NYCPS TMS: Prescriptive End-to-End DevSecOps Strategy & Implementation

I. Introduction: Philosophy & Goals

This document mandates the comprehensive DevSecOps strategy and implementation plan for the NYCPS Transportation Management System (TMS) project. Our philosophy integrates development, security, testing, and operations into a unified, automated, and collaborative workflow, executed within an Agile (Scrum) framework using GitLab as the central platform and deploying to AWS GovCloud.

The non-negotiable goals driving this strategy are:

1. **Best-in-Class Developer Experience & DORA Metrics:** We will empower our globally distributed, multi-stack team with standardized tools, efficient local environments, rapid feedback loops via CI, and minimal bureaucratic overhead to maximize

productivity and achieve excellent DORA metrics (Deployment Frequency, Lead Time for Changes, Change Failure Rate, Time to Restore Service).

2. **Hyper-Fast, Reliable CI/CD Pipeline:** We will implement highly optimized, parallelized, and cached CI/CD pipelines in GitLab to enable rapid, reliable delivery of changes from code commit to production deployment, facilitating fast time-to-market for features and fixes.

3. **Rigorous Automated Quality & Security Gating:** We will enforce stringent, automated quality and security gates at *every* stage of the pipeline (pre-commit, CI, post-deploy). Only code meeting the highest standards for correctness, performance, accessibility, and security will progress towards production. Developer ownership of testing is paramount.

4. **Maximum Automation:** We will automate every feasible aspect of the SDLC, including builds, testing (unit, integration, E2E, performance, security, accessibility), infrastructure provisioning (IaC), configuration management, deployments, monitoring, and alerting to reduce manual effort, minimize errors, and increase speed and reliability.

II. Foundational Setup & Tooling Implementation

Before development commences, we will establish the core tools, configurations, and standards.

A. Developer Environment Standardization

1. Local Machine Setup

- **Action:** Document and provide standardized setup instructions/scripts for developer machines (macOS/Linux/WSL2) covering Git, Docker Desktop, AWS CLI v2 (configured for GovCloud with SSO/AssumeRole), Terraform CLI, required language runtimes (Node, Python, Java via version managers), package managers (npm, pip, Maven), and build tools.
- **Responsibility:** DevOps/Platform Team.

2. IDE Configuration

- **Action:** Mandate use of approved IDEs (VS Code, IntelliJ, PyCharm, WebStorm) with required extensions (Language support, Linters/Formatters, Git, Docker, AWS Toolkit, Terraform, Testing frameworks). Provide shared IDE settings files/profiles for consistency.
- **Action:** Configure IDEs for auto-linting and auto-formatting on save based on project standards.
- **Responsibility:** DevOps Team, Development Leads.

3. Pre-commit Hooks Setup

- **Action:** Implement Git pre-commit hooks using tools like `Husky` (for JS/TS projects) or `pre-commit` (multi-language framework).
- **Action:** Configure hooks to automatically run:
    - Linters (ESLint, Flake8, Checkstyle, etc.).
    - Formatters (Prettier, Black, etc.).
    - Basic credential scanners (e.g., `gitleaks`, `trufflehog`) to prevent accidental secret commits.
    - (Optional but Recommended) Fast-running subset of critical unit tests.
- **Quality Gate:** Commits *must* pass all pre-commit checks.
- **Responsibility:** DevOps Team, Development Leads.

4. Local Testing Setup

- **Action:** Document procedures for running unit, integration, and component/API tests locally (e.g., `npm test`, `pytest`, `mvn test`).
- **Action:** Provide Docker Compose files or Dev Container configurations (`devcontainer.json` for VS Code) to easily spin up local dependencies (databases, caches, simulators like LocalStack) for integration testing.
- **Responsibility:** Development Teams, DevOps Team.

B. GitLab Setup & Configuration

1. Group & Project Structure

- **Action:** Create a top-level GitLab Group for the NYCPS TMS project.
- **Action:** Establish repositories within the group. Adopt a **poly-repo** strategy: separate repositories for each microservice, frontend application, shared library, and the Terraform IaC codebase. Name repositories clearly (e.g., `tms-gps-ingestion-service`, `tms-parent-app-react`, `tms-infra-terraform`).
- **Responsibility:** DevOps Lead, Project Management.

