DRAFT — UNOFFICIAL — NOT FOR OPERATIONAL USE

TECHNICAL MANUAL

SL 4O

USAREUR-AF

TM-40O — MAVEN SMART SYSTEM (MSS)

Specialist Course Manual

HEADQUARTERS
UNITED STATES ARMY EUROPE AND AFRICA
(USAREUR-AF)
Wiesbaden, Germany

DRAFT — NOT FOR OFFICIAL USE. FOR TRAINING PLANNING PURPOSES ONLY.

26 MARCH 2026

DRAFT — UNOFFICIAL — NOT FOR OPERATIONAL USE

TM-40O — MAVEN SMART SYSTEM (MSS)

Forward: SL 4O qualifies platform engineers to design, deploy, secure, and operate the infrastructure layer that MSS applications run on — Kubernetes clusters, CI/CD pipelines, container registries, GitOps workflows, and the DevSecOps toolchain. This is an infrastructure manual — it contains architecture, configuration, and operational procedures, not application code. **Prereqs:** SL 1, Maven User; SL 2, Builder; SL 3, Advanced Builder (required); Linux systems administration proficiency; familiarity with containers and version control; CONCEPTS_GUIDE_TM40O_PLATFORM_ENGINEER (read before this manual) *HQ USAREUR-AF · v1.0 · 2026 · DISTRIB: USG only · AUTH: C2DAO/UDRA v1.1*

WARNING: Infrastructure errors at SL 4O level affect every application and every user on the platform simultaneously. A misconfigured cluster, a broken pipeline, or a compromised container image has blast radius across the entire MSS ecosystem. Apply defense-in-depth. Test changes in non-production first. Maintain rollback capability for every change. CAUTION: Platform credentials, service account tokens, cluster certificates, and container registry keys are high-value operational secrets. Compromise of platform-level credentials constitutes a critical security incident with potential access to all tenant data. Report immediately to unit S6/G6, C2DAO, and ISSM.

TABLE OF CONTENTS

- CHAPTER 1 — INTRODUCTION: THE PLATFORM ENGINEER ROLE IN MSS
- CHAPTER 2 — PLATFORM-AS-PRODUCT
- CHAPTER 3 — KUBERNETES FOR MSS
- CHAPTER 4 — INFRASTRUCTURE AS CODE (IaC)
- CHAPTER 5 — CONTAINER SECURITY AND SUPPLY CHAIN
- CHAPTER 6 — CI/CD PIPELINE DESIGN
- CHAPTER 7 — COMPUTE MODULES
- CHAPTER 8 — AIR-GAPPED AND DDIL OPERATIONS
- CHAPTER 9 — OBSERVABILITY
- CHAPTER 10 — PLATFORM SECURITY AND COMPLIANCE
- APPENDIX A — GLOSSARY
- APPENDIX B — REFERENCES

CHAPTER 1 — INTRODUCTION: THE PLATFORM ENGINEER ROLE IN MSS

1-1. Platform Engineer Specialist Manual

BLUF: SL 4O qualifies platform engineers to build, operate, and secure the infrastructure that MSS runs on. The platform engineer builds the floor that every other track stands on.

This manual provides task-based instruction for platform engineers operating on the Maven Smart System (MSS). MSS is the USAREUR-AF enterprise AI/data platform built on Palantir Foundry. SL 4O personnel design and maintain the infrastructure layer — the compute, networking, security, and delivery mechanisms that make application development possible.

SL 4O covers platform-as-product: designing internal developer platforms that serve application teams; Kubernetes: cluster architecture, workload scheduling, resource management, multi-cluster operations; Infrastructure as Code (IaC): declarative configuration, GitOps workflows, continuous reconciliation, environment parity; container security: DoD-hardened images (Iron Bank), vulnerability scanning, SHA256 digest pinning, supply chain security; CI/CD pipeline design: automated build, test, scan, and deploy workflows for MSS applications; release engineering: deployment orchestration, blue/green and canary strategies, rollback procedures; air-gapped and DDIL operations: deploying across classification boundaries, disconnected operations, edge cluster management; and platform security: RMF/ATO lifecycle from the infrastructure perspective, STIG compliance, continuous monitoring.

SL 4O does NOT cover application code development (OSDK, TypeScript, Python transforms) — see SL 4L (Software Engineer); Workshop or Slate application design — see SL 4N (UI/UX Designer), SL 3; data pipeline design or Ontology modeling — see SL 3, SL 4H; ML model training or deployment — see SL 4M (ML Engineer); program/project management — see SL 4J (Program Manager); or Foundry platform administration (Foundry-native admin is managed by Palantir; SL 4O covers the surrounding infrastructure).

> **NOTE**
>
> SL 4O is peer to SL 4L (Software Engineer), SL 4N (UI/UX Designer), and SL 4J (Program Manager). The Platform Engineer supports all application teams by providing reliable, secure, and automated infrastructure. Coordinate across tracks — platform changes affect every downstream consumer.

1-2. Curriculum Position, Advanced Track, and WFF Context

Prerequisite: SL 3 (Advanced Builder) is REQUIRED. Linux systems administration proficiency (intermediate or higher), familiarity with containers (Docker or equivalent), and version control (Git) are required independently of the TM series.

