# NYCPS TMS: Prescriptive Observability, Monitoring, Alerting, & Production Support Strategy

## I. Introduction: Philosophy & Goals

This document mandates the comprehensive, hyper-detailed strategy for Observability, Monitoring, Alerting, Incident Management, and Production Support for the NYCPS Transportation Management System (TMS). This is not merely an operational function but a core pillar of our DevSecOps approach, designed to ensure the system's reliability, performance, security, and availability meet the stringent requirements of the NYCPS and its users.

Our philosophy is built upon **Site Reliability Engineering (SRE) principles**, emphasizing proactive monitoring, data-driven decision-making, automation of operational tasks (toil reduction), blameless post-mortems, and clearly defined

Service Level Objectives (SLOs). We aim to move beyond reactive troubleshooting to predictive and preventative maintenance.

### Mandatory Strategy Goals:

1. **Deep System Visibility (Observability):** Gain comprehensive insight into system behavior through the "three pillars": Metrics, Logs, and Traces, correlated across all components (infrastructure, application, database, frontend).
2. **Proactive & Actionable Monitoring/Alerting:** Detect potential issues *before* they impact users. Alerts must be specific, actionable, linked to runbooks, and minimize noise/fatigue. Alert on symptoms impacting user experience and SLOs.
3. **Rapid Incident Response & Resolution:** Implement a structured, efficient incident management process with clear roles, communication protocols, and escalation paths to meet aggressive Mean Time To Detect (MTTD) and Mean Time To Repair (MTTR) targets, aligning with RFP RTOs.
4. **Structured, Multi-Tiered Production Support:** Provide clear pathways for issue resolution, empowering L1/L2 support with knowledge and tools while ensuring efficient escalation to L3 (Dev/SRE) for complex problems.
5. **Continuous Learning & Improvement:** Utilize incident data, monitoring trends, and post-mortems to continuously refine monitoring, alerting, runbooks, system architecture, and the support process itself. Quantify reliability via SLOs and Error Budgets.

6. **Seamless SDLC Integration:** Embed observability and operational requirements into the design, development, testing, and deployment phases ("Shift Left" operations).

## II. Core Pillars of Observability: Implementation Details

We will implement a multi-faceted approach combining Metrics, Logging, and Tracing, providing complementary views into system health and behavior.

### A. Metrics: Measuring System Health & Performance

#### 1. What We Will Measure (Layered Approach):

- **Infrastructure Metrics (AWS Resources):**
  - **Compute (EC2, Fargate, Lambda):** CPU Utilization, Memory Utilization, Disk I/O (Read/Write Ops/Bytes), Network I/O (In/Out Bytes/Packets), Throttles (Lambda),

Running Tasks/Instances (ECS/EC2 AutoScaling).

- **Database (RDS, DynamoDB, ElastiCache):** CPU Utilization, Memory Usage (where applicable), Disk Queue Depth/IOPS/Latency (RDS/EBS), Database Connections, Read/Write Latency/Throughput/Throttles (DynamoDB), Cache Hit/Miss Rate (ElastiCache), Replication Lag (RDS).
- **Networking (ALB, API GW, NAT GW):** Request Count, Latency (Target/API GW), HTTP Error Codes (4xx, 5xx), Healthy/Unhealthy Host Count (ALB), Active Connections, NAT Gateway Error/Dropped Packet Counts.
- **Messaging (SQS, SNS, Kinesis/MSK):** ApproximateNumberOfMessagesVisible/Delayed (SQS), AgeOfOldestMessage (SQS), MessagesPublished (SNS), Put/Get Record Latency/Bytes (Kinesis), Broker/Topic Metrics (MSK - CPU, Network, Disk).
- **Storage (S3):** Request Counts (Get/Put), Latency, Error Rates (4xx/5xx).

    - **AWS Health Events:** Monitor via EventBridge for service degradation/outages impacting our regions/services.

