NYCPS TMS: Hyper-Detailed Prescriptive Test Engineering Strategy

I. Introduction: Principles & Goals

This document mandates the comprehensive, end-to-end Test Engineering Strategy for the NYCPS Transportation Management System (TMS). It establishes the processes, tools, roles, responsibilities, and rigorous quality gates necessary to ensure the delivery of a high-quality, secure, reliable, performant, and compliant system. This strategy is inextricably linked with the defined Development and DevSecOps processes, operating within an Agile (Scrum) framework, utilizing GitLab, and targeting AWS GovCloud.

Our core testing philosophy is **Continuous Testing**, integrated throughout the entire lifecycle, with a strong emphasis on **Automation First** and **Developer Ownership of Quality**. The dedicated QA team acts as quality advocates, automation experts, and facilitators of specialized testing, but the primary

responsibility for verifying functional correctness rests with the development team through automated tests.

Mandatory Testing Goals:

1. **Comprehensive Requirement Coverage:** Achieve verifiable coverage (aiming for 100%) of all documented functional requirements (User Story Acceptance Criteria) and non-functional requirements (Performance, Security, Availability, Accessibility SLAs/targets) through a combination of automated and manual testing techniques.
2. **Early Defect Detection & Prevention ("Shift Left"):** Identify and fix defects as early as possible in the lifecycle (ideally during development or CI) through pre-commit checks, unit testing, integration testing, SAST, and SCA.
3. **High Test Automation Rate:** Automate all unit, integration, and API tests. Automate critical E2E paths, regression suites, performance baseline tests, and security/accessibility scans. Minimize reliance on manual regression testing.
4. **Rapid Feedback Loops:** Ensure automated tests integrated into the CI/CD pipeline provide feedback to developers within minutes (<10-15 mins target for core CI).
5. **Rigorous Quality Gates:** Implement strict, automated quality gates within the CI/CD pipeline that prevent defective, insecure, or non-performant code from progressing to subsequent environments or production.

6. **Confidence in Releases:** Ensure that every release deployed to production has passed all defined quality gates and meets the highest standards of quality, reliability, and security.

II. Roles and Responsibilities in Testing

Quality is a shared responsibility, but specific roles have distinct areas of focus and accountability within this testing strategy.

- **Developers (Frontend/Backend/Mobile):**
  - **Primary Owner** of unit testing, component testing (UI), and integration testing for the code they write.
  - Write tests concurrently with feature code (TDD/BDD encouraged).
  - Ensure tests meet coverage targets and pass locally before committing/creating MRs.
  - Fix bugs identified in their code (from any testing phase).

        - Contribute to API contract tests and potentially E2E test automation for their features.
        - Address findings from SAST/SCA/DAST scans related to their code.
    - **QA Engineers / Software Development Engineers in Test (SDETs):**
        - Define and maintain the overall Test Strategy and Test Plans.
        - Design, develop, and maintain automated test frameworks and scripts (especially for API, E2E, Regression).
        - Perform exploratory testing to uncover usability issues and edge cases.
        - Facilitate and support User Acceptance Testing (UAT).
        - Manage test environments and test data strategy.
        - Analyze test results, report defects clearly, and verify fixes.
        - Champion quality practices within the Scrum teams.
        - Certify release readiness from a QA perspective.

- **Performance Engineers (potentially specialized QA/SRE):**
    - Define performance testing strategy and NFRs.
    - Develop and maintain load/stress test scripts.
    - Execute performance tests against dedicated environments.
    - Analyze performance results, identify bottlenecks, and work with Dev/Ops on tuning.
    - Certify release readiness against performance NFRs.
- **Security Engineers / Application Security (AppSec) Team:**
    - Define security testing requirements and standards.
    - Configure and manage SAST, DAST, SCA, and container scanning tools within the pipeline.
    - Triage and validate findings from security scans.
    - Coordinate and review third-party penetration tests.
    - Provide secure coding guidance and training to developers.
    - Approve releases from a security perspective based on scan results and risk assessment.
- **Accessibility Specialist (potentially specialized QA/UX):**

