# NYCPS TMS: Prescriptive Development Strategy & Implementation Guide

## Introduction: Development Philosophy & Methodology

This document prescribes the definitive end-to-end Development Strategy for the NYCPS Transportation Management System (TMS). Our approach is grounded in modern software engineering principles, emphasizing agility, security, quality, and automation to successfully deliver this complex, large-scale system on AWS GovCloud within the required timeframe. We will employ an **Agile (Scrum) methodology** tightly integrated with **DevSecOps practices**.

A core tenet of this strategy is **developer ownership of quality**. Developers are responsible for writing comprehensive automated tests (unit, integration, E2E) for their code. The dedicated QA team will focus on exploratory testing, UAT facilitation, performance testing, and security validation, complementing, but not replacing, the developer's responsibility for functional correctness verified through automation.

Adherence to the processes, standards, and tools outlined herein is mandatory for all development team members to ensure consistency, security, and predictability throughout the project lifecycle.

# I. Development Environment Setup & Tooling

We will establish a standardized, robust development environment for every engineer to ensure consistency and productivity. This environment provides the necessary tools to build, test, and debug code locally before integrating it into the shared pipelines.

## A. Developer Workstation Specifications (Minimum)

> *These specifications ensure developers can run necessary tools like Docker, IDEs, and potentially local simulators without performance bottlenecks.*

- **Operating System:** macOS (latest or N-1) or Linux (e.g., Ubuntu LTS). Windows 10/11 with WSL2 is acceptable but macOS/Linux preferred for CLI/Docker consistency.
- **Processor:** Modern multi-core processor (e.g., Intel Core i7/i9 8th gen+, AMD Ryzen 7+, Apple M1/M2/M3).
- **RAM:** Minimum 16GB (32GB strongly recommended).
- **Storage:** Minimum 512GB NVMe SSD (1TB recommended).

- **Display:** Sufficient resolution for comfortable IDE usage (e.g., FHD or higher). Dual monitors recommended.

## B. Required Software Installation

Developers must install and configure the following core software:

1. **Version Control:** Git (latest stable version). Configuration must include setting user name and email consistent with the project's Git host (e.g., GitHub/GitLab/CodeCommit).

2. **Containerization:** Docker Desktop (latest stable version) or equivalent container runtime. Ensure sufficient resources are allocated in Docker settings (CPU, RAM).

3. **Cloud CLI:** AWS CLI v2 (latest version). Must be configured for AWS GovCloud regions (\`us-gov-west-1\`, \`us-gov-east-1\`) using secure credential methods (e.g., SSO integration via AWS IAM Identity Center, AssumeRole credentials via profile, *never* hardcoded access keys). MFA must be enforced for console/API access.

4. **Infrastructure as Code CLI:** Terraform CLI (latest stable version matching the project standard).

5. **Runtime Environments:** Install specific versions required by the project's chosen backend/frontend stacks (e.g., Node.js LTS, Python 3.11+, JDK 17+). Use version managers (nvm, pyenv, SDKMAN!) where applicable for consistency.

6. **Package Managers:** Install necessary package managers (npm/yarn, pip, Maven/Gradle).

7. **Build Tools:** Ensure relevant build tools are available (e.g., Maven, Gradle, Webpack/Vite CLI).

## C. Integrated Development Environments (IDEs)

Developers will use modern IDEs equipped with appropriate plugins for enhanced productivity and quality.

- **Recommended IDEs:**
  - **Frontend:** VS Code (recommended), WebStorm.
  - **Backend (Java):** IntelliJ IDEA Ultimate.
  - **Backend (Python):** PyCharm Professional, VS Code.
  - **Backend (Node.js):** VS Code, WebStorm.
- **Mandatory IDE Plugins/Configuration:**
  - **Language Support:** Official plugins for the project's primary languages (Python, Java, TypeScript/JavaScript, etc.).
  - **Linters/Formatters:** Integration with project-defined linters/formatters (e.g., ESLint, Prettier, Black, Checkstyle) with auto-format-on-save enabled.
  - **Debugger:** Configured for local debugging of applications (including Docker containers).
  - **Git Integration:** Built-in or plugin-based Git client.
  - **Docker Integration:** Plugin for managing Docker containers and images.
  - **AWS Toolkit:** Official AWS Toolkit plugin for IDE integration with AWS services (Lambda testing, ECR browsing, etc.).

