NYCPS Transportation Management System: Detailed End-to-End SDLC

Introduction & Methodology

This document details a modern, best-in-class Software Development Lifecycle (SDLC) tailored for the NYCPS Transportation Management System project (RFP R1804). Given the project's scale, complexity, security requirements, deployment on AWS GovCloud, and the 12-month timeline, this SDLC adopts an **Agile (Scrum/Kanban hybrid) methodology integrated with DevSecOps principles**.

This approach emphasizes:

- **Iterative Development:** Delivering working software in frequent, incremental cycles (sprints/iterations) aligned with the phased project plan.
- **Collaboration:** Continuous communication and feedback between development teams, security teams, operations, and NYCPS stakeholders.
- **Automation:** Maximizing automation in testing, integration, deployment, and infrastructure management.
- **Security Integration ("Shift Left"):** Embedding security considerations and testing throughout the entire lifecycle, not just at the end.
- **Flexibility & Adaptability:** Ability to respond to changing requirements and feedback effectively.
- **Quality Focus:** Building quality in through practices like code reviews, extensive automated testing, and clear Definitions of Done (DoD).

Phase 1: Planning & Requirements Refinement

Goal: Establish a shared understanding of project goals, scope, priorities, and detailed requirements for upcoming iterations/phases.

Best Practices:

- **Agile Ceremonies:** Conduct regular sprint planning, daily stand-ups, sprint reviews, and retrospectives.
- **Product Backlog Management:** Maintain a prioritized product backlog (using tools like Jira, Azure DevOps Boards). Epics broken down into User Stories.
- **User Story Definition:** Write user stories adhering to INVEST criteria (Independent, Negotiable, Valuable, Estimable, Small, Testable) with clear acceptance criteria.
- **Stakeholder Collaboration:** Regular backlog grooming and requirements validation sessions with OPT, DIIT, School Admin representatives, and other key stakeholders.
- **Definition of Ready (DoR):** Define clear criteria for when a backlog item is ready to be pulled into a sprint.
- **Phase/Sprint Goal Setting:** Clearly define the goals for each major project phase and individual sprint.
- **Compliance Mapping:** Explicitly map requirements to relevant compliance standards (FERPA, NY Ed Law 2-d, WCAG, RFP NFRs).

Tool Examples: *Jira, Azure DevOps Boards, Confluence, Miro.*

Phase 2: Architecture & Design

Goal: Define the system's technical architecture, design individual components/microservices, plan data models, and design user interfaces, ensuring alignment with requirements (functional, non-functional, security, compliance).

Best Practices:

- **Architecture Definition:** Refine and detail the AWS GovCloud architecture (as previously outlined), selecting specific services and configurations. Document key architectural decisions (ADRs).
- **Microservice Design:** Apply Domain-Driven Design (DDD) principles to define bounded contexts and design loosely coupled microservices.
- **API-First Design:** Define clear API contracts (e.g., using OpenAPI/Swagger) for inter-service communication and frontend consumption *before* implementation.
- **Data Modeling:** Design database schemas (relational and NoSQL) considering normalization, performance, and scalability. Plan data migration strategies if needed.
- **Security Design & Threat Modeling:** Conduct threat modeling exercises (e.g., STRIDE) for critical components and data flows. Design security controls (authentication, authorization, encryption) based on threats and requirements. Review designs against secure coding standards.
- **Infrastructure as Code (IaC) Design:** Plan IaC structure using CloudFormation/CDK/Terraform modules for reusability and maintainability.
- **UX/UI Design & Prototyping:** Create wireframes, mockups, and interactive prototypes. Conduct usability testing with target user

representatives (drivers, parents, admins). Ensure WCAG 2.0 AA compliance in designs.

- **High Availability/Disaster Recovery Design:** Detail multi-AZ deployment strategies, database replication/failover mechanisms, backup strategies, and DR procedures based on RPO/RTO targets.

Tool Examples: *Lucidchart/Draw.io (Diagramming), Figma/Sketch/Adobe XD (UI/UX), Swagger Editor/Stoplight (API Design), AWS Well-Architected Framework Tool.*

Phase 3: Development (Implementation)

Goal: Build, test, and integrate software components according to design specifications and user stories within agile sprints.