2. Repository Settings

- **Action:** Configure Branch Protection rules for `main` and `develop` branches:
    - Prevent direct pushes.
    - Require Merge Requests (MRs).
    - Require CI pipeline success before merging.
    - Require a minimum number of reviewer approvals (e.g., 1 or 2 depending on criticality).
    - Prevent authors from approving their own MRs.
    - (Optional) Require resolution of all comments/threads.
- **Action:** Configure repository settings to enforce Conventional Commit message format (e.g., using commit message linting in CI or server-side hooks if available).

- **Action:** Set up GitLab Issue tracking integration (linking commits/MRs to issues).
- **Responsibility:** DevOps Lead, Repository Maintainers.

3. CI/CD Runner Configuration

- **Action:** Provision and register GitLab Runners capable of executing project jobs. Use dedicated runners (e.g., EC2 instances in AWS GovCloud managed via GitLab Runner Operator on EKS, or SaaS runners if using GitLab.com with appropriate security).
- **Action:** Configure runners with necessary tags (e.g., `docker`, `aws`, `terraform`, `nodejs`, `python`) to route jobs appropriately. Ensure runners have necessary tools installed (Docker, AWS CLI, Terraform, language runtimes) or use Docker-in-Docker / appropriate base images in job definitions.
- **Action:** Configure runner caching (e.g., S3 cache) to speed up builds (dependency downloads, Docker layers).
- **Responsibility:** DevOps/Platform Team.

4. GitLab CI/CD Variables & Secrets Management

- **Action:** Configure GitLab CI/CD Variables at the Group or Project level for environment-specific settings (AWS regions, account IDs, cluster names, non-sensitive API endpoints). Use "Protected" variables for sensitive environments like Staging/Prod, only accessible to protected branches/tags.
- **Action:** Integrate GitLab with a secure secrets manager (like HashiCorp Vault or AWS Secrets Manager) for handling highly sensitive credentials (AWS keys for runners if not using IAM roles,

database passwords needed during CI, third-party API keys). *Avoid storing sensitive secrets directly in GitLab CI/CD variables.*

- **Responsibility:** DevOps Team, Security Team.

5. Package & Container Registries

- **Action:** Enable and configure GitLab's built-in Container Registry for Docker images.
- **Action:** Enable and configure GitLab's built-in Package Registry for language-specific packages (npm, Maven, PyPI) if desired, or configure integration with AWS CodeArtifact/external repos.
- **Responsibility:** DevOps Team.

III. Development, Code Review & Continuous Integration (CI)

This phase covers the core loop of coding features, ensuring quality through reviews and automated checks, and integrating code into the main development line.

A. Feature Development Workflow (per User Story/Task)

1. Branch Creation

- **Action:** Developer checks out the latest `develop` branch (`git checkout develop && git pull origin develop`).
- **Action:** Developer creates a feature branch using the naming convention `feature/TICKET-###-short-description` (e.g., `feature/TMS-123-add-gps-timestamp`).
- **Responsibility:** Developer.

2. Coding & Local Testing

- **Action:** Developer writes application code and associated automated tests (Unit, Integration) following TDD/BDD principles and adhering to secure coding standards and style guides.
- **Action:** Developer frequently runs linters, formatters, and tests locally using IDE integration or CLI commands (`npm run lint`, `npm test`, `pytest`, `mvn verify`).
- **Action:** Developer uses local environment (Docker Compose/Dev Containers) to test interactions with dependent services (databases, caches, mocks).
- **Responsibility:** Developer.

3. Committing Code

- **Action:** Developer commits changes frequently to the feature branch with clear, concise messages adhering to the Conventional Commits standard (e.g., `feat(api): add endpoint for route retrieval #TMS-124`, `fix(ui): correct ETA display formatting #TMS-125`, `test(svc): add unit tests for validation logic #TMS-126`).