        - **Terraform Plugin:** Syntax highlighting, validation, and auto-completion for Terraform HCL.
        - **Testing Framework Integration:** Plugins to run and debug unit/integration tests directly within the IDE (e.g., Jest Runner, Pytest plugin).

## D. Local AWS Simulation (Optional but Recommended)

- **Tooling:** Utilize tools like `LocalStack` or specific service simulators (e.g., `DynamoDB Local`) to emulate AWS services locally. This accelerates development cycles by reducing reliance on deployed cloud environments for basic testing.
- **Configuration:** Configure application code and test frameworks to target local endpoints when running in a local development/test mode.

## E. Access Provisioning & Onboarding

1. **AWS Access:** Provision developer IAM users/roles with least-privilege access to necessary GovCloud DEV environment services via federation or IAM Identity Center. MFA must be enforced.
2. **Tool Access:** Grant access to Git repositories, Jira/ADO project boards, Confluence/SharePoint spaces, CI/CD platform, artifact repositories.
3. **Onboarding Documentation:** Provide developers with a comprehensive onboarding guide covering project setup, tool configuration, workflow procedures, and points of contact.

# II. Coding Standards & Secure Practices

We will enforce rigorous coding standards and secure development practices across all codebases (frontend, backend, infrastructure) to ensure quality, maintainability, readability, and security.

## A. Language-Specific Style Guides

- **Mandate:** All code must adhere to widely accepted style guides for the respective language.
    - **JavaScript/TypeScript:** Airbnb JavaScript Style Guide or Google TypeScript Style Guide (TBD based on team preference).
    - **Python:** PEP 8 (Style Guide for Python Code).
    - **Java:** Google Java Style Guide.
    - *(Add others as needed)*
- **Enforcement:** Utilize automated linters (ESLint, Flake8/Pylint, Checkstyle) and formatters (Prettier, Black, google-java-format) configured according to the chosen style guide. These tools will be integrated into IDEs (auto-format on save) and the CI pipeline (build fails on linting errors).

## B. General Coding Principles

- **Readability:** Write clear, concise, self-documenting code. Use meaningful variable and function names. Keep functions/methods short and focused (Single Responsibility Principle).

- **Modularity:** Design code in reusable, loosely coupled modules/components/classes (SOLID principles). Avoid monolithic structures.
- **Comments:** Comment complex logic, non-obvious decisions, and public APIs/interfaces. Avoid commenting obvious code. Keep comments up-to-date.
- **Error Handling:** Implement robust error handling using exceptions or appropriate error-signaling mechanisms. Log errors effectively (see Accountability section in SDLC). Avoid swallowing exceptions silently. Provide meaningful error messages for debugging without exposing sensitive internal details to end-users.
- **Configuration Management:** Externalize all environment-specific configurations (database connections, API endpoints, feature flags). Never hardcode configurations or secrets in the codebase. Use environment variables, configuration files loaded at runtime, or secrets management services.

## C. Secure Coding Practices (Mandatory)

This is paramount. All developers must understand and apply secure coding principles based on OWASP Top 10, SANS Top 25, and the specific **NYCPS Secure Coding Standards for Vendors (Attachment 1B)**.

- **Input Validation:** Rigorously validate *all* input from external sources (user interfaces, APIs, files, databases, other services) on the server-side. Use allow-listing (whitelisting) rather than deny-listing (blacklisting). Validate for type, length, format, and range. Sanitize data intended for interpretation by other systems (e.g., SQL, OS commands, XML/JSON parsers).

- **Output Encoding:** Contextually encode all output displayed to users or passed to interpreters (HTML, JavaScript, SQL) to prevent Cross-Site Scripting (XSS) and other injection attacks. Use framework-provided encoding mechanisms where available and appropriate for the context.
- **Authentication & Session Management:** Implement securely. Do not expose session IDs in URLs. Use secure, HttpOnly, SameSite cookies. Regenerate session IDs upon login/privilege change. Implement proper logout functionality that invalidates the session server-side. Protect against session fixation and hijacking.
- **Access Control (Authorization):** Enforce authorization checks server-side for *every* request accessing protected resources or performing sensitive actions. Implement checks based on user roles/permissions derived from a trusted source (e.g., validated JWT claims, session data linked to backend authorization system). Adhere to the principle of least privilege. Protect against Insecure Direct Object References (IDOR).
- **Cryptographic Practices:** Use strong, industry-standard cryptographic algorithms and libraries for encryption and hashing (as mandated by NYCPS/NYC3 standards and AWS KMS/Secrets Manager capabilities). Never store sensitive data (especially PII) unencrypted at rest unless absolutely necessary and approved with compensating controls. Never implement custom cryptographic algorithms. Manage keys securely using KMS.
- **Error Handling & Logging:** Do not leak sensitive information (stack traces, system details, PII) in error messages presented to users. Log security-relevant events (logins, failures, access control decisions, exceptions) with sufficient detail for auditing and forensics (see Accountability).
- **Dependency Security:** Regularly scan third-party libraries and dependencies for known vulnerabilities (SCA). Update vulnerable dependencies promptly.