- **Application Performance Metrics (APM):**
    - **Request Rate/Throughput:** Requests per second/minute per API endpoint/service.
    - **Request Latency (End-to-End & Internal):** P50, P90, P95, P99 latency distributions for key API endpoints and critical internal service calls.
    - **Error Rates:** Rate of HTTP 5xx, 4xx errors per endpoint/service. Rate of application exceptions (uncaught errors).
    - **Resource Usage (Runtime Specific):** JVM Heap/GC Metrics (Java), Event Loop Lag (Node.js), Process CPU/Memory (Python/Node/Java within container).
    - **Dependency Metrics:** Latency and error rates for calls to external services (DBs, caches, other microservices, 3rd party APIs).
- **Frontend Performance / Real User Monitoring (RUM):**
    - **Page Load Times:** Core Web Vitals (LCP, FID, CLS).

        - **JavaScript Errors:** Count and rate of frontend JS errors.
        - **API Call Performance (Client-Side):** Latency and error rates for API calls initiated from the browser/mobile app.
        - **Route Transitions / User Action Timings:** Time taken for specific user interactions.
    - **Synthetic Monitoring:**
        - **API Endpoint Checks:** Regularly ping key API endpoints to verify availability and basic response correctness.
        - **UI Workflow Checks:** Simulate critical user journeys (login, view map, submit absence) via browser automation to check end-to-end functionality and availability.
    - **Business KPIs:**
        - GPS Pings Processed per Minute.
        - Ridership Scans Processed per Minute.
        - Active Routes / Buses Online.
        - Route Calculation Success/Failure Rate & Latency.
        - Notification Delivery Success/Failure Rate.

- User Login Success/Failure Rate.
- (Derived) On-Time Performance metrics (based on comparison of actual vs scheduled times from logs/DB).

## 2. How We Will Implement Metrics Collection:

1. **AWS Native Metrics:** Leverage default CloudWatch metrics provided by AWS services (EC2, RDS, ALB, Lambda, SQS, etc.). Enable detailed monitoring where appropriate (e.g., EC2).
2. **CloudWatch Agent:** Deploy the CloudWatch agent via Systems Manager Distributor/State Manager (for EC2) or build into container images/sidecars (for ECS/Fargate) to collect detailed OS-level metrics (memory, disk space) and custom application metrics (via StatsD/collectd protocols or embedded metric format in logs).
3. **APM Tooling (if used, e.g., Datadog/Dynatrace):** Deploy APM agents alongside application code (as libraries or sidecars) to automatically instrument applications and collect detailed performance metrics and traces. Configure agents securely for GovCloud endpoints.
4. **Frontend Monitoring:** Integrate CloudWatch RUM SDK or a third-party RUM provider SDK into web and mobile applications.
5. **Synthetic Monitoring:** Create CloudWatch Synthetics Canaries (using Node.js or Python blueprints) for API and UI checks. Schedule regular execution (e.g., every 1-5 minutes for critical checks).

6. **Custom Metrics:** Applications will emit critical business KPIs or custom performance metrics to CloudWatch directly via AWS SDK calls or by publishing structured logs that Metric Filters can parse.
7. **Metric Storage & Visualization:** Primarily use CloudWatch Metrics. Build comprehensive dashboards in CloudWatch Dashboards, potentially supplemented by Grafana (if using Prometheus/other sources) or QuickSight (for BI/longer-term analysis). APM tools provide their own dashboards.

*Tools: AWS CloudWatch (Metrics, Agent, Synthetics, RUM), Terraform/CloudFormation (for agent deployment/config), APM Tool (Optional: Datadog, Dynatrace, New Relic), Grafana/QuickSight.*

**Responsibility: SRE/Ops (Infrastructure, Agent Deployment, Base Dashboards), Developers (App Instrumentation, Custom Metrics, APM Integration), QA (Synthetic Scripting).**

## B. Logging: Recording Events & State

### 1. What We Will Log:

- **Application Logs:**
  - Request/Response details for APIs (method, path, status code, latency, user ID, correlation ID - *carefully exclude PII/secrets from request/response bodies*).

- Key business events (e.g., route generated, ridership scanned, notification sent, user profile updated).
- Application lifecycle events (startup, shutdown).
- Errors and Exceptions (with full stack traces).
- Warnings for potential issues.
- Debug level logs (configurable, disabled in production by default).
- Security events (authentication success/failure, authorization failure, potential input validation failures).

- **Infrastructure Logs:**
  - Load Balancer Access Logs (ALB/NLB).
  - VPC Flow Logs (sampled or full, depending on security needs/cost).
  - RDS Database Logs (Postgres logs, audit logs if enabled).
  - Operating System Logs (syslog, auth logs - collected via CloudWatch Agent).