        - Define accessibility testing strategy based on WCAG 2.0 AA.
        - Configure automated accessibility scanning tools.
        - Perform manual accessibility audits (screen reader, keyboard navigation).
        - Provide guidance to Dev/UX teams on accessible design and implementation.
        - Certify release readiness against accessibility standards.
    - **Site Reliability Engineers (SRE) / Operations Team:**
        - Implement and manage monitoring, logging, and alerting infrastructure.
        - Participate in defining operational readiness criteria for releases.
        - Execute DR tests.
        - Respond to production incidents and participate in RCAs.
        - Manage production infrastructure deployment (via automated CD pipelines).
    - **Product Owner (NYCPS Representative):**
        - Owns and prioritizes the Product Backlog.

- Defines Acceptance Criteria for User Stories.
- Participates in Sprint Reviews and formally accepts/rejects completed stories.
- Provides clarification on requirements during development and testing.
- Champions UAT and provides final business acceptance.

- **UAT Participants (NYCPS Stakeholders/End-Users):**
  - Execute predefined UAT scenarios in the Staging/UAT environment.
  - Provide feedback on usability and functional correctness from an end-user perspective.
  - Formally sign off on UAT completion.

III. Test Environments & Test Data Management

Stable, consistent, and well-managed test environments and data are crucial for effective testing.

A. Test Environments (Provisioned via Terraform)

- **Local Development:** Developer machines with Docker Compose/Dev Containers for running services and dependencies locally.
- **DEV (Development Integration):** AWS GovCloud environment deployed automatically from the `develop` branch. Used for CI builds, basic integration tests, and developer smoke testing. Data is ephemeral or uses anonymized/synthetic seed data.
- **QA (Quality Assurance):** AWS GovCloud environment deployed manually/scheduled from `develop`. Target for comprehensive automated testing (Integration, API, E2E, DAST, Accessibility) and manual exploratory testing by QA. Uses a stable, larger set of anonymized/synthetic data, refreshed periodically.
- **Staging / UAT:** AWS GovCloud environment, configured as closely to Production as possible (may use slightly smaller scale). Deployed manually from `release/` branches. Target for UAT by NYCPS stakeholders, final regression testing, and potentially penetration testing. Uses stable, production-like anonymized/synthetic data.
- **PERF (Performance Testing):** AWS GovCloud environment, scaled to mirror Production infrastructure load capacity. Deployed manually from `release/` branches specifically for load and stress testing. May use dedicated (synthetic) data sets designed for performance testing.
- **PROD (Production):** Live AWS GovCloud environment. Deployments are highly controlled. Monitoring is critical.

Implementation Steps:

1. Define distinct Terraform configurations (`.tfvars`) for each environment (DEV, QA, UAT, PERF, PROD) within the `environments/` directory structure.
2. Use Terraform modules to ensure consistency in resource provisioning across environments.
3. Automate environment provisioning and updates using Terraform via GitLab CI/CD pipelines, with manual approvals required for Staging/UAT, Perf, and Prod.
4. Implement strict network segmentation (Security Groups, NACLs) between environments and between tiers within environments.

Responsibility: DevOps Team, Cloud Architect.

B. Test Data Management (TDM)

Strict adherence to FERPA, NY Ed Law 2-d, and other privacy regulations is mandatory. Production PII MUST NOT be used in non-production environments (DEV, QA, PERF). Staging/UAT requires rigorously anonymized/synthesized data if production-like data is needed.

- **Strategy Definition:** Define a clear TDM strategy outlining data sources, anonymization/synthesization techniques, storage locations, refresh cadence, and access controls for test data in each environment.
- **Data Generation Tools:** Utilize data masking tools, synthetic data generation libraries (e.g., Faker), or custom scripts to create

realistic, referentially intact, but non-PII datasets.

- **Data Refresh Process:** Establish automated or semi-automated processes (e.g., scheduled Glue jobs, database scripts) to refresh test data in QA and Staging/UAT environments periodically, ensuring data relevance without using fresh production data.
- **Environment Isolation:** Ensure test environments cannot access production databases or data stores.
- **Data for Specific Tests:** Create specific, smaller datasets tailored for unit, integration, and performance tests where needed.