Advanced track: Upon completing SL 4O, qualified Platform Engineers should pursue **SL 5O (Advanced Platform Engineer)** for advanced topics including multi-cluster fleet management, platform reliability engineering (SLOs/SLIs), RMF/ATO automation, and developer experience engineering.

Peer specialist tracks: The Platform Engineer provides the infrastructure that all other tracks depend on. Coordinate with SL 4L (Software Engineer) on CI/CD pipeline requirements, deployment workflows, and development environment provisioning. Coordinate with SL 4H (AI Engineer) and SL 4M (ML Engineer) on compute resource requirements for model training and inference workloads. Coordinate with SL 4N (UI/UX Designer) on performance budgets and deployment constraints that affect application design. Coordinate with SL 4K (Knowledge Manager) on data pipeline infrastructure requirements.

WFF awareness: Platform Engineers on MSS maintain the infrastructure that every WFF-qualified user (SL 4A through SL 4F) depends on. When the platform is down, every application is down. When the pipeline is broken, no code reaches production. When the security posture degrades, the ATO is at risk and the entire MSS capability may be suspended. Platform reliability is the foundational WFF readiness factor.

1-3. The Platform Engineer's Role in USAREUR-AF

USAREUR-AF is the Army Service Component Command (ASCC) to USEUCOM and USAFRICOM. MSS supports theater land operations across the European and African AOR including III Corps, V Corps, 21st TSC, 7th ATC, 10th AAMDC, 56th MDC-E, SETAF-AF, G2 all-source, and multinational elements. Platform Engineers at SL 4O level are the infrastructure architects and operators of the USAREUR-AF data ecosystem.

The SL 4O role in the data chain:

```
MISSION REQUIREMENT
        |
   APPLICATION DESIGN (SL 4N, SL 4J)
        |
   APPLICATION CODE (SL 4L, SL 4H, SL 4M)
        |
   CI/CD PIPELINE (SL 4O)
        |
   CONTAINER BUILD + SECURITY SCAN (SL 4O)
        |
   DEPLOYMENT TO CLUSTER (SL 4O)
        |
   PLATFORM MONITORING (SL 4O)
        |
OPERATIONAL USE (all tracks, all users)
```

The Platform Engineer operates at the base of the stack. Application developers (SL 4L) write code; the Platform Engineer ensures that code can be built, tested, scanned, deployed, monitored, and rolled back — automatically, securely, and repeatably. In USAREUR-AF, this means the platform must function

across multiple classification domains, in OCONUS environments with variable network connectivity, and under the security requirements of a forward-deployed theater command.

CHAPTER 2 — PLATFORM-AS-PRODUCT

2-1. Internal Developer Platform (IDP) Concept

BLUF: The platform is not a set of tools — it is a product, and application developers are your users. Apply product thinking to infrastructure.

An Internal Developer Platform (IDP) provides self-service capabilities that application teams use to build, deploy, and operate their applications without filing tickets or waiting for infrastructure provisioning. The Platform Engineer builds and maintains the IDP.

IDP components for MSS:

Component	Purpose	Consumer
Service catalog	List of available platform services (databases, caches, message queues)	Application teams
Golden paths	Opinionated, pre-configured project templates that embed best practices	New projects / SL 4L SWEs
CI/CD pipelines	Automated build → test → scan → deploy workflows	All code-producing tracks
Environment provisioning	Self-service creation of dev/staging/prod environments	Application teams
Observability stack	Logging, metrics, tracing, alerting	All tracks (consumers); SL 4O (operators)
Documentation portal	Platform usage guides, API references, runbooks	All tracks

2-2. Developer Experience (DevEx)

BLUF: If the platform is hard to use, developers will work around it. Workarounds create shadow infrastructure, security gaps, and operational risk.

DevEx principles for MSS platform engineering:

1. **Self-service over tickets:** Developers should be able to provision environments, run pipelines, and access logs without filing a request.

2. **Fast feedback loops:** Build times under 5 minutes. Pipeline failures surface within 10 minutes of commit. Test results available before merge.
3. **Safe defaults:** New projects start with security scanning, linting, and test requirements pre-configured. Opt out is deliberate, not accidental.
4. **Minimal cognitive load:** Developers should not need to understand Kubernetes internals to deploy an application. Abstract complexity behind well-designed interfaces.

CHAPTER 3 — KUBERNETES FOR MSS

3-1. Cluster Architecture

BLUF: Kubernetes is the compute substrate for MSS platform services. Understand the cluster topology, resource model, and scheduling behavior before deploying workloads.

MSS cluster architecture considerations:

Component	Purpose	Sizing Consideration
Control plane	API server, scheduler, controller manager, etcd	High availability (3+ nodes); etcd backup schedule
Worker nodes	Run application workloads as pods	Size based on workload profile (CPU-bound vs. memory-bound)
Namespaces	Logical isolation for teams/environments	One namespace per application per environment
Network policies	Control pod-to-pod and pod-to-external communication	Default deny; explicitly allow required traffic
Storage classes	Persistent volume provisioning	Match storage class to workload requirements (IOPS, capacity, durability)

3-2. Workload Management

BLUF: Deploy workloads declaratively. If it is not in version control, it does not exist.