    - CloudTrail Logs (already configured for AWS API calls).
    - WAF Logs.
  - **Correlation ID:** A unique request identifier generated at the edge (API Gateway/ALB) or by the first service, passed through downstream service calls (via HTTP headers, message attributes) and included in *all* log messages related to that request.

### 2. How We Will Implement Logging:

1. **Structured Logging (JSON):** Mandate use of structured logging libraries (e.g., python-json-logger, Logback JSON encoder, Winston for Node.js) in all applications. Logs must be written to standard output/error within containers/Lambda functions. Standard fields include timestamp, log level, service name, correlation ID, message, and contextual key-value pairs.
2. **Log Collection:**
    - **Lambda:** Natively integrates with CloudWatch Logs.
    - **Fargate/ECS:** Configure task definitions to use the `awslogs` log driver, sending container stdout/stderr directly to designated CloudWatch Log Groups.
    - **EC2:** Deploy and configure the CloudWatch Agent to collect application logs, OS logs (syslog, auth.log), and metrics.

3. **Log Storage & Centralization:** Configure CloudWatch Log Groups per service/environment with appropriate retention policies (e.g., 30 days for DEV/QA, 1 year for PROD application logs, longer if required for compliance/audit logs). Consider streaming logs from CloudWatch to a centralized logging platform (AWS OpenSearch Service, Splunk, Datadog Logs) via Kinesis Data Firehose for advanced querying, analysis, and longer retention if CloudWatch native capabilities are insufficient.

4. **Log Access & Analysis:** Use CloudWatch Logs Insights for powerful querying and analysis of logs stored in CloudWatch. If using a centralized platform, leverage its query language (OpenSearch Query DSL, Splunk SPL). Grant appropriate IAM permissions for log access based on roles.

*Tools: CloudWatch Logs, CloudWatch Agent, Fluentd/FluentBit (optional sidecars), Structured Logging Libraries (language-specific), AWS OpenSearch Service / ELK / Splunk / Datadog Logs (optional central platform), Kinesis Data Firehose.*

**Responsibility: Developers (instrumenting code with structured logs), DevOps/SRE (configuring agents, log groups, retention, central platform if used).**

## C. Tracing: Understanding Request Flow

### 1. What We Will Trace:

- End-to-end requests starting from the user interface or external API call.
- Flows across microservices (synchronous API calls, asynchronous message passing via SQS/SNS/Kinesis).
- Interactions with AWS managed services (DynamoDB, RDS, S3, Lambda).
- Identify latency contributions of each component in a request path.
- Visualize service dependencies.

### 2. How We Will Implement Tracing:

1. **Instrumentation Standard:** Adopt **OpenTelemetry (OTel)** as the standard for instrumentation across all services. OTel provides vendor-neutral APIs and SDKs.
2. **Automatic Instrumentation:** Utilize OTel auto-instrumentation agents/libraries where available for supported frameworks (e.g., Java Agent, Python auto-instrumentation, Node.js auto-instrumentation) to capture common interactions (HTTP requests, DB queries) with minimal code changes.
3. **Manual Instrumentation:** Manually instrument critical code paths, asynchronous boundaries, and business logic using OTel SDKs to create custom spans and add relevant attributes (e.g., user ID, route ID, business context).
4. **Context Propagation:** Ensure trace context (Trace ID, Span ID) is automatically propagated across process boundaries (HTTP headers using W3C Trace Context standard, message

attributes for SQS/SNS/Kinesis). OTel SDKs typically handle this when configured correctly.

5. **Trace Backend/Exporter:** Configure OTel SDKs/Collectors to export trace data to **AWS X-Ray**. Use the AWS Distro for OpenTelemetry (ADOT) Collector for streamlined collection and export. (Alternatively, could export to other OTel-compatible backends like Jaeger, Zipkin, or APM vendors if required, but X-Ray provides native AWS integration).
6. **Sampling:** Configure appropriate sampling rates (e.g., 100% for critical transactions in dev/test, potentially lower percentage-based or adaptive sampling in production to manage cost, while ensuring X-Ray traces key requests).
7. **Visualization & Analysis:** Use the AWS X-Ray console to view service maps, trace timelines, identify errors, and analyze latency distributions. Integrate X-Ray trace IDs into structured logs for correlation.