Responsibility: QA Lead, Data Architect/Engineer, Development Teams, Security/Compliance Officer.

IV. Detailed Test Levels & Types: Execution and Implementation

This section details *what* specific testing activities will occur at each level, *how* they will be implemented and automated using GitLab CI/CD, and the associated quality gates.

A. Unit Testing

1. What & Why:

- Verify individual units of code (functions, methods, classes, UI components) work correctly in isolation.
- Fastest feedback loop for developers during coding. Catches logic errors early.
- Forms the base of the test automation pyramid.

2. How (Implementation):

1. Developers write unit tests using standard frameworks (Jest/RTL for Frontend, Pytest for Python, JUnit/Mockito for Java) concurrently with feature code (TDD/BDD).
2. Focus on testing business logic, boundary conditions, error handling, and state changes within the unit.
3. External dependencies (other classes, API calls, database access) are mocked or stubbed using libraries (e.g., Jest `fn()`, Python `unittest.mock`, Mockito).
4. Tests are stored alongside the source code (e.g., `*.test.js`, `tests/test_*.py`).
5. Executed locally by developers before commits via IDE plugins or CLI commands (`npm test`, `pytest`).
6. Executed automatically in GitLab CI pipeline on every commit/MR using defined jobs in `.gitlab-ci.yml` (e.g., `script: npm test -- --coverage`).

7. Code coverage reports generated (e.g., using Jest `--coverage`, `coverage.py`, JaCoCo) and potentially checked against thresholds in CI (e.g., using GitLab CI test coverage parsing).

Tools: Jest, React Testing Library, Pytest, `unittest.mock`, JUnit, Mockito, JaCoCo, `coverage.py`, GitLab CI.

Responsibility: Developers.

Quality Gate (Local): Pre-commit hook may run fast unit tests. Quality Gate (CI - MR & `develop`): 100% pass rate required. Code coverage must meet defined threshold (e.g., >80% statement/branch). Build Fails otherwise.

B. Integration Testing

1. What & Why:

- Verify the interaction and communication between integrated components or a service and its direct external dependencies (database, message queue, cache, closely coupled internal API).
- Ensures components work together as expected, validating data flow, contracts, and basic interaction logic.
- Catches issues related to data persistence, messaging, configuration, and network connectivity within a controlled scope.

2. How (Implementation):

1. Developers write integration tests focusing on interaction points.

2. For local execution: Use Testcontainers to spin up ephemeral Docker instances of dependencies (Postgres, Redis, LocalStack for SQS/SNS).
3. For CI/CD execution: Run against deployed services in the DEV or QA environment. Tests target the service's API or trigger message consumers.
4. Tests assert correct state changes in dependencies (e.g., data written to DB, message published to queue), correct responses from dependencies, or correct processing of messages.
5. External 3rd party services or less closely coupled internal services are typically mocked/stubbed at this level to maintain focus and stability (using tools like `WireMock`, `moto`, custom mocks).
6. Executed locally by developers. Executed automatically by GitLab CI *after* successful deployment to DEV and QA environments.

Tools: Pytest, JUnit, Testcontainers, WireMock, `moto`, Newman (for API interactions), GitLab CI/CD.

Responsibility: Developers.

> **Quality Gate (Post-Deploy DEV/QA): Critical integration test suite must pass. Failures block promotion to the next environment (e.g., QA -> Staging) and require investigation/fixes.**

C. Component Testing (UI)

1. What & Why:

- Test individual UI components or small groups of interacting components in isolation from the full application.
- Verifies rendering, state changes, event handling, and props interaction without needing a full backend or browser environment. Faster and more reliable than E2E tests for component-level logic.

2. How (Implementation):

1. Frontend developers write component tests using frameworks like React Testing Library (preferred for testing user perspective) or potentially Cypress Component Testing.
2. Tests mount the component(s), simulate user interactions (clicks, input), and assert the rendered output or state changes.
3. Backend API calls are mocked at the network level or via service layer mocks. State management stores might be initialized with specific test states.
4. Executed locally by developers and automatically in the CI pipeline (MR & `develop`) alongside unit tests.