- **Database Security:** Use parameterized queries or prepared statements exclusively to prevent SQL injection. Grant database users least privilege access.
- **File Handling:** Validate filenames and paths to prevent directory traversal. Scan uploaded files for malware. Store user-uploaded files outside the web root with appropriate permissions.

> *Regular secure coding training (e.g., based on OWASP resources) will be mandatory for all developers. Security champions within teams will promote best practices.*

## III. Development Workflow (Git & Agile/Scrum)

We will follow a structured Git workflow integrated with our Scrum process to manage code changes, collaboration, and releases effectively.

### A. Git Branching Strategy (Gitflow Variant)

-
  ```
  main
  ```
  : Represents the production-ready code. Only receives merges from approved `release` or `hotfix` branches. Protected branch.

-
  ```
  develop
  ```

: Represents the latest integrated development state. Feature branches are merged here. Protected branch (requires PRs). Nightly/regular builds deployed to DEV/QA environment from here.

-
  ```
  feature/TICKET-###-short-description
  ```

  : Developers create feature branches from `develop` for each user story or bug fix task (where `TICKET-###` is the Jira/ADO issue key).

-
  ```
  release/vX.Y.Z
  ```

  : Branched from `develop` when preparing for a production release. Used for final stabilization, bug fixing, and documentation updates specific to the release. Merged into `main` upon release, and back into `develop` to incorporate release-specific fixes.

-
  ```
  hotfix/TICKET-###-critical-fix
  ```

  : Branched directly from `main` to address critical production bugs. Merged back into both `main` and `develop` upon completion.

## B. Agile/Scrum Process Integration

1. **Sprint Planning:** The team selects user stories from the prioritized Product Backlog that fit within the sprint capacity (typically 2 weeks). Stories are broken down into technical tasks. A sprint goal is defined.
2. **Development Cycle:**
   - For each task/story, a developer creates a `feature/` branch from `develop`.
   - Code is written following standards, including comprehensive automated tests (unit, integration).

- Code is committed frequently to the feature branch with meaningful messages referencing the ticket ID.
- Developer ensures all tests pass locally and code lints correctly.

3. **Pull Request (PR) Process:**
    - Developer creates a PR from their `feature/` branch targeting `develop`.
    - PR description links to the Jira/ADO ticket(s), summarizes changes, and details any manual testing performed or specific review instructions.
    - CI pipeline automatically runs (build, lint, unit tests, SAST, SCA). PR cannot be merged if CI fails.
    - Code review is performed (see Section VII). Reviewers approve or request changes.
    - Developer addresses feedback, pushes updates to the feature branch (triggering CI again).
    - Once approved and CI passes, the PR is merged into `develop` (typically squashed for cleaner history).
4. **Daily Stand-up:** Team members briefly share progress, plans for the day, and any blockers.
5. **Testing (Post-Merge):** Automated integration and E2E tests run against the `develop` branch build deployed to the QA environment.
6. **Sprint Review:** Team demonstrates the completed and tested features (meeting Definition of Done) from the sprint to the Product Owner and stakeholders. Feedback is gathered.

7. **Sprint Retrospective:** Team discusses what went well, what could be improved, and agrees on process adjustments for the next sprint.

## IV. Frontend Development Strategy (Web & Mobile)

We will develop intuitive, accessible, performant, and secure user interfaces for all modules (Parent/Caregiver, Student, Driver, School Admin, OPT Admin, SBC Admin) using modern web and mobile technologies.