- **Action:** Pre-commit hooks automatically run linters, formatters, credential scans, and potentially fast unit tests. **Quality Gate:** Commit is blocked if hooks fail.
- **Responsibility:** Developer.

B. Code Review via Merge Request (MR)

1. MR Creation

- **Action:** Once the feature is complete and all local tests pass, developer pushes the feature branch to GitLab (`git push origin feature/TMS-123...`).
- **Action:** Developer creates a Merge Request in GitLab targeting the `develop` branch.
- **Action:** Developer fills out the MR template, linking the relevant Jira/ADO ticket(s), providing a clear summary of changes, explaining the "why", detailing testing performed, and adding specific instructions for reviewers or QA.
- **Responsibility:** Developer.

2. Automated Pre-Merge Checks (CI Pipeline Triggered by MR)

- **Action:** GitLab CI pipeline automatically runs on the feature branch/MR context.
- **Pipeline Stages:****
    1. **Build: Compile code, install dependencies.**
    2. **Static Analysis: Run Linters, SAST (SonarQube/GitLab SAST), SCA**

(Dependency Scanning).

3. Unit Test: Execute all unit tests, calculate code coverage.
4. Package (Dry Run): Build container images/packages without pushing.

- Quality Gate: MR is blocked from merging if any stage fails (e.g., build error, lint error, unit test failure, coverage below threshold [e.g., 80%], critical/high SAST/SCA vulnerability found). Pipeline status clearly visible on the MR.
- Responsibility: CI/CD System, Developer (to fix failures).

3. Peer Code Review

- Action: Developer assigns designated peer reviewer(s) (1-2 depending on team policy/change complexity).
- Action: Reviewers thoroughly examine the code changes against the checklist (Functionality, Security, Testing, Readability, Performance, Standards).
- Action: Reviewers provide constructive feedback via GitLab comments/threads or approve the MR. Use GitLab's "suggestion" feature for minor fixes.
- Quality Gate: MR requires explicit approval from the required number of reviewers.
- Responsibility: Designated Reviewers, Developer (to address comments).

4. Merging

- **Action: Once all discussions are resolved, required approvals are given, and the CI pipeline succeeds, the developer (or designated integrator) merges the MR into `develop`.**
- **Action: Use the "Squash commits" option when merging to keep the `develop` branch history clean (one commit per feature/fix). Ensure the squashed commit message links back to the MR and ticket ID.**
- **Action: Delete the feature branch after successful merge.**
- **Responsibility: Developer / Integrator.**

C. Continuous Integration on `develop` Branch

1. **Post-Merge Pipeline**
    - **Action: A merge to `develop` automatically triggers a more comprehensive CI pipeline.**
    - **Pipeline Stages:****
        1. **Build: (Same as MR pipeline)**
        2. **Static Analysis: (Same as MR pipeline)**
        3. **Unit Test: (Same as MR pipeline)**
        4. **Package: Build *and push* artifacts (Docker images to ECR, packages to GitLab Registry/CodeArtifact, Lambda zips to S3) tagged with commit hash or build ID.**

5. **(Optional) Deploy to DEV: Automatically trigger CD pipeline to deploy to the Development environment (see next section).**

- **Quality Gate: Failures in any stage block subsequent stages and trigger notifications.**
- **Responsibility: CI/CD System.**

IV. Continuous Testing (Automated Quality Gates)

Automated testing is integrated throughout the pipeline to provide rapid feedback and enforce quality gates.

A. Testing Levels & Pipeline Integration

1. Unit Tests

- **Trigger: Run locally by developers; run automatically in CI pipeline on every commit/MR and merge to `develop`.**
- **Scope: Individual functions/classes/components in isolation.**
- **Tools: Jest, Pytest, JUnit, etc.**

- **Quality Gate (CI): Build fails if any unit test fails. Code coverage checked against threshold (e.g., 80% line/branch); build fails if below target.**
- **Responsibility: Developer (writing & fixing), CI System (execution).**