*Tools: OpenTelemetry SDKs/APIs (language-specific), OpenTelemetry Collector (specifically ADOT Collector), AWS X-Ray.*

**Responsibility: Developers (instrumentation), DevOps/SRE (collector configuration, sampling strategy).**

## III. Alerting Strategy: From Detection to Action

Our alerting strategy focuses on detecting user-impacting issues and SLO violations quickly, routing actionable alerts to the right teams, and minimizing alert fatigue.

## A. Alerting Philosophy

- **Alert on Symptoms, Not Causes (Primarily):** Focus alerts on high-level indicators of problems affecting users or SLOs (e.g., high latency P99, high 5xx error rate, low success rate for core transactions, SQS queue depth exceeding processing capacity) rather than low-level causes (e.g., high CPU on one instance, unless it *directly* correlates to user impact).
- **Actionability:** Every alert MUST be actionable. If the recipient doesn't know what to do when an alert fires, the alert is ineffective. Link alerts to specific runbooks or troubleshooting guides.
- **Severity Levels & Routing:** Define clear severity levels (e.g., Critical/SEV1, Warning/SEV2, Info/SEV3) with distinct notification targets and response expectations.
    - **Critical (SEV1):** Production outage, significant user impact, SLO breach imminent or occurring, critical security event. **Target:** PagerDuty/Opsgenie -> On-call SRE/Ops immediate engagement. Expected response: Within 5-15 minutes.

        - **Warning (SEV2):** Potential future issue, performance degradation nearing SLO threshold, non-critical service errors increasing, resource saturation warnings. **Target:** Dedicated Slack/Teams channel for SRE/Ops/Dev, potentially low-urgency PagerDuty notification. Expected response: Within business hours or next on-call shift.
        - **Info (SEV3):** Informational events, successful completion of critical batch jobs, scaling events. **Target:** Slack/Teams channel, Email. No immediate action required.
    - **Minimize Noise:** Aggressively tune alert thresholds and evaluation periods based on historical data and baselines to avoid flapping and false positives. Use composite alarms where appropriate. Implement alert silencing/maintenance windows during planned activities.

## B. Alert Implementation

1. **Defining Alarms (IaC):**
    - Define CloudWatch Alarms primarily using Terraform (or CloudFormation). Store alarm definitions in Git alongside infrastructure/application code.

- Use variables in Terraform to manage thresholds and dimensions per environment.
- Standardize alarm naming conventions (e.g., `---`).
- Link `alarm_actions` and `ok_actions` to appropriate SNS topics configured for different severities.

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

### 2. Key Alert Categories (Examples):

- **Availability:** Synthetic check failures (API & UI), ALB 5xx error rate high, ALB UnHealthyHostCount > 0, Lambda Throttle/Error rate high, ECS Service desired count vs running count mismatch.**
- **Latency:** ALB/API GW Target Latency P95/P99 > X ms, RDS Query Latency high, Lambda Duration P95 > X ms.**
- **Traffic/Saturation:** SQS ApproximateNumberOfMessagesVisible > X for Y minutes, Kinesis GetRecords.IteratorAgeMilliseconds > X seconds, RDS DB Connections > X%, Compute (EC2/Fargate/Lambda) CPU/Memory Utilization consistently > X%.**
- **Errors:** Application 5xx error rate > X%, High rate of specific exceptions in logs (via Metric Filters), Database error log patterns.**
- **Security:** GuardDuty high/medium severity findings, Critical WAF blocks/counts, IAM policy changes (via Config/EventBridge), Root account usage.**

- **Business KPIs:** Critical job failure, Significant drop in ridership scan rate (anomaly detection).**

**Responsibility: SRE/Ops (Infrastructure/Platform), Developers (Application-specific alerts), Security Team (Security alerts).**

### 3. Alert Routing & On-Call:

- **Configure SNS topics for each severity level/team responsibility area.**
- **Integrate Critical/SEV1 SNS topics with PagerDuty/Opsgenie.**
- **Configure PagerDuty/Opsgenie services, escalation policies, and on-call schedules for SRE/Ops and potentially core Dev teams.**
- **Integrate Warning/Info SNS topics with Slack/Teams channels for broader visibility.**