### A. Technology Stack (Example: React & React Native)

*The specific framework choice (React, Angular, Vue for web; React Native, Native Swift/Kotlin, Flutter for mobile) will be finalized during detailed design, but the principles below apply generally. This example assumes React for web and React Native for mobile for consistency.*

- **Web Framework:** React (latest stable version) with TypeScript.
- **Mobile Framework:** React Native with TypeScript.
- **State Management:** Redux Toolkit or Zustand (TBD based on complexity) for global state; React Context API for localized state.
- **Component Library/UI Kit:** Utilize a pre-built, accessible component library (e.g., Material UI, Ant Design, Chakra UI) customized to NYCPS

branding/style guide, or develop a custom library.

- **Routing:** React Router (Web), React Navigation (Mobile).
- **API Interaction:** Axios or Fetch API, potentially with generated clients from OpenAPI specs (e.g., using `openapi-generator`). Implement robust error handling and loading states.
- **Mapping:** Integrate mapping libraries (e.g., Leaflet, Mapbox GL JS, Google Maps SDK wrappers) compatible with backend GIS data (potentially GeoJSON served via APIs) and AWS Location Service (if used).
- **Build Tool:** Vite or Create React App (managed via react-scripts) for web; Metro bundler for React Native.

## B. Development Practices

- **Component-Based Architecture:** Build UIs using small, reusable, well-defined components with clear props and state management.
- **TypeScript Enforcement:** Utilize TypeScript for static typing to catch errors early and improve code maintainability. Enforce strict type checking.
- **State Management Strategy:** Clearly define where state resides (local component state, context, global store) to avoid prop drilling and ensure predictability.
- **API Integration:** Develop dedicated service modules/hooks for interacting with backend APIs. Implement consistent error handling, request cancellation, and caching strategies (e.g., using React Query or SWR).
- **Accessibility (WCAG 2.0 AA):** Mandatory adherence. Use semantic HTML, ARIA attributes where necessary, ensure keyboard navigability,

sufficient color contrast, provide text alternatives for non-text content. Integrate accessibility linters (e.g., `eslint-plugin-jsx-a11y`) and perform manual testing (keyboard, screen reader).

- **Performance Optimization:** Code splitting, lazy loading components/routes, image optimization, memoization (React.memo, useMemo, useCallback), efficient state updates, network request optimization.
- **Security:** Sanitize any user-generated content displayed in the UI (though primary sanitization is backend). Protect against client-side vulnerabilities like XSS (via framework mechanisms and output encoding), CSRF (if using session cookies - less common with token auth), insecure data storage (use secure storage mechanisms like SecureStore/Keychain on mobile).

## C. Developer-Owned Testing (Frontend)

- **Unit Tests:** Test individual components and utility functions in isolation using Jest and React Testing Library (RTL). Mock API calls and external dependencies. Focus on rendering output, state changes, and event handling logic. Aim for high coverage of component logic.
- **Integration/Component Tests:** Test interactions between related components, context providers, or components interacting with mocked service layers using RTL or potentially Cypress Component Testing.
- **End-to-End (E2E) Tests:** Automate critical user flows using Cypress or Playwright. Tests run against a deployed application in a test environment interacting with live (or mocked) backend APIs. Examples: User login, viewing bus location, submitting absence report, scanning QR code (simulated).

- **Accessibility Checks:** Integrate automated accessibility checks (e.g., `jest-axe`, `cypress-axe`) into test suites.
- **CI Integration:** All unit and component tests run on every PR/commit. E2E tests run against deployments to QA/Staging environments.

## V. Backend Development Strategy (Microservices & API)

We will develop backend functionality as a set of independent, scalable, and resilient microservices communicating via well-defined APIs and asynchronous events, deployed primarily as containers on AWS Fargate/ECS or serverless functions via Lambda.

### A. Technology Stack (Example: Python/FastAPI & Node.js/Express)

> *The choice of language/framework per microservice can vary based on the specific task (e.g., Python for data processing/routing, Node.js for I/O-bound API gateways). Consistency within a domain is encouraged. This example uses Python/FastAPI for a core service and Node.js for an API aggregation layer.*

- **Primary Language/Framework:** Python 3.11+ with FastAPI (for performance and type hints) or Node.js LTS with Express/NestJS (for I/O heavy tasks). Java/Spring Boot is also a viable option. (Decision per service team/domain).