Workload resource types:

Resource	Use Case	MSS Application
Deployment	Stateless applications (web servers, APIs)	Most MSS application components

Resource	Use Case	MSS Application
StatefulSet	Stateful applications (databases, message brokers)	Platform services requiring persistent identity
DaemonSet	Node-level agents (log collectors, monitoring)	Observability agents on every node
Job / CronJob	Batch processing, scheduled tasks	Data pipeline triggers, maintenance tasks

3-3. Resource Management and Quotas

BLUF: Without resource limits, one runaway workload can starve every application on the cluster. Set limits. Enforce quotas. Monitor usage.

Every workload deployment must specify: - `requests`: minimum resources guaranteed to the pod - `limits`: maximum resources the pod can consume - Resource quotas per namespace prevent any single team from consuming disproportionate cluster resources

CHAPTER 4 — INFRASTRUCTURE AS CODE (IAC)

4-1. Declarative Configuration

BLUF: Infrastructure is code. It lives in version control, goes through code review, and is deployed through pipelines — not by SSH-ing into a server and running commands.

IaC principles for MSS:

1. **Declarative over imperative:** Describe the desired state; let the tooling reconcile actual to desired. Do not write scripts that execute sequential steps.
2. **Version controlled:** All infrastructure configuration in Git. Every change has a commit, an author, and a review.
3. **Idempotent:** Applying the same configuration twice produces the same result. No side effects from re-runs.
4. **Environment parity:** Dev, staging, and production use the same templates with environment-specific parameters. Drift between environments is a deployment risk.

4-2. GitOps Workflows

BLUF: Git is the single source of truth for infrastructure state. Changes flow through pull requests, not manual commands.

GitOps workflow for MSS:

```
Developer commits → PR created → Peer review → Merge to main
                                                    |
                                              GitOps controller detects change
                                                    |
                                              Reconciles cluster state to match Git
                                                    |
                                              Drift detection alerts if manual changes
found
```

GitOps principles: - The Git repository is the source of truth for the desired state of the system - All changes to the system are made through Git commits (no `kubectl apply` from laptops) - A GitOps controller (e.g., kapp-controller, Flux, ArgoCD) continuously reconciles actual state to desired state - Drift detection alerts when actual state diverges from Git (indicates manual intervention or configuration drift)

4-3. Configuration Templating

BLUF: Templating separates configuration structure from environment-specific values. One template, multiple environments.

Use templating tools (ytt, Helm, Kustomize) to: - Define base configurations that are shared across environments - Overlay environment-specific values (replica counts, resource limits, feature flags) - Validate configuration before deployment (schema validation, policy checks) - Generate final manifests that are auditable and diffable

CHAPTER 5 — CONTAINER SECURITY AND SUPPLY CHAIN

5-1. DoD Container Hardening

BLUF: Every container image in production must be hardened, scanned, and traceable. No exceptions.

Container security requirements for MSS:

Requirement	Implementation
Base images from approved registry	Use Iron Bank (DoD-hardened) images; no pulling from Docker Hub in production
Vulnerability scanning	Scan every image at build time AND before deployment; block deployment if critical/high CVEs present
Image signing	Sign images with a trusted key; verify signature before deployment

Requirement	Implementation
Digest pinning	Reference images by SHA256 digest, not tag; tags are mutable, digests are not
Minimal images	No shells, package managers, or debugging tools in production images; use multi-stage builds
Non-root execution	All containers run as non-root user; drop all capabilities except those explicitly needed

5-2. Supply Chain Security

BLUF: The software supply chain extends from the developer's workstation through every dependency, build tool, and registry to the production cluster. Compromise at any point compromises the output.

Supply chain controls:

1. **Dependency management:** Pin all dependencies to exact versions. Audit transitive dependencies. Monitor for known vulnerabilities.
2. **Build environment:** Build pipelines run in ephemeral, isolated environments. No persistent state between builds.
3. **Artifact provenance:** Every artifact (container image, binary, configuration bundle) has a traceable lineage from source commit to production deployment.
4. **Registry security:** Container registry access controlled by role. Pull and push permissions separated. Audit log enabled.

CHAPTER 6 — CI/CD PIPELINE DESIGN

6-1. Pipeline Architecture

BLUF: The CI/CD pipeline is the factory floor. It turns code into deployed, running, monitored applications — automatically, repeatedly, and securely.

MSS CI/CD pipeline stages:

```
SOURCE → BUILD → TEST → SCAN → PACKAGE → DEPLOY → VERIFY → MONITOR
  |        |       |      |       |         |        |         |
  Git    Compile  Unit   SAST   Container  Target   Smoke    Alerts
  push   deps     Integ  DAST   image      env      tests    Metrics
         lint     E2E    SCA    sign/push  promote  Health   Logs
                         CVE               rollback  checks
```

6-2. Pipeline Security Gates

BLUF: Security is not a phase at the end — it is a gate at every stage. Shift left: catch issues as early in the pipeline as possible.