2. Integration Tests

- **Trigger: Run locally by developers against test containers; run automatically in CI/CD pipeline *after* deployment to DEV/QA environments.**
- **Scope: Interaction between a service and its direct dependencies (database, cache, message queue, closely coupled internal APIs - potentially mocked).**
- **Tools: Testcontainers, Pytest fixtures, Spring Boot integration tests, etc.**
- **Quality Gate (Post-Deploy): Deployment to next stage (e.g., QA -> Staging) blocked if critical integration tests fail.**
- **Responsibility: Developer (writing & fixing), CI/CD System (execution).**

3. API Contract Tests

- **Trigger: Run automatically in CI/CD pipeline *after* deployment to DEV/QA environments.**
- **Scope: Validating API request/response schemas against OpenAPI specs. Testing authentication/basic authorization.**
- **Tools: Postman/Newman, RestAssured, Pytest with `requests`, Pact (Consumer-Driven Contracts).**

- **Quality Gate (Post-Deploy): Deployment to next stage blocked if API contract tests fail.**
- **Responsibility: Developer (writing & fixing), CI/CD System (execution).**

4. Component Tests (UI)

- **Trigger: Run locally by developers; run automatically in CI pipeline on every commit/MR and merge to `develop`.**
- **Scope: Testing UI components in isolation or groups, mocking backend API calls.**
- **Tools: React Testing Library, Cypress Component Testing.**
- **Quality Gate (CI): Build fails if component tests fail.**
- **Responsibility: Frontend Developer (writing & fixing), CI System (execution).**

5. End-to-End (E2E) Tests

- **Trigger: Run automatically in CI/CD pipeline *after* deployment to QA and/or Staging environments (can be time-consuming, may run less frequently or in parallel).**
- **Scope: Simulate critical user journeys across multiple services and UI layers.**
- **Tools: Cypress, Playwright, Selenium.**
- **Quality Gate (Post-Deploy): Deployment to next stage (e.g., Staging -> Prod) blocked if critical E2E flows fail. Failures trigger investigation.**

- **Responsibility: QA Team (primary suite management), Developers (contributing tests for new features), CI/CD System (execution).**

6. Security Testing (Automated)

- **SAST: Run in CI pipeline on MRs and merges to `develop`. (See CI Section). **Quality Gate:** Block merge/build on high/critical findings.**
- **SCA: Run in CI pipeline on MRs and merges to `develop`. (See CI Section). **Quality Gate:** Block merge/build on high/critical findings.**
- **DAST: Run automatically (e.g., nightly or post-deploy) against QA/Staging environments using tools like OWASP ZAP integrated into GitLab CI/CD. **Quality Gate:** High/critical findings block promotion to Prod and create high-priority bugs.**
- **Container Scanning: Run automatically after image build in CI or via ECR scanning features. **Quality Gate:** High/critical findings block deployment.**
- **Responsibility: CI/CD System (execution), Security Team (tool config & rule tuning), Developers (fixing findings).**

7. Accessibility Testing (Automated)

- **Trigger: Run automatically in CI/CD pipeline *after* deployment to QA/Staging environments, integrated with E2E test runs.**
- **Scope: Scan key UI pages/components for common WCAG violations.**

- **Tools: Axe-core integrated with Cypress (`cypress-axe`) or Playwright.**
- **Quality Gate (Post-Deploy): Report generated. Critical violations may block promotion depending on severity and defined policy.**
- **Responsibility: Frontend Developers (fixing findings), QA Team (managing tests), CI/CD System (execution).**

8. Performance Testing (Automated)

- **Trigger: Run automatically (e.g., nightly or on-demand before release) against a dedicated, production-scaled Performance Testing environment deployed via the CD pipeline.**
- **Scope: Simulate realistic user load (mix of API calls, user interactions) based on expected peak usage patterns. Measure response times, throughput, error rates, resource utilization (CPU, memory, DB connections).**
- **Tools: k6, JMeter, Gatling (scripts version controlled).**
- **Quality Gate (Pre-Release): Performance results compared against NFRs/SLOs. Significant regressions or failure to meet targets block production release.**
- **Responsibility: Performance Engineering/QA Team (scripting, execution, analysis), Developers (addressing bottlenecks), CI/CD System (execution).**

V. Continuous Deployment (CD) & Release Management

We will automate deployments to all environments using GitLab CI/CD, incorporating quality gates and appropriate deployment strategies.