- **API Specification:** OpenAPI 3.0 (Swagger) for defining all RESTful APIs. Use code-first (FastAPI/NestJS) or spec-first approaches.
- **Data Access:**
    - **PostgreSQL/PostGIS:** SQLAlchemy (Python), TypeORM/Prisma (Node.js), JPA/Hibernate (Java).
    - **DynamoDB:** AWS SDK (Boto3 for Python, AWS SDK for JavaScript/Java).
    - **ElastiCache (Redis/Memcached):** Standard client libraries for the chosen language.
- **Messaging/Streaming:** AWS SDK for interacting with Kinesis, SQS, SNS. Libraries like `aiobotocore` (Python) or `@aws-sdk/client-sqs` (Node.js) for asynchronous operations.
- **Containerization:** Docker with multi-stage builds for optimized, secure images based on official language base images.

## B. Development Practices

- **API Design (RESTful):** Follow REST principles. Use standard HTTP methods (GET, POST, PUT, PATCH, DELETE) appropriately. Use clear resource naming conventions. Implement versioning (e.g., `/v1/` in URL path).
- **Microservice Design:** Design services around business capabilities (DDD). Ensure loose coupling and high cohesion. Communicate via APIs or events, avoid direct database sharing between services. Implement resilience patterns (retries, timeouts, circuit breakers) for inter-service calls (e.g., using libraries like `resilience4j` (Java), `polly` (.NET via Lambda Powertools), or custom logic).

- **Asynchronous Processing:** Utilize message queues (SQS) and pub/sub (SNS) for decoupling time-consuming tasks or broadcasting events. Design services to be idempotent where possible when consuming messages.
- **Configuration Management:** Load configurations (database URLs, API keys, feature flags) from environment variables or AWS Systems Manager Parameter Store / Secrets Manager at startup.
- **Logging:** Implement structured logging (JSON format preferred) including correlation IDs to trace requests across microservices. Log key events, errors, and security-relevant actions.
- **Security Implementation:**
    - Implement authentication/authorization middleware in API gateways or individual services to validate JWTs/tokens and check permissions based on roles/claims.
    - Apply input validation rigorously at API boundaries.
    - Use parameterized queries/ORMs to prevent SQL injection.
    - Minimize attack surface; expose only necessary endpoints.

## C. Developer-Owned Testing (Backend)

- **Unit Tests:** Test individual functions, classes, controllers in isolation using frameworks like Pytest (Python), JUnit/Mockito (Java), Jest (Node.js). Mock external dependencies (database calls, API calls to other services, message queues). Focus on business logic, validation rules, error handling. Aim for high code coverage.

- **Integration Tests:** Test service interaction with its direct dependencies (e.g., database, cache, message queue) within a controlled test environment. Use test containers (e.g., `testcontainers`) locally or run against dedicated test instances in the cloud QA environment. Verify data persistence, message consumption/production, interactions with mocked external APIs.
- **API/Contract Tests:** Test API endpoints directly using tools like Postman or automated frameworks (Pytest with `requests`, RestAssured). Validate request/response schemas against OpenAPI specs. Consider using Pact for consumer-driven contract testing between microservices to ensure compatibility without full E2E setup.
- **E2E Tests (Service Level):** Test critical business flows that span multiple microservices owned by the team, typically by invoking the public API gateway and verifying results/side effects across services (e.g., creating a route triggers DB updates and notifications). Mock external dependencies beyond the team's control.
- **CI Integration:** All unit and integration tests run on every PR/commit. API/Contract/E2E tests run against deployments to QA environment.

## VI. Integration Strategy

We will ensure seamless and secure data flow between internal microservices, NYCPS enterprise systems, and required third-party services.

### A. Internal Microservice Communication

- **Synchronous (Request/Response):** Primarily via RESTful APIs exposed through internal API Gateway endpoints or service discovery mechanisms within ECS/EKS. Use HTTPS (TLS) for all internal traffic. Implement appropriate authentication/authorization between services (e.g., mutual TLS, service-to-service JWTs).
- **Asynchronous (Event-Driven):** Utilize SNS for pub/sub (broadcasting events like 'RouteUpdated') and SQS for reliable queuing of commands or events requiring specific processing (e.g., 'ProcessNewRidershipScan'). Leverage Kinesis/MSK for high-volume event streams (GPS data). Design consumers to be idempotent.