***Tools: CloudWatch Alarms, SNS, PagerDuty/Opsgenie, Slack/Teams, Terraform.***

**Responsibility: SRE/Ops Team.**

### 4. Runbooks:

- **For every *actionable* alert, create a corresponding runbook/playbook stored in Confluence/Git wiki.**
- **Runbooks should include: Alert description, potential causes, initial diagnostic steps (e.g., specific CloudWatch dashboards/log queries, commands to run), mitigation/resolution steps, escalation contacts.**

- Link the runbook directly from the alert notification (e.g., in PagerDuty/Slack message).
- Keep runbooks up-to-date through regular review and post-incident updates.

Responsibility: SRE/Ops Team, Development Teams (for application-specific runbooks).

## V. Incident Management Process

We will follow a standardized, structured process for managing incidents impacting production services, prioritizing rapid restoration and communication, followed by thorough learning.

### A. Incident Lifecycle Stages:

1. Detection: Incident detected via automated alerting (CloudWatch, APM), monitoring dashboards, synthetic checks, or user reports (escalated via Support Tiers).
2. Engagement & Assessment: Automated alert triggers PagerDuty/Opsgenie, notifying the on-call SRE/Ops engineer. On-call acknowledges the alert, performs initial

assessment of impact and severity (assigns SEV level), and declares an incident if warranted.

3. Mobilization & Communication (for SEV1/SEV2):
    - Incident Commander (IC) role assigned (typically initial on-call, may rotate).
    - Dedicated incident communication channel created (e.g., Slack channel `#incident-YYYYMMDD-description`, persistent video call bridge).
    - Relevant SMEs (Dev, DB, Network, Security) paged/invited to join based on initial assessment.
    - Comms Lead assigned to manage internal/external stakeholder communication (Status Page updates, internal summaries).
    - Scribe assigned to document timeline, key decisions, actions taken in a shared document (e.g., Confluence page, Google Doc).
4. Diagnosis & Investigation: Team collaborates in the war room/channel, using monitoring dashboards, logs, traces, and runbooks to identify the root cause or contributing factors. IC coordinates efforts, avoids jumping to conclusions.

5. Mitigation & Resolution: Implement actions to restore service as quickly as possible. This might be a temporary workaround (e.g., rollback deployment, scale up resources, disable feature flag) or a permanent fix. Prioritize service restoration over root cause finding if necessary. Validate fix restores service.

6. Communication & Closure: Comms Lead provides regular updates internally and externally (Status Page). Once service is confirmed stable, IC declares the incident resolved. Final communications sent.

7. Post-Mortem (RCA): For all SEV1/SEV2 incidents (and others as deemed necessary), conduct a blameless post-mortem within 1-3 business days.
    - Focus on "what happened?", "what was the impact?", "how did we respond?", "what went well?", "what could be improved?", and "how do we prevent recurrence?".
    - Document detailed timeline, root cause(s), contributing factors, actionable follow-up items (assigned owners, due dates) tracked in Jira/ADO.
    - Share post-mortem report widely for learning.

### B. Implementation Details:

- Define clear SEV level definitions based on user impact (e.g., SEV1=System wide outage/major data loss, SEV2=Significant feature impairment/performance degradation for many users, SEV3=Minor feature impairment/performance issue for some users, SEV4=Cosmetic issue/low impact bug).
- Configure PagerDuty/Opsgenie schedules, escalation policies (e.g., escalate to secondary on-call if no ack within 10 mins, escalate to manager if unresolved after 1 hour).
- Create incident response runbook templates in Confluence.
- Create incident ticket templates in Jira/ADO.
- Set up Status Page (e.g., Statuspage.io, Cachet) for external communication during major incidents.
- Conduct regular incident response drills or "Game Days" to practice the process.