A. Deployment Pipeline Flow

1. **Merge to `develop`**: Triggers CI pipeline (Build, Unit Test, Scan, Package). On success, automatically triggers deployment to **DEV** environment.
2. **Post-DEV Deploy**: Automated integration and API contract tests run against DEV.
3. **Manual Trigger/Schedule**: Deploy approved `develop` build from DEV to **QA** environment.
4. **Post-QA Deploy**: Automated E2E, DAST, Accessibility tests run against QA. Manual exploratory testing by QA team occurs here.
5. **Release Branch Creation**: When ready for a release candidate, create `release/vX.Y.Z` branch from `develop`.
6. **Deploy Release Branch**: Trigger deployment of the `release/` branch to **Staging/UAT** environment.
7. **Post-Staging/UAT Deploy**: Re-run E2E, DAST, Accessibility tests. Conduct formal UAT with NYCPS

stakeholders. Run performance tests against dedicated Perf environment (deployed from `release/` branch). Address any critical bugs via commits to the `release/` branch (which triggers re-deploy/re-test).

8. **Production Release Preparation**:
    - Final Release Readiness Review and Go/No-Go decision.
    - Merge `release/vX.Y.Z` branch into `main` and tag `main` with `vX.Y.Z`.
    - Merge `release/vX.Y.Z` branch back into `develop` to incorporate release bug fixes.
9. **Production Deployment**: Trigger CD pipeline from the tag on `main`. **Quality Gate:** Requires manual approval step in GitLab CI pipeline (authorized personnel only).
10. **Post-Prod Deploy**: Execute automated smoke tests. Intensive monitoring. Phased rollout if using Canary strategy.

B. Infrastructure Deployment (Terraform via GitLab CI/CD)

- **Pipeline Integration:** Create dedicated GitLab CI/CD jobs for running `terraform plan` and `terraform apply`.
- **Workspaces/Directories:** Configure jobs to run within the correct `environments/` subdirectory based on the target environment.

- **Plan Stage:** Run `terraform plan -var-file=".tfvars" -out="tfplan.out"`. Store the `tfplan.out` file as a pipeline artifact.
- **Apply Stage:**
    - For Dev/QA: Automatically run `terraform apply "tfplan.out"`.
    - For Staging/Prod: Include a `when: manual` step requiring authorized user interaction in the GitLab UI to trigger `terraform apply "tfplan.out"`.
- Security:** Configure GitLab Runners with secure access to AWS GovCloud (e.g., using IAM roles via OIDC federation if possible, or securely managed temporary credentials).

C. Deployment Strategies

- Default: Blue/Green:** For ECS/Fargate services, configure CodeDeploy (integrated with GitLab CI) to perform Blue/Green deployments. This allows zero-downtime traffic shifting and instant rollback.
- Optional: Canary Releases:** For high-risk changes or initial feature rollouts, configure CodeDeploy or use ALB weighted target groups/Lambda aliases (managed via Terraform/CI scripts) to gradually shift traffic to the new version while monitoring metrics.

D. Release Management & Governance

- **Release Versioning:** Use Semantic Versioning (SemVer - MAJOR.MINOR.PATCH) for releases tagged on `main`.**
- **Release Notes:** Automatically generate baseline release notes from Conventional Commit messages between release tags. Augment with manual summaries of key features/fixes. Publish using GitLab Releases.**
- **Change Control Board (CCB):** Integrate with NYCPS CCB process for production deployment approvals. GitLab's manual approval gates can represent CCB sign-off within the pipeline.**
- **Rollback Plan:** Maintain documented rollback procedures for each type of deployment (application code, infrastructure, database migration). Test rollback procedures periodically.**

VI. Production Operations, Observability & Incident Response

Post-deployment, we will focus on maintaining system health, performance, and security through proactive monitoring, alerting, and a robust incident response process.