## B. NYCPS System Integration (ATS, NPSIS, Ticketing, Contracts, etc.)

- **Discovery & Definition:** Collaborate closely with DIIT and NYCPS SMEs to precisely define data requirements, formats, frequency (real-time API vs. batch file), and security protocols for each integration point.
- **API-Based Integration:** Prefer RESTful APIs where available from NYCPS systems. Develop dedicated 'Adapter' microservices within the TMS architecture to handle communication, data transformation, and error handling specific to each NYCPS system API. Secure communication using appropriate authentication (API keys, OAuth, mTLS) managed via Secrets Manager and ensure transport encryption (TLS).
- **File-Based Integration:** If APIs are unavailable, establish secure file transfer mechanisms (SFTP preferred over FTP) using dedicated S3 buckets and Lambda/Glue ETL jobs triggered by file arrival events (S3 Event Notifications). Define file formats, naming conventions, PGP encryption requirements, and error handling processes. Utilize AWS Transfer Family for managed SFTP endpoints.

- **Database Integration (Least Preferred):** Direct database links are generally discouraged due to tight coupling and security risks. If absolutely necessary and approved by security/architecture review, establish read-only replicas or use secure database links with tightly restricted permissions, accessed only via dedicated adapter services.
- **Security & Connectivity:** Utilize the established secure AWS Direct Connect/VPN tunnels for connectivity to internal NYCPS resources. Implement appropriate firewall rules and security group configurations on both ends.

## C. Third-Party Service Integration (Traffic Data, Mapping/GIS, Notifications)

- **API Keys & Credentials:** Store all third-party API keys and credentials securely in AWS Secrets Manager. Grant access only to specific IAM roles used by the services needing them.
- **SDK/Client Libraries:** Use official vendor-provided SDKs or well-vetted client libraries where available.
- **Resilience:** Implement retry logic with exponential backoff and circuit breakers for calls to external services to handle transient failures or rate limiting.
- **Rate Limiting:** Be mindful of and respect any rate limits imposed by third-party APIs. Implement client-side throttling if necessary.
- **Error Handling:** Implement robust error handling for failed API calls, timeouts, or unexpected responses from third parties.

# VII. Testing Strategy (Developer Ownership)

Quality is a team responsibility, with developers taking primary ownership of ensuring functional correctness through comprehensive, automated testing integrated deeply into the development workflow.

## A. Developer Testing Responsibilities

- **Unit Testing:** Developers *must* write unit tests covering the logic within their code (functions, classes, components). Mocks/stubs will be used to isolate the unit under test. Goal: High code coverage verified by tools (e.g., `coverage.py`, Jacoco, Istanbul) and validation during code review.
- **Integration Testing:** Developers *must* write integration tests verifying the interaction of their service with its direct, essential dependencies (e.g., database persistence layer, message queue consumers/producers, interactions with *closely coupled* internal services). These tests may run against local test containers or dedicated QA environment resources.
- **API/Contract Testing:** Developers *must* write tests that validate the API contract (request/response structure, status codes, basic auth/authz) of the services they build. Tools like Postman (collections run via Newman in CI) or framework-specific test clients will be used. Pact contract testing is encouraged between highly interdependent services.

- **End-to-End (E2E) Testing Contribution:** Developers will collaborate with QA engineers to identify critical user flows and contribute to building/maintaining automated E2E test suites (e.g., using Cypress/Playwright). While QA may take the lead on the overall E2E suite, developers are responsible for ensuring their features are testable and contributing tests for core paths.
- **Test Automation:** All unit, integration, and API/contract tests *must* be automated and integrated into the CI pipeline. E2E tests should be automated for critical paths and run regularly.

## B. Definition of Done (DoD)

A User Story is considered "Done" only when:

- Code implementing the functionality is complete.
- Code adheres to all coding and security standards.
- Comprehensive unit and integration tests are written and *passing* (meeting coverage targets).
- Relevant API/Contract tests are written and *passing*.
- E2E test scenarios (if applicable) are identified, and automated tests are created/updated and *passing*.
- Code has been successfully peer-reviewed and merged into the `develop` branch.
- The feature has been successfully deployed and verified in the QA environment via automated tests.
- Necessary documentation (code comments, API spec updates, user guides if applicable) is complete.

- The Product Owner has formally accepted the story during the Sprint Review based on meeting acceptance criteria.