Best Practices:

- **Agile Development:** Work in sprints (typically 2-4 weeks), delivering potentially shippable increments.
- **Version Control (Git):** Use Git for source code management. Employ a defined branching strategy (e.g., Gitflow, GitHub Flow) with Pull Requests (PRs) for code integration.
- **Secure Coding Practices:** Adhere strictly to secure coding standards (OWASP Top 10, NYCPS Standards). Sanitize all inputs, validate outputs, handle errors securely, implement proper authentication/authorization checks.
- **Test-Driven Development (TDD) / Behavior-Driven Development (BDD):** Write unit tests *before* or *alongside* feature code. Use BDD frameworks (e.g., Cucumber, SpecFlow) to define acceptance criteria as executable tests.

- **Code Reviews:** Mandatory peer code reviews for all PRs, focusing on logic, security, performance, maintainability, and adherence to standards. Automated static analysis (SAST) integrated into the review process.
- **Dependency Management & Security (SCA):** Use package managers (npm, Maven, pip, etc.). Regularly scan dependencies for known vulnerabilities (using tools like Snyk, OWASP Dependency-Check, GitHub Dependabot) and update/patch promptly.
- **Containerization (Docker):** Package applications and dependencies into Docker containers for consistency across environments.
- **Infrastructure as Code (IaC) Development:** Develop and version control IaC templates/scripts (CloudFormation/CDK/Terraform) alongside application code.
- **Continuous Integration (CI):** Automate builds, unit tests, SAST scans, and packaging upon every code commit/PR using the CI/CD pipeline.

Tool Examples: *IDEs (VS Code, IntelliJ), Git (GitHub/GitLab/CodeCommit), Docker, JUnit/NUnit/PyTest (Unit Testing), SonarQube/Checkmarx (SAST), Snyk/Dependency-Check (SCA), AWS CodeBuild/Jenkins/GitLab CI.*

Phase 4: Testing & Quality Assurance

Goal: Verify system functionality, performance, security, usability, and compliance through rigorous, multi-layered testing integrated throughout the SDLC.

Best Practices:

- **Multi-Layered Test Strategy:** Implement a comprehensive strategy including:
    - **Unit Testing:** Developer-written tests verifying individual code units/functions (automated in CI).

    - **Integration Testing:** Verifying interactions between components/microservices (partially automated).
    - **Component/API Testing:** Testing individual microservice APIs directly (automated).
    - **End-to-End (E2E) Testing:** Testing complete user flows across multiple components/UI (automated where feasible).
    - **User Acceptance Testing (UAT):** Formal testing by NYCPS stakeholders/end-users against defined acceptance criteria.
    - **Performance & Load Testing:** Simulating expected and peak user loads to measure response times, throughput, and resource utilization against NFRs.
    - **Security Testing:** Dynamic Application Security Testing (DAST), Interactive Application Security Testing (IAST), manual penetration testing (internal & third-party).
    - **Accessibility Testing:** Automated and manual testing against WCAG 2.0 AA standards.
    - **Regression Testing:** Re-running relevant tests after code changes/bug fixes to prevent regressions (heavily automated).
    - **Disaster Recovery Testing:** Periodic testing of failover and recovery procedures.
- **Test Automation:** Automate tests at all levels where practical (Unit, API, E2E, Regression, Performance) and integrate into the CI/CD pipeline.
- **Dedicated Test Environments:** Maintain separate, stable environments for different testing phases (e.g., Dev Integration, QA, Staging/UAT, Performance). Provision using IaC.
- **Test Data Management:** Use anonymized or synthetically generated data for testing, especially in lower environments, to protect PII.

- **Defect Tracking:** Use a bug tracking system (e.g., Jira) to log, prioritize, assign, and track defects through resolution.
- **Testing as part of DoD:** Ensure all relevant tests (unit, integration, acceptance criteria) pass before a user story is considered "Done".

Tool Examples: *JUnit/NUnit/PyTest (Unit), Postman/RestAssured (API), Selenium/Cypress/Playwright (E2E), JMeter/k6/Gatling (Load), OWASP ZAP/Burp Suite (Security), Axe/WAVE (Accessibility), Jira/Azure DevOps (Defect Tracking).*

Phase 5: Deployment & Release Management

Goal: Reliably, securely, and efficiently deploy tested code and infrastructure changes to production environments with minimal downtime and risk.