Security gates by pipeline stage:

Stage	Gate	Block Deploy?
Source	Secrets detection (pre-commit hook)	Yes — hard block
Build	Dependency vulnerability scan (SCA)	Yes if critical/high CVE
Test	SAST (static analysis)	Yes if high-severity finding
Scan	Container image vulnerability scan	Yes if critical/high CVE
Scan	DAST (dynamic analysis) on staging	Advisory (block for critical)
Package	Image signature verification	Yes — hard block
Deploy	Policy admission (OPA/Gatekeeper)	Yes — hard block

6-3. Deployment Strategies

BLUF: Not every deployment needs to be all-or-nothing. Choose the strategy that matches the risk level of the change.

Strategy	How It Works	When to Use	Rollback Speed
Rolling update	Replace pods incrementally	Standard releases; low-risk changes	Fast (revert deployment)
Blue/green	Run two environments; switch traffic	High-risk changes; need instant rollback	Instant (switch back)
Canary	Route small % of traffic to new version	New features with uncertain impact	Fast (route 100% to old)
Recreate	Terminate all old, start all new	Incompatible schema changes	Slow (full redeploy)

DRAFT

CHAPTER 7 — COMPUTE MODULES

7-1. Compute Modules Overview

BLUF: Compute Modules allow platform engineers to deploy and run containerized workloads directly on Foundry infrastructure without provisioning external Kubernetes clusters. GA as of February 2026, Compute Modules eliminate the need to stand up and maintain separate compute environments for workloads that can run within the Foundry ecosystem.

Task: Deploy and operate containerized workloads using Compute Modules on Foundry infrastructure.

Conditions: Given access to a Foundry environment with Compute Modules enabled, a container image stored in an approved registry, and a defined resource allocation.

Standards: - Container deploys successfully and reaches a healthy running state within the target environment - Resource requests and limits are configured per workload requirements - Workload is observable through Foundry-native monitoring and the History tab - Scaling configuration matches operational demand patterns

What Compute Modules are:

Compute Modules are a Foundry-native capability for deploying and running containers directly on Foundry infrastructure. Rather than provisioning and managing external Kubernetes clusters, platform engineers define container workloads within Foundry and let the platform handle scheduling, networking, and lifecycle management. This reduces operational overhead for workloads that do not require full cluster autonomy.

Key characteristics:

Characteristic	Description
Container-native	Deploy standard OCI-compliant container images
Foundry-managed infrastructure	No cluster provisioning, node management, or control plane maintenance
Integrated observability	Replica history, run logs, and change tracking built in
Scheduled scaling	Time-based replica scaling with override windows
Auditability	Full history of changes, deployments, and replica state

When to use Compute Modules vs. external Kubernetes:

Factor	Compute Modules	External Kubernetes
Workload scope	Single-purpose containers that serve Foundry workflows	Complex multi-service applications requiring custom networking, service mesh, or stateful orchestration

Factor	Compute Modules	External Kubernetes
Operational overhead	Low — Foundry manages infrastructure	High — team manages cluster lifecycle
Customization	Foundry-defined resource and networking model	Full control over cluster configuration
Air-gapped/edge	Requires Foundry connectivity	Can operate independently
Compliance posture	Inherits Foundry's ATO boundary	Separate ATO boundary for each cluster

NOTE

Compute Modules do not replace Kubernetes for all workloads. Workloads requiring custom networking policies, service mesh, persistent volumes with specific IOPS guarantees, or autonomous edge operation still require dedicated cluster infrastructure (see Chapters 3 and 8). Compute Modules are the right choice when the workload fits within Foundry's managed compute model and the operational simplicity outweighs the reduced customization.

7-2. Deploying Containerized Workloads

BLUF: Deploying a container on Compute Modules follows a repeatable sequence: define the image, configure resources, set environment variables, and deploy. The platform handles scheduling and networking.

Task: Deploy a containerized workload to a Compute Module.

Conditions: Given an approved container image (Iron Bank or organization-approved registry), defined resource requirements, and appropriate Foundry project permissions.

Standards: - Image referenced by SHA256 digest (not mutable tag) - Resource requests and limits explicitly configured - Environment variables and secrets injected through Foundry's configuration interface (not baked into the image) - Deployment verified healthy before routing traffic or enabling downstream dependencies

Deployment procedure:

1. **Select or create the Compute Module** within the target Foundry project. Name it according to the organization's naming convention.
2. **Specify the container image.** Use the full image reference including registry path and SHA256 digest. Do not use `:latest` or other mutable tags — digest pinning prevents silent image changes between deployments.

3. **Configure resource allocation.** Set CPU and memory requests and limits based on workload profiling. Over-provisioning wastes shared capacity; under-provisioning causes OOMKilled events and degraded performance.
4. **Set environment variables and secrets.** Use Foundry's configuration management to inject runtime configuration. Secrets must come from Foundry's secrets integration — never embed credentials in the container image or environment variable definitions visible in version control.
5. **Configure networking.** Define exposed ports and any required ingress rules. Foundry manages internal networking; the platform engineer configures what is exposed and to whom.
6. **Deploy and verify.** Initiate the deployment and monitor replica status. Verify the workload reaches a healthy state by checking container logs and health endpoints.

> **CAUTION**
>
> Container images deployed through Compute Modules must meet the same security standards as any other production container on MSS. Iron Bank base images, vulnerability scanning, digest pinning, non-root execution — all requirements from Chapter 5 apply. Compute Modules simplify infrastructure management; they do not relax security standards.