**Failure to include adequate automated tests (Unit, Integration, API) will block code reviews and prevent stories from meeting the Definition of Done.**

## C. QA Team Role

The QA team complements developer testing by focusing on:

- **Exploratory Testing:** Manually exploring the application to find edge cases, usability issues, and unexpected behavior missed by automated tests.
- **UAT Facilitation:** Organizing and supporting UAT sessions with NYCPS stakeholders.
- **Performance Testing:** Designing, executing, and analyzing results from large-scale performance and load tests.
- **Security Testing Oversight:** Coordinating penetration tests, reviewing DAST/SAST results, and validating security fixes.
- **E2E Test Suite Management:** Owning the overall E2E test automation strategy, framework, and execution, collaborating with developers.
- **Test Environment Management:** Overseeing the setup and maintenance of dedicated QA/Staging/Performance environments.
- **Release Certification:** Providing a final quality assessment before production releases based on overall test results and risk analysis.

# VIII. Code Review Process

Mandatory peer code reviews are a critical quality gate. All code changes merged into `develop` (and subsequently `main` via releases/hotfixes) must undergo a thorough review.

## A. Process Steps

1. Developer completes work on a feature branch, ensuring all local tests pass and code adheres to standards.
2. Developer creates a Pull Request (PR) targeting the `develop` branch.
3. PR description clearly explains the purpose of the change, links to the relevant Jira/ADO ticket(s), and highlights any areas needing specific attention or manual testing steps.
4. Automated CI checks (build, lint, unit tests, SAST, SCA) are triggered and must pass.
5. Developer assigns 1-2 designated peer reviewers (or team lead for critical changes).
6. Reviewer(s) examine the code against a defined checklist (see below).
7. Reviewer(s) provide constructive feedback via PR comments or approve the changes.
8. Developer addresses all comments, pushing updates to the feature branch (re-triggering CI).
9. Once all reviewers approve and CI checks pass, the PR is merged (preferably using squash merge for clean history).

## B. Review Checklist (Minimum Criteria)

- **Functionality:** Does the code correctly implement the requirements/acceptance criteria? Does it handle edge cases?
- **Security:** Are there potential vulnerabilities (OWASP Top 10)? Is input validated? Is output encoded? Are auth checks correct? Is sensitive data handled appropriately?
- **Testing:** Are there sufficient, meaningful unit and integration tests? Do tests cover main paths and edge cases? Is coverage adequate?
- **Readability/Maintainability:** Is the code clear, well-formatted, and understandable? Are names meaningful? Are comments adequate? Is complexity minimized? Does it follow SOLID/modular principles?
- **Performance:** Are there obvious performance bottlenecks (e.g., inefficient loops, unnecessary database calls)? Is resource handling (e.g., connections, file handles) correct?
- **Standards Adherence:** Does the code follow language style guides and project conventions?
- **Error Handling:** Are errors handled gracefully? Are logs sufficient for debugging?
- **Documentation:** Is necessary documentation (code comments, README updates, API spec changes) included?

# IX. Build & Continuous Integration (CI)

Our CI process, implemented using AWS CodePipeline and CodeBuild (or similar), automates the verification and packaging of code changes, providing rapid feedback to developers.

## A. CI Pipeline Stages (Example for a Microservice)

1. **Source Trigger:** Pipeline automatically starts upon commit/merge to specified branches (e.g., `develop`, `feature/*` via PR). Source pulled from CodeCommit/GitHub.
2. **Lint & Format Check:** Run linters (ESLint, Flake8) and formatters (Prettier, Black) to enforce code style. Fail build if checks fail.
3. **Unit Tests:** Compile code (if necessary) and run all unit tests using the appropriate framework (Jest, Pytest, JUnit). Fail build if any tests fail. Generate code coverage reports.
4. **Security Scans:**
    - Run SAST scan (SonarQube, Checkmarx) on the source code. Fail build based on configured quality gates (e.g., number/severity of new vulnerabilities).
    - Run SCA scan (Snyk, Dependency-Check) on dependencies. Fail build based on configured quality gates (e.g., presence of critical/high vulnerabilities without patches).
5. **Build Artifact:** Build the deployment artifact (e.g., create Docker image using a multi-stage Dockerfile, package Lambda deployment .zip).
6. **Push Artifact:** Push the built artifact to its repository (e.g., ECR for Docker images, S3 for Lambda zips) tagged appropriately (e.g., with Git

commit hash or build number).

7. **(Optional) Trigger CD:** Optionally, trigger the Continuous Deployment pipeline to deploy the artifact to the DEV environment.
8. **Notifications:** Notify developers (e.g., via Slack/Teams/Email) of build success or failure, linking back to the pipeline execution details and relevant commit/PR.

> *Integration and E2E tests are typically run in separate pipelines triggered *after* deployment to QA/Staging environments, not usually within the primary CI build pipeline itself.*