Tools: Jest, React Testing Library, Cypress Component Testing, MSW (Mock Service Worker).

Responsibility: Frontend Developers.

> **Quality Gate (CI - MR & `develop`): 100% pass rate required. Build fails otherwise.**

D. API Contract Testing

1. What & Why:

- Verify that API clients and providers adhere to the agreed-upon contract (OpenAPI specification). Checks request/response schemas, data types, status codes, and basic headers.
- Catches breaking changes in APIs early, preventing integration issues between microservices or between frontend and backend.

2. How (Implementation):

1. **Schema Validation:** Integrate OpenAPI schema validation into automated API tests (using libraries like `jest-openapi`, `chai-openapi-response-validator`, or custom validation against the spec).
2. **Consumer-Driven Contracts (Optional but Recommended):** Use Pact or similar tools. Consumers define expected interactions (requests/responses) in a contract file. Provider tests verify they fulfill these contracts. CI pipelines check for contract compatibility. This ensures providers don't break existing consumers.
3. **Automated API Tests:** Basic API tests (using Postman/Newman, RestAssured, Pytest) executed post-deployment to DEV/QA environments check status codes and basic response structures against the spec.

Tools: OpenAPI Specification, Schema validation libraries, Pact/PactFlow, Postman/Newman, RestAssured, Pytest.

Responsibility: Backend Developers (primarily), Frontend Developers (for consumer contracts).

> **Quality Gate (Post-Deploy DEV/QA): API schema validation and critical contract tests must pass. Failures block promotion.**

E. End-to-End (E2E) Testing

1. What & Why:

- Simulate real user scenarios by interacting with the application through the UI (Web or Mobile) and verifying workflows across multiple components and services.
- Provides highest confidence that integrated system meets user requirements. Catches issues missed by lower-level tests.

2. How (Implementation):

1. QA Automation Engineers (SDETs) design and implement E2E test scripts using frameworks like Cypress (for Web) or Appium/Detox (for Mobile). Developers contribute scripts for their features.
2. Tests focus on critical user journeys identified during requirements/design (e.g., Parent views bus location, Driver completes route with ridership scans, Admin assigns route).
3. Tests interact with the application deployed in QA or Staging environments, using dedicated test user accounts and interacting with the full (or near-full) integrated system stack. External 3rd party dependencies might be mocked if unstable or costly.

4. Test data needs careful management (creation before test, cleanup after).
5. Executed automatically via GitLab CI/CD after deployments to QA and Staging (e.g., nightly or triggered).
6. Utilize parallel execution features of frameworks/runners to reduce execution time.
7. Integrate visual regression testing tools (e.g., Percy, Applitools) if UI consistency is critical.

Tools: Cypress, Playwright, Selenium (Web); Appium, Detox, Espresso/XCUITest wrappers (Mobile); GitLab CI/CD; Percy/Applitools (Visual Regression).

Responsibility: QA Automation Engineers (primary), Developers (contributors).

Quality Gate (Post-Deploy QA/Staging): Critical E2E test suite must pass. Failures block promotion to the next stage (Staging/Prod) and require investigation. Flakiness must be actively managed.

F. Manual & Exploratory Testing

1. What & Why:

- Leverage human intuition and domain knowledge to uncover bugs, usability issues, and edge cases not easily caught by automated scripts.
- Provide qualitative feedback on the user experience.

- Verify complex scenarios or visual aspects difficult to automate.

2. How (Implementation):

1. QA Engineers perform session-based exploratory testing in QA and Staging environments, guided by test charters or areas of focus based on new features or risk analysis.
2. Focus on usability, workflow coherence, error handling resilience, visual consistency across browsers/devices, and areas not covered well by automation.
3. Utilize browser developer tools, proxies (like Charles/Fiddler), and mobile device simulators/emulators.
4. Document findings clearly as bugs or feedback in Jira/ADO.
5. Performed iteratively throughout sprints (on QA) and more formally before releases (on Staging).