Best Practices:

- **Continuous Deployment (CD):** Automate the deployment process through the CI/CD pipeline, triggered after successful testing phases.
- **Infrastructure as Code (IaC):** Provision and manage all environments (Dev, Test, Prod) using IaC scripts/templates versioned in Git.
- **Environment Parity:** Strive for maximum consistency between Staging/UAT and Production environments.
- **Deployment Strategies:** Utilize strategies like Blue/Green deployments or Canary Releases to minimize deployment risk and allow for easy rollback.
- **Automated Rollbacks:** Implement automated rollback mechanisms in the CD pipeline triggered by deployment failures or critical monitoring alerts post-deployment.

- **Configuration Management:** Manage environment-specific configurations securely (e.g., using AWS Systems Manager Parameter Store, Secrets Manager, or dedicated configuration management tools). Avoid hardcoding configurations.
- **Release Management Process:** Define a clear process for scheduling, approving, and communicating production releases, coordinating between Dev, Ops, Security, and NYCPS stakeholders.
- **Smoke Testing:** Perform automated smoke tests immediately after deployment to verify critical functionality is working in production.
- **Zero Downtime Deployments:** Architect applications and deployment processes to minimize or eliminate user-facing downtime during releases where feasible.

Tool Examples: *AWS CodeDeploy, Jenkins/GitLab CI/GitHub Actions, Terraform/CloudFormation/CDK, AWS Systems Manager Parameter Store/Secrets Manager, Docker, Kubernetes/ECS/Fargate.*

Phase 6: Operations & Maintenance

Goal: Ensure the ongoing stability, performance, security, and availability of the production system, respond to incidents, and implement continuous improvements.

Best Practices:

- **Monitoring & Alerting:** Implement comprehensive monitoring (using CloudWatch, APM tools) covering infrastructure health, application performance (latency, error rates), security events, and key business metrics. Configure actionable alerts for anomalies or threshold breaches.
- **Log Aggregation & Analysis:** Centralize application and infrastructure logs (e.g., using CloudWatch Logs Insights, ELK stack, Splunk). Enable searching, analysis, and dashboarding for troubleshooting and auditing.

- **Incident Management:** Establish a clear process for incident detection, response, escalation (aligned with SLAs), root cause analysis (RCA), and post-mortem reviews. Utilize tools for on-call scheduling and incident tracking.
- **Patch Management:** Regularly apply OS, runtime, and dependency security patches following a defined testing and rollout process. Automate where possible (e.g., using AWS Systems Manager Patch Manager).
- **Backup & Recovery:** Regularly execute and test backup procedures for databases and critical data on S3. Ensure RPO/RTO targets are met.
- **Disaster Recovery (DR) Drills:** Conduct periodic DR tests (at least annually) to validate the DR plan and recovery procedures.
- **Performance Optimization:** Continuously monitor performance metrics and proactively optimize resource utilization, database queries, caching, etc.
- **Security Monitoring & Auditing:** Regularly review security logs, alerts (GuardDuty, Security Hub), and compliance status (Config). Conduct periodic vulnerability scans and security audits.
- **Cost Optimization:** Monitor AWS costs using Cost Explorer and other tools. Implement cost optimization strategies (e.g., Reserved Instances, Savings Plans, S3 lifecycle policies, right-sizing resources).
- **Feedback Loop:** Feed operational insights, performance data, and incident RCAs back into the Planning and Design phases for continuous improvement.

Tool Examples: *AWS CloudWatch, AWS Config, AWS CloudTrail, Security Hub, GuardDuty, Systems Manager, AWS Backup, PagerDuty/Opsgenie (Incident Mgmt), ELK Stack/Splunk/Datadog (Logging/APM), Grafana/QuickSight (Dashboards).*

SDLC Setup: Step-by-Step Instructions

Setting up this DevSecOps-focused SDLC involves configuring tools, defining processes, and establishing workflows. This requires collaboration between Development, Operations, Security, and Project Management teams.