7-3. Replica History and Debugging

BLUF: Compute Modules maintain a full history of every replica that has run, including start/stop times, exit codes, and logs. Use replica history as the first-line debugging and auditing tool.

Task: Access and interpret replica history for debugging failed or degraded workloads.

Conditions: Given a deployed Compute Module with one or more historical replica runs.

Standards: - Platform engineer can identify failed replicas, their exit codes, and the time window of failure - Root cause determination uses replica logs and state transitions, not guesswork - Findings are documented in the incident record or maintenance log

Accessing replica history:

Replica history is available within the Compute Module's management interface. Each replica entry includes:

Field	Purpose
Replica ID	Unique identifier for the specific replica instance
Start time	When the replica was scheduled and started
Stop time	When the replica terminated (if applicable)
Exit code	Process exit code (0 = clean shutdown; non-zero = error)
Status	Current state (running, succeeded, failed, terminated)

Field	Purpose
Logs	Container stdout/stderr output for the replica's lifetime

Debugging workflow:

1. **Identify the failure window.** Correlate user-reported issues or alert timestamps with replica history to find which replica(s) were running during the incident.
2. **Check exit codes.** A non-zero exit code indicates the container process terminated abnormally. Common patterns: exit code 137 (OOMKilled — increase memory limits), exit code 1 (application error — check logs).
3. **Review replica logs.** Examine stdout/stderr for error messages, stack traces, and application-level diagnostics. If logging is structured (JSON), filter by severity or component.
4. **Compare with previous successful runs.** Identify what changed between the last successful replica and the failed one — configuration change, image update, or environmental factor.
5. **Cross-reference with the History tab** (see Section 7-5) to determine whether a configuration change immediately preceded the failure.

Auditing use case: Replica history provides a tamper-evident record of what ran, when, and with what result. This record supports ATO continuous monitoring requirements by demonstrating that only approved containers executed on the platform and that anomalous terminations were investigated.

7-4. Scheduled Replica Scaling

BLUF: Compute Modules support time-based replica scaling with override windows. Use scheduled scaling to match capacity to predictable demand patterns without manual intervention.

Task: Configure scheduled replica scaling with time-based overrides for a Compute Module.

Conditions: Given a deployed Compute Module with a known demand pattern (e.g., higher usage during duty hours, surge during exercises).

Standards: - Baseline replica count matches off-peak demand - Scheduled scale-up aligns with known high-demand periods - Override windows are documented with justification - Scaling changes are verified through replica history

Scaling configuration:

Parameter	Description
Baseline replicas	Default replica count during normal operations
Scheduled scale-up	Time-based rules that increase replica count during defined windows (e.g., 0600–1800 CEST duty hours)

Parameter	Description
Scheduled scale-down	Time-based rules that reduce replicas during low-demand periods
Override window	Temporary scaling rule that supersedes the schedule (e.g., exercise surge from D-3 to D+5)

Configuration guidance:

1. **Establish baseline.** Use historical metrics to determine the minimum replica count that sustains acceptable performance during off-peak hours.
2. **Define scheduled windows.** Map known demand patterns to scaling rules. For MSS, typical patterns include: duty-hour scaling (CEST business hours), exercise surge periods, and reporting deadlines.
3. **Set override windows for exercises and operations.** When an exercise or operational surge is planned, create a time-bounded override that increases capacity for the duration. Document the override in the operation order or FRAGO as an infrastructure preparation task.
4. **Monitor and adjust.** After each scaling event, review replica history and performance metrics. If scale-up was insufficient or scale-down caused degradation, adjust thresholds.

> **NOTE**
>
> Scheduled scaling addresses predictable demand. For unpredictable spikes, coordinate with Foundry platform administration on autoscaling capabilities or maintain headroom in the baseline replica count. Do not rely on reactive scaling alone for mission-critical workloads.

7-5. History Tab — Viewing Changes and Past Runs

BLUF: The History tab provides a chronological audit trail of every configuration change, deployment, and scaling event for a Compute Module. Use it to answer the question: "What changed, and when?"

Task: Use the History tab to audit configuration changes and correlate them with operational events.

Conditions: Given a Compute Module with deployment history.

Standards: - Platform engineer can trace any current configuration to the change that set it - Configuration changes are correlated with operational outcomes (success/failure) - Audit evidence is available for RMF/ATO continuous monitoring

What the History tab captures:

Event Type	Details Recorded
Configuration change	What changed (image, environment variables, resource limits, scaling rules), who changed it, when

Event Type	Details Recorded
Deployment	Image deployed, replica count, deployment outcome
Scaling event	Scheduled or manual scaling action, previous and new replica count
Replica lifecycle	Replica start/stop events, exit codes, duration

Operational use of the History tab:

- **Incident investigation.** When a workload degrades, check the History tab first. If a configuration change occurred immediately before the degradation, it is the likely cause. This follows the "what changed?" principle — most incidents are caused by changes, not spontaneous failures.
- **Change verification.** After applying a configuration change, verify in the History tab that the change was applied as intended. Do not assume — verify.
- **Compliance evidence.** The History tab provides the audit trail that RMF requires: who changed what, when, and what the system state was before and after. Export or screenshot this data during ATO evidence collection.
- **Knowledge transfer.** When a platform engineer transitions responsibility for a Compute Module, the History tab provides context that runbooks alone cannot capture — the operational history of the workload.