Tools: Browser DevTools, Charles Proxy/Fiddler, Mobile Simulators/Emulators, Jira/ADO.

Responsibility: QA Engineers.

> **Quality Gate (Pre-Release): No blocking bugs found during final exploratory testing cycle before production release decision.**

G. User Acceptance Testing (UAT)

1. What & Why:

- Formal validation by end-users/stakeholders (NYCPS representatives) confirming the system meets business requirements and is acceptable for production use.
- Final check from the perspective of those who will actually use the system day-to-day.

2. How (Implementation):

1. QA Lead/Product Owner develops UAT Plan outlining scope, schedule, participants, environment (Staging/UAT), and scenarios based on user stories/business processes.
2. QA team prepares the UAT environment and test data.
3. Facilitated UAT sessions are conducted where NYCPS users execute test scenarios.
4. Participants provide feedback and formally sign off (or raise blocking issues) via defined process (e.g., UAT sign-off forms, Jira workflow).
5. Defects raised during UAT are triaged and prioritized for fixing before production release.

Responsibility: NYCPS UAT Participants (Execution/Sign-off), QA Lead/Product Owner (Facilitation/Planning), Developers (Fixing UAT defects).

> **Quality Gate (Pre-Release): Formal UAT sign-off obtained from designated NYCPS representatives. No blocking UAT defects remain unresolved.**

H. Performance & Load Testing

1. What & Why:

- Verify system performance (response time, throughput), scalability, and stability under realistic and peak load conditions.
- Ensure the system meets defined NFRs and SLAs. Identify and eliminate performance bottlenecks before production.

2. How (Implementation):

1. Performance Engineer/QA defines load scenarios based on expected usage patterns (user concurrency, API call rates, data volumes) for peak times (e.g., morning/afternoon school rush).
2. Develop automated scripts using tools like k6, JMeter, or Gatling to simulate load. Parameterize scripts to use realistic test data.
3. Execute tests against the dedicated, production-scaled PERF environment deployed via CI/CD.
4. Run different test types:
    - **Load Tests:** Simulate expected peak load to verify performance against NFRs.
    - **Stress Tests:** Gradually increase load beyond peak to identify breaking points and resource bottlenecks.
    - **Soak Tests:** Run moderate load for extended periods to detect memory leaks or resource

exhaustion issues.

5. Monitor key metrics during tests (response times P95/P99, error rates, resource utilization - CPU/Mem/Network/DB connections) using CloudWatch and APM tools.
6. Analyze results, identify bottlenecks (e.g., slow database queries, inefficient code, under-provisioned resources), and work with Dev/Ops teams to tune/fix issues.
7. Re-run tests after fixes to verify improvements.

Tools: k6, JMeter, Gatling, CloudWatch Metrics, APM tools (Datadog, Dynatrace).

Responsibility: Performance Engineer/QA, Developers (fixing bottlenecks), SRE/Ops (infrastructure tuning).

> **Quality Gate (Pre-Release): Performance test results demonstrate acceptable performance under load, meeting defined NFRs/SLOs. No critical performance regressions introduced.**

I. Security Testing (In-depth)

1. What & Why:

- Proactively identify and mitigate security vulnerabilities throughout the lifecycle.
- Ensure compliance with NYCPS, NYC3, and regulatory requirements (FERPA, etc.).
- Validate effectiveness of implemented security controls.

2. How (Implementation):

1. **Static Application Security Testing (SAST):**
    - Integrate SAST tools (SonarQube, GitLab SAST, Checkmarx) into GitLab CI pipeline (run on MRs and merges to `develop`).
    - Configure rulesets based on OWASP Top 10, SANS Top 25, and secure coding standards.
    - Triage findings automatically based on severity. Fail builds for Critical/High severity findings requiring immediate developer attention.
    - Track findings and remediation in Jira/ADO and security dashboards (e.g., SonarQube dashboard).

        Responsibility: CI/CD System, Developers (fixing), Security Team (tuning/review).

        > **Quality Gate (CI): Build fails on new Critical/High severity SAST findings.**

2. **Software Composition Analysis (SCA):**
    - Integrate SCA tools (GitLab Dependency Scanning, Snyk, OWASP Dependency-Check) into GitLab CI pipeline (run on MRs and merges to `develop`).