A. Monitoring & Observability

- **Metrics (The Four Golden Signals):**
    - **Latency: Track request duration for APIs, database queries, page loads (CloudWatch metrics from ALB, API Gateway, Lambda; APM tools; frontend monitoring).**
    - **Traffic: Monitor request rates (requests/sec) for services (ALB, API Gateway, Lambda). Monitor queue depths (SQS), stream throughput (Kinesis).**
    - **Errors: Track HTTP error codes (4xx, 5xx from ALB/API GW), application exceptions (Lambda errors, logs), database errors (RDS logs/metrics).**
    - **Saturation: Monitor resource utilization (CPU, Memory, Disk I/O for EC2/RDS/ECS; Concurrency limits for Lambda; Connection counts for DBs).**
- **Logging: Implement centralized, structured (JSON) logging for all applications and infrastructure components using CloudWatch Logs. Include correlation IDs to trace requests across services.**
- **Distributed Tracing: Implement OpenTelemetry (or AWS X-Ray) across microservices to trace requests as they flow through the system, identifying bottlenecks and dependencies.**

- **Dashboarding: Create environment-specific dashboards in CloudWatch (or Grafana/Datadog) visualizing key metrics (Golden Signals, business KPIs like active routes, successful boardings), log queries, and trace data. Provide high-level dashboards for OPT leadership and detailed dashboards for SRE/Dev teams.**

B. Alerting

- **Strategy: Focus on actionable alerts based on symptoms (high latency, high error rates, resource saturation, critical security events) rather than just causes. Define clear thresholds and evaluation periods in CloudWatch Alarms.**
- **Routing: Route alerts via SNS to appropriate channels based on severity and service:**
    - **Critical Production Issues: PagerDuty/Opsgenie triggering on-call rotation for SRE/Ops.**
    - **Warning/Non-Critical Production Issues: Dedicated Slack/Teams channel for Ops/Dev.**
    - **Test Environment Failures: Slack/Teams channel for relevant Dev/QA team.**
- **Runbooks: Develop runbooks linked to critical alerts, outlining initial diagnostic steps and mitigation procedures.**

C. Incident Response & SRE

- **On-Call Rotation: Establish a formal on-call schedule for SRE/Ops team for production support.**
- **Incident Commander Role: Define roles and responsibilities during major incidents.**
- **Communication: Clear internal (war room/Slack channel) and external (status page, stakeholder notifications) communication protocols during incidents.**
- **Root Cause Analysis (RCA): Conduct blameless post-mortems for all significant production incidents. Document findings, contributing factors, timeline, impact, corrective actions, and lessons learned. Track action items to completion.**
- **Service Level Objectives (SLOs): Define measurable SLOs for key user journeys/services (e.g., API latency < 200ms P95, login success rate > 99.9%). Monitor SLIs (Service Level Indicators) and use error budgets to balance reliability work with feature development.**
- **Chaos Engineering (Future): Consider implementing controlled chaos engineering experiments in staging environments to proactively test system resilience.**

D. Support Handoff

- **Ensure clear documentation and training are provided to Tier 1/2 support teams (if distinct from SRE/DevOps) on using**

monitoring tools, dashboards, basic troubleshooting steps, and the escalation process.

VII. Feedback & Continuous Improvement

The DevOps lifecycle is a continuous loop. Feedback from production operations, testing, and users directly informs future development cycles.

- Retrospectives: Use Agile retrospectives to identify bottlenecks or inefficiencies in the *DevOps process itself* and implement improvements.
- Monitoring -> Backlog: Insights from monitoring (performance bottlenecks, frequent non-critical errors) should generate tasks/stories in the product backlog.
- Incident RCA -> Backlog: Action items from RCAs (e.g., fix bug, improve monitoring, add resilience pattern) *must* be added to the backlog and prioritized.
- User Feedback -> Backlog: Feedback collected from end-users (via support channels, surveys, UAT) should be reviewed by the Product Owner and translated into potential backlog items.
- Metric Review: Regularly review DORA metrics and SLO adherence to identify systemic areas for improvement in the

development and deployment process.