7-6. CI/CD Integration

BLUF: Compute Modules integrate into existing CI/CD pipeline patterns. The deployment target changes from a Kubernetes cluster to a Compute Module; the pipeline stages (build, test, scan, package, deploy) remain the same.

Task: Integrate Compute Module deployment into an existing CI/CD pipeline.

Conditions: Given an existing CI/CD pipeline (Chapter 6) and a target Compute Module.

Standards: - Pipeline deploys to the Compute Module without manual intervention - All security gates (Chapter 6, Section 6-2) are enforced before deployment - Deployment is verified via automated health checks - Rollback capability is maintained

Integration pattern:

```
SOURCE → BUILD → TEST → SCAN → PACKAGE → DEPLOY TO COMPUTE MODULE → VERIFY
  |        |       |       |       |              |                      |
  Git    Compile  Unit   SAST   Container    Foundry API /            Replica
  push   deps     Integ  DAST   image        CLI push to             health
         lint     E2E    SCA    sign/push    Compute Module           check
                         CVE
```

The pipeline stages from Chapter 6 (Section 6-1) remain unchanged through the PACKAGE stage. The DEPLOY stage targets the Compute Module through Foundry's API or CLI tooling instead of `kubectl apply` against a Kubernetes cluster.

Implementation considerations:

1. **Deployment automation.** Use Foundry's API or CLI to update the Compute Module's container image reference as part of the pipeline's deploy stage. The pipeline should pass the new image digest (not tag) to the Compute Module configuration.
2. **Health verification.** After deployment, the pipeline should poll the Compute Module's replica status until the new replica reaches a healthy state or a timeout threshold is exceeded. If the health check fails, the pipeline should trigger a rollback to the previous image digest.
3. **Rollback procedure.** Maintain the previous known-good image digest. If the newly deployed image fails health checks, the pipeline automatically reverts the Compute Module to the previous digest. This mirrors the rolling update strategy from Chapter 6 (Section 6-3).
4. **Security gate enforcement.** All security gates from Chapter 6 (Section 6-2) apply. The container image must pass vulnerability scanning, SAST, and SCA checks before the pipeline proceeds to the deploy stage. Compute Modules do not bypass security requirements — they change the deployment target, not the security posture.
5. **GitOps compatibility.** If the team uses a GitOps workflow (Chapter 4, Section 4-2), the Compute Module configuration can be stored in Git alongside other infrastructure definitions. The GitOps controller reconciles the Compute Module's desired state from the repository, maintaining the same change control discipline as cluster-based deployments.

> **NOTE**
>
> Teams transitioning workloads from self-managed Kubernetes deployments to Compute Modules should update their pipeline configurations incrementally. Run both deployment targets in parallel during transition to verify parity before decommissioning the cluster-based deployment.

CHAPTER 8 — AIR-GAPPED AND DDIL OPERATIONS

8-1. Air-Gapped Deployment

BLUF: MSS operates across classification boundaries. Platform Engineers must deploy software to environments with no internet connectivity. Every external dependency must be pre-packaged.

Air-gapped deployment requirements:

1. **Bundle everything:** Container images, configuration, dependencies, and tooling packaged into transferable bundles (e.g., imgpkg bundles). Nothing downloads at deploy time.
2. **Internal registries:** Each air-gapped environment runs its own container registry. Images are transferred via approved media (e.g., data diode, burned media, cross-domain solution).
3. **Dependency mirroring:** Language package registries (npm, PyPI, Maven) mirrored internally. Build pipelines reference internal mirrors, not public registries.
4. **Certificate management:** Internal PKI for TLS within the air-gapped environment. Certificate rotation procedures documented and tested.

8-2. DDIL-Resilient Infrastructure

BLUF: USAREUR-AF operates across the European AOR where network connectivity ranges from fiber to satellite to nothing. Design infrastructure that degrades gracefully, not catastrophically.

DDIL design principles:

Principle	Implementation
Autonomy over connectivity	Edge clusters can operate independently when disconnected from hub
Eventual consistency	Data synchronizes when connectivity restores; conflict resolution strategy defined
Local caching	Critical services cache data locally; stale-while-revalidate patterns
Bandwidth awareness	Sync operations prioritize by mission criticality; bulk transfers scheduled during high-bandwidth windows
Health reporting	Edge clusters report health status when connected; alerts trigger on missed check-ins

8-3. Edge Cluster Management

BLUF: When you manage clusters at the edge — forward deployed, low bandwidth, possibly disconnected — traditional cluster management breaks down. Plan for autonomous operation.

Edge cluster considerations: - Minimal control plane footprint (single-node or 3-node clusters) - Pre-staged workloads that can operate without hub connectivity - Automated recovery procedures that do not require remote access - Update strategy: staged rollout from hub to edge with verification gates - Data sovereignty: some data must remain at the edge and not sync to hub

CHAPTER 9 — OBSERVABILITY

9-1. The Three Pillars: Metrics, Logs, Traces

BLUF: If you cannot observe the platform, you cannot operate it. Observability is not optional tooling — it is the difference between diagnosing an incident in minutes and guessing for hours.