- Scan `package.json`, `pom.xml`, `requirements.txt`, etc., for known vulnerabilities (CVEs) in third-party libraries.
- Check for license compliance issues.
- Fail builds for Critical/High severity vulnerabilities with available patches. Alert on others.

> **Responsibility: CI/CD System, Developers (fixing), Security Team (review).**

> **Quality Gate (CI): Build fails on new Critical/High severity SCA findings with known fixes/patches.**

3. **Dynamic Application Security Testing (DAST):**
   - Integrate DAST tools (OWASP ZAP, Burp Suite integration) into GitLab CI/CD pipeline to run scans *after* deployment to QA and Staging environments (e.g., nightly or triggered).
   - Configure scans to crawl the application and test for common web vulnerabilities (XSS, SQLi, CSRF, etc.). Authenticated scans should be configured using test user credentials.
   - Report findings automatically to Jira/ADO and security dashboards.

Responsibility: Security/QA Team (config/triggering), CI/CD System (execution), Developers (fixing).

> **Quality Gate (Post-Deploy QA/Staging): New Critical/High DAST findings block promotion and must be addressed.**

4. **Container Image Scanning:**
    - Enable ECR Enhanced Scanning or integrate tools like Trivy/Clair into the CI pipeline *after* the `docker build` step.
    - Scan container images for OS package vulnerabilities and potentially misconfigurations.
    - Fail builds if Critical/High severity OS vulnerabilities are found in the final image.

Responsibility: CI/CD System, DevOps Team, Developers (updating base images/dependencies).

> **Quality Gate (CI): Build fails on Critical/High severity container image vulnerabilities.**

5. **Manual Penetration Testing:**
    - Schedule formal penetration tests by an approved third-party vendor and/or internal Red Team against the Staging environment before

major production releases or significant architectural changes.

- Scope should cover web applications, mobile applications, APIs, and cloud infrastructure configuration.
- Findings are tracked in Jira/ADO. Critical/High severity findings *must* be remediated before production release. Lower severity findings are added to the backlog based on risk assessment.

Responsibility: Security Team (coordination/review), Third-Party Vendor/Red Team (execution), Developers (fixing).

> **Quality Gate (Pre-Release): No unresolved Critical/High severity penetration test findings. Formal sign-off from Security Team.**

6. **Threat Modeling Review:** Periodically review and update threat models as the application architecture evolves.

J. Accessibility Testing (WCAG 2.0 AA)

1. What & Why:

- Ensure the application is usable by people with disabilities, including those using assistive technologies (screen readers, keyboard navigation).
- Mandated compliance requirement (WCAG 2.0 Level AA).

2. How (Implementation):

1. **Automated Scans:** Integrate Axe-core with E2E test frameworks (e.g., `cypress-axe`, `axe-playwright`). Run automated checks for common WCAG violations as part of the E2E test suite against QA/Staging environments.
2. **Manual Keyboard Testing:** QA/Accessibility Specialist performs manual checks ensuring all interactive elements are focusable and operable via keyboard alone in a logical order.
3. **Manual Screen Reader Testing:** QA/Accessibility Specialist tests key user flows using common screen readers (e.g., NVDA, JAWS, VoiceOver) to ensure content is perceivable, operable, and understandable.
4. **Design Review:** Accessibility Specialist reviews UI designs/prototypes for potential issues (color contrast, layout structure, input labeling) *before* development.
5. **Developer Training:** Provide training on accessible coding practices (semantic HTML, ARIA roles/attributes, focus management).

Tools: Axe-core, WAVE, NVDA, JAWS, VoiceOver, Browser DevTools (Accessibility Tree).

Responsibility: Frontend Developers (implementation/fixing), Accessibility Specialist (guidance/manual testing), QA (automated test

integration/manual checks).

> **Quality Gate (Pre-Release): Automated scans passing. No blocking (Level A/AA critical) accessibility issues identified during manual review. Formal sign-off from Accessibility Specialist/QA Lead.**

K. Regression Testing

1. What & Why:

- Ensure that new code changes, bug fixes, or infrastructure updates do not negatively impact existing functionality.
- Prevent re-introduction of previously fixed bugs.