**Responsibility: SRE/Ops Team (leads process), All Engineers (participate as needed), Incident Commander (coordination), Comms Lead (communication).**

## VI. Production Support Model (Tiered)

**A multi-tiered support structure ensures issues are handled efficiently by the appropriate team, providing clear escalation paths and leveraging specialized knowledge.**

## A. Support Tiers & Responsibilities:

- Tier 1 (L1): Service Desk / Initial Triage
    - **Who: Dedicated Service Desk team (potentially shared NYCPS resource or vendor team). Requires basic system understanding and strong customer service skills.**
    - **Responsibilities: First point of contact for user-reported issues (calls, emails, tickets). Log all issues in the Ticketing System (Jira Service Management/ServiceNow). Perform initial classification and prioritization. Resolve basic issues using Knowledge Base (KB) articles and standard operating procedures (SOPs) (e.g., password resets, simple configuration guidance, known issue workarounds). Gather necessary information for escalation. Escalate**

unresolved issues to L2 within defined SLA timeframes.

    - **Tools: Ticketing System, Knowledge Base (Confluence), Basic Monitoring Dashboards (Read-only).**

- **Tier 2 (L2): Application / System Support**
    - **Who: Application Support specialists, junior SRE/Ops engineers with deeper system knowledge.**
    - **Responsibilities: Handle escalations from L1. Perform in-depth troubleshooting using monitoring tools (CloudWatch, APM, logs). Analyze application logs and configurations. Execute more complex runbooks. Investigate system performance issues. Identify potential bugs or infrastructure problems. Resolve configuration issues, application-level problems not requiring code changes. Escalate code bugs, platform issues, or complex infrastructure problems to L3. Contribute to KB/runbooks based on resolved issues.**
    - **Tools: Ticketing System, CloudWatch (Logs Insights, Metrics, Dashboards), APM Tool, Tracing Tool (X-Ray), Runbooks**

(Confluence), potentially read-only database access for specific queries.

- Tier 3 (L3): Engineering / Expert Support
  - Who: Development Teams (Backend, Frontend, Mobile), Senior SRE/Ops Engineers, Security Engineers, Database Administrators (DBAs), Cloud Architects.
  - Responsibilities: Handle escalations from L2 requiring deep technical expertise or code/infrastructure changes. Debug complex application code. Diagnose and resolve infrastructure/platform issues (AWS service problems, networking, OS-level). Fix code bugs (following standard Dev workflow). Address security vulnerabilities/incidents. Perform complex database operations/tuning. Implement architectural changes needed for reliability/performance. Provide definitive root cause analysis. Update runbooks/documentation based on complex fixes.
  - Tools: All L2 tools + IDEs, Debuggers, Code Repositories (GitLab), CI/CD Pipelines, Terraform, AWS Console (with

appropriate permissions), Database Admin tools, Security analysis tools.

## B. Implementation & Process:

1. Ticketing System Configuration: Set up Jira Service Management (or equivalent) with queues for each support tier, automated routing rules based on issue type/severity, and SLA tracking.
2. Knowledge Base (KB) Development: Create and maintain a comprehensive KB in Confluence, populated with SOPs, runbooks, FAQs, known error database (KEDB), and troubleshooting guides. L2/L3 teams are responsible for documenting solutions.
3. Escalation Procedures: Clearly document criteria and steps for escalating issues between tiers (e.g., unresolved within X hours, requires specific permissions, requires code change). Define warm handoff procedures.
4. Training: Provide specific training to each tier on the tools, processes, and scope of their responsibilities. L1/L2 need strong KB/runbook usage training.
5. Feedback Loop: Regularly review ticket trends, escalation patterns, and resolution times to identify areas for improvement (e.g., better L1 documentation, automation opportunities, recurring bugs needing L3 fixes).

Responsibility: Support Manager (overall process), L1/L2/L3 Team Leads/Members (execution within tier), SRE/DevOps (KB/Runbook contribution).

## VII. Site Reliability Engineering (SRE) Practices

We will adopt core SRE principles to systematically improve reliability, performance, and operational efficiency.

- Service Level Objectives (SLOs) & Error Budgets:
  - Define SLOs: Collaboratively define specific, measurable SLOs for critical user journeys (e.g., Parent App bus location check latency < 500ms P95, Ridership scan processing success rate > 99.9%, API Gateway availability > 99.95%). Base SLOs on user expectations and business requirements.
  - Define SLIs: Identify the corresponding Service Level Indicators (metrics) used to measure SLOs (e.g., ALB latency metric, custom success rate metric).