Observability rests on three pillars. Each serves a distinct purpose; all three are required for a production Kubernetes platform.

Pillar	What It Captures	Example in MSS Context
Metrics	Numeric measurements over time (counters, gauges, histograms)	Pod CPU utilization, API request latency (p99), node memory pressure, pipeline build duration
Logs	Discrete, timestamped event records	Application error messages, audit trail entries, container stdout/stderr
Traces	End-to-end request flow across services	A user action in Workshop → API gateway → Ontology query → data source, with latency at each hop

Metrics tell you *something is wrong*. Logs tell you *what happened*. Traces tell you *where in the chain it happened*. A platform that collects only one or two pillars leaves blind spots that extend incident resolution time and increase operational risk.

9-2. What to Monitor in a Kubernetes Environment

BLUF: Monitor at every layer — cluster infrastructure, platform services, and application workloads. A healthy node does not guarantee a healthy application.

Cluster-level monitoring (infrastructure): - Node CPU, memory, disk pressure, and network I/O - etcd health, leader elections, and write latency - Kubernetes API server request latency and error rate - Pod scheduling failures, pending pods, and evictions - Persistent volume capacity and IOPS utilization

Platform service monitoring: - CI/CD pipeline success rate, build duration, and queue depth - Container registry availability, pull latency, and storage consumption - GitOps controller reconciliation status and drift alerts - Certificate expiration dates (alert at 30, 14, and 7 days)

Application workload monitoring: - Pod restart counts and OOMKilled events - HTTP response codes (alert on sustained 5xx rates) - Request latency by percentile (p50, p95, p99) - Resource utilization vs. requests/limits (identify over-provisioned or starved workloads)

9-3. Alerting

BLUF: An alert that fires constantly is ignored. An alert that never fires is untested. Design alerts that are actionable, routed to the right responder, and tied to a runbook.

Alerting principles for MSS platform operations:

1. **Alert on symptoms, not causes.** Alert on "API error rate exceeded 5% for 10 minutes," not on "pod restarted." The symptom tells the operator what users experience; the cause is what the operator investigates.
2. **Severity tiers.** Define severity levels and route accordingly:
3. **Critical:** User-facing impact NOW. Pages on-call. Example: cluster API server unreachable.
4. **Warning:** Degradation likely within hours. Notifies team channel. Example: node disk at 85%.
5. **Info:** Anomaly worth tracking. Logged, not paged. Example: build duration increased 20%.
6. **Every alert has a runbook.** The alert message links to a procedure. On-call personnel should not need to reverse-engineer the response during an incident.
7. **Suppress flapping.** Use appropriate evaluation windows and hysteresis to prevent alerts from firing and clearing repeatedly during transient spikes.

9-4. Observability and Operational Readiness

BLUF: A platform without observability is not operationally ready. Observability provides the evidence base for readiness assessments, capacity planning, and ATO continuous monitoring.

Observability directly supports MSS operational readiness in three ways:

- **Incident response.** When an application fails during an operational event, the Platform Engineer must isolate whether the issue is infrastructure (node failure, network partition), platform (pipeline stuck, registry unreachable), or application (code bug, resource exhaustion). Correlated metrics, logs, and traces across layers make this determination possible in minutes rather than hours.
- **Capacity planning.** Historical metrics drive resource forecasting. When a new exercise or operational surge is planned, the Platform Engineer uses utilization trends to determine whether current cluster capacity is sufficient or additional nodes must be provisioned.
- **Continuous monitoring for ATO.** RMF requires ongoing evidence that security controls remain effective. Observability tools generate this evidence automatically — access logs, scan results, configuration drift alerts — reducing the manual burden on the ISSM and Platform Engineer.

> **NOTE**
>
> Observability data itself is operationally sensitive. Metrics, logs, and traces may reveal system architecture, user behavior patterns, and security posture details. Apply the same access controls and classification handling to observability data as to the systems it monitors.

CHAPTER 10 — PLATFORM SECURITY AND COMPLIANCE

10-1. RMF/ATO from the Platform Perspective

BLUF: The Risk Management Framework (RMF) and Authority to Operate (ATO) are not paperwork exercises — they are the legal authorization for MSS to process operational data. Platform Engineers generate the evidence that sustains the ATO.

Platform Engineer ATO responsibilities:

Artifact	Platform Engineer Role
System Security Plan (SSP)	Provide accurate architecture diagrams, data flow documentation, boundary definitions
Security controls evidence	Generate automated evidence: scan results, patch status, configuration baselines, access logs
Continuous monitoring	Operate monitoring tools that produce the data ISSM needs for ongoing authorization
POA&M items	Remediate platform-level findings; provide timelines and evidence of closure
Incident response	Execute platform-level IR procedures; preserve evidence; support forensics

10-2. STIG Compliance

BLUF: Security Technical Implementation Guides (STIGs) define the hardening standard for DoD systems. Platform Engineers automate STIG compliance — manual STIG checks do not scale.