2. How (Implementation):

1. Maintain comprehensive automated test suites (Unit, Integration, API, E2E).
2. Tag tests appropriately (e.g., `critical_path`, `regression`, `smoke`).
3. Run relevant regression suites automatically in GitLab CI/CD pipelines:
    - Fast unit/integration regression subset on MRs.
    - Full unit/integration/API suites on merge to `develop` and post-deploy to DEV/QA.

- Critical E2E regression suite post-deploy to QA/Staging (e.g., nightly or per deployment).

4. Prioritize automation of regression tests for previously fixed critical bugs.
5. QA performs targeted manual regression testing around areas impacted by recent changes, especially before major releases.

Responsibility: Developers (maintaining unit/integration tests), QA Automation Engineers (maintaining E2E/API suites), QA Engineers (manual regression).

Quality Gate (CI/CD & Pre-Release): Automated regression suites must pass. No critical regressions identified during manual checks.

L. Disaster Recovery (DR) Testing

1. What & Why:

- Validate the effectiveness of the DR plan and procedures.
- Ensure the team can meet the defined RPO and RTO targets in a simulated disaster scenario.
- Build confidence in the ability to recover critical services.

2. How (Implementation):

1. Schedule DR tests regularly (at least annually, potentially more frequently for critical components).

2. Define test scenarios (e.g., simulate primary region failure, database failure, AZ failure).
3. Execute the documented DR plan:
    - Failover database instances (e.g., promote RDS read replica or restore snapshot in DR region).
    - Deploy application infrastructure in the DR region using IaC (Terraform `apply` against DR environment config).
    - Update DNS (Route 53 failover records) to point to the DR environment.
    - Restore necessary data from backups/snapshots replicated to the DR region.
4. Validate system functionality and data integrity in the DR environment. Measure actual recovery time (RTO) and data loss (RPO).
5. Document test results, identify issues/gaps in the DR plan, and create action items for improvement.
6. Execute failback procedures to return operations to the primary region once testing is complete.

Responsibility: SRE/Ops Team, DevOps Team, Database Administrators, Application Teams.

Quality Gate (Post-Test): DR test report documenting success/failure against RPO/RTO targets, lessons learned, and remediation plan for any issues identified.

V. Test Artifacts & Reporting

Consistent documentation and reporting are essential for tracking progress, communicating status, and demonstrating compliance.

- **Test Plan (Master & Phase-Specific):** Documented in Confluence. Outlines strategy, scope, environments, tools, roles, schedule, entry/exit criteria.
- **Test Cases/Scenarios:** Managed in Jira (e.g., using Zephyr/Xray plugins) or Confluence. Linked to User Stories/Requirements. Includes preconditions, steps, expected results.
- **Automated Test Scripts:** Stored in Git repositories alongside application/infrastructure code.
- **Test Data:** Scripts for generation/anonymization stored in Git. Actual data stored securely according to TDM strategy.
- **Defect Reports:** Logged in Jira/ADO with standardized fields (severity, priority, steps to reproduce, environment, assigned owner,

linked tests/stories).

- **Test Execution Reports:** Generated automatically by CI/CD tools (GitLab CI/CD dashboards) and test frameworks. Shows pass/fail rates, duration, coverage.
- **Test Summary Reports:** Created by QA Lead at the end of major phases or before releases. Summarizes testing activities, results, coverage achieved, outstanding defects, risks, and go/no-go recommendation. Stored in Confluence.
- **Performance Test Reports:** Detailed analysis of load test results, comparison against NFRs, bottleneck identification. Stored in Confluence.
- **Security Scan Reports:** Output from SAST, DAST, SCA, Pen Test tools. Stored securely, findings tracked in Jira/ADO or dedicated security tool dashboards (e.g., GitLab Security Dashboard).
- **Accessibility Audit Reports:** Output from automated tools and manual review findings. Stored in Confluence/Jira.

