prefer temporary credentials via OIDC or AssumeRole).

Responsibility: DevOps Team, CI/CD System, Authorized Approvers (for Staging/Prod).

Quality Gates: `terraform plan` must succeed. Manual review and approval of plan required for Staging/Prod `apply`. `terraform apply` must complete successfully.

3. Activity: Release Management & Governance

Implementation Steps:

1. Follow SemVer for versioning releases tagged on `main`.
2. Generate automated Release Notes from Conventional Commits using GitLab Releases feature. Supplement with manual overview.
3. Conduct formal Release Readiness Review meeting prior to production deployment approval. Review test results, open defects, performance metrics, security scan summaries, rollback plan. Obtain Go/No-Go decision.
4. Integrate production deployment approval with NYCPS CCB process. Use GitLab's manual approval gate as evidence.
5. Communicate release schedule and potential impact to all stakeholders in advance.

Responsibility: Release Manager, Project Manager, Tech Leads, QA Lead, Ops Lead, Security Lead, Product Owner, CCB.

F. Phase: Operations, Observability & Incident Response

1. Activity: Monitoring & Alerting Implementation

Implementation Steps:

1. Configure CloudWatch Agent/Fluentd/OTel Collector to ship structured logs and metrics from all compute resources (EC2,

Fargate, Lambda) to CloudWatch Logs and Metrics.

2. Configure CloudWatch Synthetics Canaries for key user endpoint monitoring (API checks, UI workflow checks).
3. Set up CloudWatch Alarms with specific, actionable thresholds for the "Four Golden Signals" (Latency, Traffic, Errors, Saturation) per service, plus key business metrics and security events (GuardDuty findings, WAF blocks).
4. Configure SNS topics for alert routing based on severity (e.g., `critical-prod-alerts`, `warning-prod-alerts`, `nonprod-alerts`).
5. Integrate SNS topics with PagerDuty/Opsgenie for on-call notifications for critical production alerts. Integrate with Slack/Teams for lower severity/non-prod alerts.
6. Build comprehensive dashboards in CloudWatch/Grafana/QuickSight visualizing key SLIs, operational metrics, logs, and traces per service and environment.

Tools: CloudWatch (Logs, Metrics, Alarms, Synthetics), SNS, PagerDuty/Opsgenie, Grafana/QuickSight, AWS X-Ray/OpenTelemetry.

Responsibility: SRE/Ops Team, DevOps Team.

2. Activity: Incident Management & SRE Practices

Implementation Steps:

1. Define and document the Incident Management process: Roles (Incident Commander, Comms Lead, Tech Lead), severity

levels (SEV1-SEV4), communication channels (War Room, Status Page), escalation matrix, RCA process.

2. Configure PagerDuty/Opsgenie with on-call schedules and escalation policies.
3. Integrate alerting tools with incident tracking (e.g., automatically create Jira tickets for critical alerts).
4. Conduct regular blameless post-mortems for SEV1/SEV2 incidents, tracking action items rigorously.
5. Define and track SLOs/SLIs for critical services. Calculate error budgets and use them to inform prioritization between feature work and reliability improvements.
6. Implement automated operational tasks ("ChatOps" via Slack/Teams bots if appropriate) for common diagnostic or remediation actions.

Tools: PagerDuty/Opsgenie, StatusPage.io, Jira/ADO, Confluence, Slack/Teams.

Responsibility: SRE/Ops Team, Incident Commander (during incidents), Development Teams (participating in RCAs).

3. Activity: Ongoing Maintenance & Optimization

Implementation Steps:

1. Establish regular cadence for reviewing and applying OS/runtime/dependency patches using automated tools (Systems Manager Patch Manager) and CI/CD pipelines, starting with lower environments.

2. Schedule and perform regular DR tests (failover/failback).
3. Continuously monitor AWS costs and implement optimization recommendations (right-sizing, storage tiering, RI/SP purchases).
4. Periodically review and tune CloudWatch alarm thresholds based on historical performance.
5. Review security logs and findings (GuardDuty, Security Hub, WAF logs) regularly.

Responsibility: SRE/Ops Team, Security Team, DevOps Team.

G. Phase: Feedback & Continuous Improvement

1. Activity: Collect & Analyze Feedback

Implementation Steps:

1. Establish channels for collecting end-user feedback (in-app forms, support tickets, UAT sessions, surveys).
2. Analyze operational data: monitoring metrics, incident trends, log patterns, CI/CD pipeline performance (DORA metrics).
3. Conduct regular Agile Retrospectives (per sprint) focusing on process improvements for the DevSecOps workflow itself.

4. Review SLO adherence and error budget consumption.

Responsibility: Product Owner, Project Manager, Scrum Master, SRE/Ops, QA, Dev Teams.

2. Activity: Integrate Learnings into Backlog

Implementation Steps:

1. Product Owner/BAs translate validated user feedback and operational requirements into new User Stories or Bugs in the backlog.
2. Action items from Incident RCAs and Retrospectives are added to the backlog as technical debt, bug, or process improvement tasks.
3. Prioritize these items alongside new feature development during Backlog Grooming and Sprint Planning, ensuring a balance between new functionality and system health/reliability/security.

Responsibility: Product Owner, Scrum Master, Tech Leads.