- Automate STIG compliance checking as part of the CI/CD pipeline
- Use hardened base images that are pre-STIGged (Iron Bank)
- Document exceptions/waivers for STIG findings that cannot be remediated (with ISSM approval)
- Re-scan after every platform change; compliance is continuous, not point-in-time

10-3. Access Control and Audit

BLUF: Platform-level access is god-mode access. Control it accordingly.

- **Least privilege:** Platform engineers get the minimum permissions needed for their current task. No standing admin access.
- **Just-in-time access:** Elevated permissions granted for a specific task with automatic expiration.

- **Audit logging:** Every administrative action logged, timestamped, and attributed to a specific individual. Logs are immutable and forwarded to a separate system.
- **Break-glass procedures:** Documented emergency access procedures for when normal access paths are unavailable. Break-glass use triggers automatic review.

10-4. Secrets Management

BLUF: Hardcoded secrets in source code, container images, or pipeline configurations are unacceptable. A single leaked credential can compromise the entire platform. Manage secrets as first-class infrastructure objects with encryption, access control, and rotation.

Why hardcoded secrets are a critical risk: Secrets committed to Git persist in repository history even after deletion. Container images with embedded credentials expose those credentials to anyone with pull access. Environment variables in pipeline definitions are visible in build logs. Every instance of a hardcoded secret is a latent incident waiting for discovery by an adversary.

Secrets management patterns for Kubernetes:

Pattern	How It Works	When to Use
Sealed Secrets	Encrypt secrets client-side using a cluster-specific public key; only the controller in-cluster can decrypt. Encrypted secrets are safe to store in Git.	GitOps workflows where secrets must live in version control alongside other manifests
External Secrets Operator	Controller syncs secrets from an external secrets store (Vault, AWS Secrets Manager, Azure Key Vault) into Kubernetes Secret objects. Applications consume native K8s Secrets.	Environments with an established enterprise secrets store; multi-cluster deployments sharing a central vault
Vault integration (sidecar/CSI)	HashiCorp Vault provides secrets via sidecar injection or the Secrets Store CSI Driver. Applications retrieve secrets at runtime, never stored as K8s Secret objects.	High-security workloads requiring dynamic secrets, lease-based access, and fine-grained audit trails

Secret rotation: - All platform credentials (database passwords, API keys, service account tokens, TLS certificates) must have defined rotation schedules. - Automate rotation where possible. Manual rotation introduces human error and schedule drift. - Design applications and services to handle credential rotation without downtime — support dual-credential windows where the old and new credential are both valid during transition. - Log and alert on rotation failures. A failed rotation is a security event.

Operational controls: - Encrypt Kubernetes Secrets at rest (etcd encryption). Default K8s Secrets are base64-encoded, NOT encrypted. - Restrict Secret access via RBAC. Only the pods and service accounts that need a secret should be able to read it. - Audit secret access. Enable Kubernetes audit logging for all Secret read/list operations. - Never log secret values. Ensure application logging frameworks redact or mask sensitive fields.

WARNING

Base64 encoding is NOT encryption. A Kubernetes Secret stored without etcd encryption is readable by anyone with etcd access or cluster-admin RBAC. Enable etcd encryption at rest and verify it is active — do not assume.

APPENDIX A — GLOSSARY

Term	Definition
Compute Module	Foundry-native capability for deploying and running containers on Foundry-managed infrastructure (GA Feb 2026)
IDP	Internal Developer Platform — self-service infrastructure for application teams
IaC	Infrastructure as Code — managing infrastructure through version-controlled configuration files
GitOps	Operating model where Git is the source of truth for infrastructure and application state
DDIL	Denied, Disrupted, Intermittent, and Limited bandwidth
Iron Bank	DoD-hardened container image repository
STIG	Security Technical Implementation Guide — DoD hardening standards
RMF	Risk Management Framework — DoD cybersecurity risk management process
ATO	Authority to Operate — formal authorization to operate an information system
ISSM	Information System Security Manager
SCA	Software Composition Analysis — scanning for vulnerabilities in dependencies
SAST	Static Application Security Testing
DAST	Dynamic Application Security Testing
OPA	Open Policy Agent — policy engine for Kubernetes admission control
CSI	Container Storage Interface — standard for exposing storage systems to K8s workloads
CReATE	Code Resource and Transformation Environment — ASF's Kubernetes-based DevSecOps platform
SLO/SLI	Service Level Objective / Indicator — reliability targets and the metrics that measure them
WFF	Warfighting Function

DRAFT

Term	Definition
AOR	Area of Responsibility

APPENDIX B — REFERENCES

Reference	Relevance
SL 4L — Software Engineer	Primary platform consumer; CI/CD pipeline user
SL 4N — UI/UX Designer	Application design constraints from platform capabilities
SL 4J — Program Manager	Release planning, deployment scheduling
SL 5L — Advanced Software Engineer, Chapter 6	DevSecOps from the application perspective (companion reading)
SL 5O — Advanced Platform Engineer	Advanced topics: fleet management, SRE, RMF automation
DoD Software Modernization Strategy (Feb 2022)	Strategic context for DevSecOps and platform engineering
NIST SP 800-37 (RMF)	Risk Management Framework
DISA STIGs	Security hardening standards
CReATE / Carvel case study	ASF platform engineering reference implementation

DRAFT