1. **Establish Foundational AWS Environment (Ref: Phase 0):**
    - 2 Provision separate AWS GovCloud accounts for Development, Testing (QA/Staging/UAT/Perf), and Production.
    - 3 Implement baseline networking (VPCs, subnets, security groups, NACLs, endpoints, Direct Connect/VPN) using IaC (e.g., Terraform/CloudFormation).
    - 4 Configure core IAM roles, groups, policies, and MFA enforcement. Set up federation if possible.
    - 5 Enable and configure core security services: CloudTrail (central logging bucket), Config, GuardDuty, Security Hub.
6. **Select & Configure Toolchain:**
    - 7 **Version Control:** Set up Git repositories (e.g., AWS CodeCommit, GitHub, GitLab) with branch protection rules. Define branching strategy (e.g., Gitflow).
    - 8 **Project Management/Backlog:** Configure Jira/Azure DevOps project(s) with boards (Scrum/Kanban), workflows, issue types, and reporting dashboards. Integrate with Git repository.
    - 9 **CI/CD Platform:** Configure AWS CodePipeline/CodeBuild/CodeDeploy or alternative (Jenkins, GitLab CI). Set up build agents/runners. Integrate with source control.
    - 10 **Collaboration:** Set up communication channels (Slack/Teams), documentation platform (Confluence/SharePoint).
    - 11 **Artifact Repository:** Configure Docker image repository (ECR), package repositories (CodeArtifact, Nexus, Artifactory) if needed.
12. **Define Workflows & Processes:**

- 13 **Code Review Process:** Document requirements for Pull Requests (description, linked issues, testing evidence), required reviewers, approval process.
- 14 **Testing Strategy:** Define types of tests required for different components, environments for each test type, test data requirements, and quality gates within the CI/CD pipeline. Document DoD to include testing criteria.
- 15 **Deployment Process:** Define promotion criteria between environments (Dev -> Test -> Staging -> Prod), approval workflows, rollback procedures, release scheduling/communication plan.
- 16 **IaC Workflow:** Define process for developing, testing, reviewing, and applying IaC changes.
- 17 **Agile Ceremonies Cadence:** Schedule regular sprint planning, stand-ups, reviews, retrospectives, and backlog grooming sessions.

18 **Integrate Security into CI/CD ("Sec" in DevSecOps):**

- 19 Integrate SAST tools (e.g., SonarQube, Checkmarx) into CodeBuild/Jenkins stage to scan code on commit/PR. Fail builds on critical/high vulnerabilities.
- 20 Integrate SCA tools (e.g., Snyk, OWASP Dependency-Check) to scan dependencies. Fail builds on critical/high vulnerabilities with no available patch.
- 21 Integrate DAST tools (e.g., OWASP ZAP) into testing stages against deployed applications in test environments.
- 22 Configure pipeline IAM roles with least privilege.
- 23 Secure sensitive credentials/keys used by the pipeline (e.g., using Secrets Manager).

24 **Set Up Monitoring & Logging Infrastructure:**

- 25 Ensure CloudWatch Agent is configured/deployed to EC2 instances (if used) and containerized applications (e.g., via sidecar or daemonset) to collect detailed metrics and logs.
- 26 Configure centralized logging aggregation (e.g., CloudWatch Logs Insights, or streaming to ELK/Splunk if preferred).
- 27 Set up baseline CloudWatch dashboards and alarms for key infrastructure and application metrics in test and prod environments.
- 28 Configure APM tool integration if used.

29 **Provision Initial Environments via IaC:**

- 30 Use defined IaC templates/scripts to provision the initial Development and Testing environments, ensuring consistency.

31 **Team Onboarding & Training:**

- 32 Onboard project team members to tools and processes.
- 33 Conduct training on secure coding practices, AWS GovCloud specifics, CI/CD pipeline usage, and defined workflows.
- 34 Ensure understanding of compliance requirements (FERPA, NY Ed Law 2-d, etc.).

35 **Establish Governance & Continuous Improvement:**

- 36 Set up regular architecture review meetings.
- 37 Use sprint retrospectives to identify process bottlenecks or areas for improvement in the SDLC itself.
- 38 Periodically review and update security standards, tool configurations, and workflows.