    - **Establish Error Budgets: Calculate the acceptable level of failures/unavailability based on the SLO target (e.g., 99.95% availability allows ~22 mins downtime/month).**
    - **Monitor & Alert on Budgets: Track SLI performance against SLOs. Alert proactively when the error budget is being consumed too quickly.**
    - **Use Budgets for Prioritization: If the error budget is consistently depleted, prioritize reliability work (bug fixes, performance improvements, infrastructure upgrades) over new feature development for that service until reliability improves.**

        **Responsibility: SRE Team, Product Owner, Tech Leads.**

- **Toil Reduction & Automation:**
    - **Identify Toil: Actively identify manual, repetitive, automatable, tactical tasks with no enduring value performed by Ops/SRE/Support teams (e.g., manual deployments, report generation, common alert responses, user onboarding).**

        - Automate: Dedicate engineering time (SREs and Devs) to automate identified toil using scripting (Python, Bash), IaC (Terraform), CI/CD pipelines, or building internal tools.
        - Goal: Aim for SREs to spend <50% of their time on operational toil, freeing up time for engineering projects that improve reliability and automation.

            Responsibility: SRE Team, DevOps Team, Development Teams.

    - Capacity Planning:
        - Monitor Trends: Track historical resource utilization (CPU, memory, disk, network, DB connections, queue depths) and request rates.
        - Forecast Needs: Project future resource requirements based on usage trends, anticipated growth (e.g., start of school year), and new feature rollouts.
        - Provision Proactively: Adjust auto-scaling configurations, provision additional database/cache capacity, or upgrade instance types *before* resource

saturation impacts performance. Utilize load testing to validate capacity limits.

Responsibility: SRE/Ops Team.

- **Release Engineering:**
    - SRE team collaborates with Dev and QA on designing safe and reliable deployment strategies (Blue/Green, Canary).
    - Define operational readiness checklists for new services/features entering production.
    - Automate rollout/rollback procedures within the CI/CD pipeline.

    Responsibility: SRE Team, DevOps Team, Development Leads.

- **Blameless Culture:** Foster and enforce a culture where system failures and incidents are treated as learning opportunities to improve the system and processes, not to assign individual blame. Post-mortems focus on technical and process factors.

## VIII. Integration with SDLC & Development/Testing Strategy

**This Observability and Support strategy is not separate from development and testing but deeply integrated:**

- **Planning/Requirements: Non-functional requirements related to performance, availability (SLOs), and supportability are defined early. Observability needs for new features are considered.**
- **Design: Systems are designed for observability (structured logging, tracing points, key metrics). Runbooks are drafted during design. Monitoring/alerting strategy influences architectural choices (e.g., choosing managed services with built-in monitoring).**
- **Development: Developers instrument code for logs, metrics, and traces using standard libraries/SDKs. Developers write unit/integration tests that may verify logging output or metric emission. Runbooks are refined.**
- **Testing: QA verifies monitoring coverage and alert accuracy in test environments. Performance testing validates SLOs. Security testing validates audit logging. DR testing validates recovery monitoring and alerting. Exploratory testing may uncover gaps in observability.**

- Deployment (CD): Pipelines include steps to configure monitoring/alerting for new resources (e.g., applying CloudWatch alarm Terraform modules). Post-deployment smoke tests check key observability endpoints. Monitoring status is a key factor in Blue/Green/Canary rollout decisions.
- Operations: Monitoring data, incident trends, and post-mortem action items feed directly back into the product backlog, influencing prioritization for future sprints (continuous improvement loop).

## IX. Continuous Improvement

We will continuously refine this strategy based on operational experience and evolving best practices.

- Regular Reviews: Conduct quarterly reviews of the overall observability strategy, tooling effectiveness, alert fatigue levels, incident trends, and SLO adherence.
- Post-Mortem Actions: Ensure action items from incident RCAs related to monitoring, alerting, or runbooks are tracked and implemented.
- Tool Evaluation: Periodically assess new AWS features or third-party tools related to observability, APM, log analysis,

and incident management.

- Training: Provide ongoing training to Dev, Ops, SRE, and Support teams on observability tools, incident response processes, and SRE principles.
- Documentation Updates: Keep all related documentation (this strategy, runbooks, KB articles, monitoring dashboards) current.

Responsibility: SRE Lead, Ops Manager, QA Lead, Dev Leads.