Responsibility: QA Lead (overall ownership), QA Engineers, Developers, Performance Engineers, Security Engineers, Accessibility Specialists.

VI. Defect Management Process

A structured process for identifying, tracking, prioritizing, and resolving defects is critical for maintaining quality.

1. **Logging:** Defects found via automated tests, manual testing, UAT, or production incidents are logged promptly in Jira/ADO using a standardized defect template. Minimum info: Title, Steps to Reproduce, Actual Result, Expected Result, Severity, Priority, Environment, Component/Feature, Screenshots/Logs.
2. **Triage:** Daily "Bug Scrub" meeting (QA Lead, Dev Lead, PO/BA) reviews newly logged defects.
    - Confirm validity and reproducibility.
    - Assign Severity (e.g., Blocker, Critical, Major, Minor, Trivial) based on impact.
    - Assign Priority (e.g., Highest, High, Medium, Low) based on urgency and business value.
    - Assign defect to the appropriate developer or team lead for investigation/fixing.
3. **Resolution:** Developer fixes the defect on a dedicated branch (e.g., `bugfix/TMS-XXX`), including writing/updating automated tests to cover the scenario. Code follows standard review/merge process into `develop`.
4. **Verification:** QA Engineer verifies the fix in the appropriate environment (QA/Staging) where the bug was initially found. If verified, the defect is closed. If not, it's reopened and reassigned to the developer with details.

5. **Tracking & Reporting:** Defect status, aging, density, and resolution rates are tracked via Jira/ADO dashboards and reported regularly (e.g., in Sprint Reviews, weekly status reports).

Responsibility: All Team Members (logging), QA Lead/Dev Lead/PO (triage), Developers (fixing), QA Engineers (verifying).

VII. Alignment with Development & DevOps Strategy

This Test Engineering Strategy is designed to be intrinsically woven into the Agile Development and DevSecOps processes:

- **Agile/Scrum Integration:**
  - Test planning occurs during Sprint Planning (understanding acceptance criteria, identifying test scenarios).
  - Developers write unit/integration tests during the sprint as part of feature development.
  - QA performs exploratory testing and automation development throughout the sprint.

        - Meeting the testing criteria within the Definition of Done is required for story acceptance in Sprint Review.
        - Testing process improvements are discussed during Sprint Retrospectives.
    - **Git Workflow Integration:**
        - Pre-commit hooks enforce basic quality checks before code leaves the developer machine.
        - Automated tests (Unit, Component, SAST, SCA) run automatically on Merge Requests, acting as mandatory quality gates before merging to `develop`.
    - **CI/CD Pipeline Integration:**
        - The CI pipeline includes automated build, unit test, component test, SAST, SCA, and packaging steps with quality gates.
        - The CD pipeline includes automated deployment to test environments followed by automated Integration, API, E2E, DAST, and Accessibility tests, acting as further quality gates blocking promotion if failures occur.
        - Performance and DR tests are executed against dedicated environments provisioned via the CD pipeline.

- **DevSecOps Culture:**
    - Security testing (SAST, SCA, DAST, Container Scanning) is automated and integrated early ("Shift Left").
    - Developers are responsible for addressing security findings in their code.
    - Operations (SRE) collaborates on monitoring, incident response, DR testing, and performance optimization, providing feedback into development.
    - Shared responsibility for quality and security across Dev, QA, Sec, and Ops.

VIII. Continuous Improvement of Test Strategy

This strategy is a living document and will be continuously improved:

- **Metrics Review:** Regularly analyze test metrics (coverage, pass rates, defect escape rates, automation execution time, flaky test counts) to identify areas for improvement.

- **Retrospective Feedback:** Use Sprint Retrospectives and incident post-mortems to gather feedback on testing processes, tools, and effectiveness.
- **Tool Evaluation:** Periodically evaluate new testing tools and frameworks to enhance automation, coverage, or efficiency.
- **Strategy Updates:** Update this Test Strategy document based on lessons learned, process changes, and evolving project needs. Communicate changes to the entire team.

Responsibility: QA Lead, Test Engineering Team, in collaboration with Dev, Ops, Sec, and Project Management.