CONCEPTS GUIDE

SL 4O

USAREUR-AF

CONCEPTS GUIDE — SL 4O COMPANION — PLATFORM ENGINEER · MAVEN SMART SYSTEM (MSS)

Specialist Course Manual

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

26 MARCH 2026

DRAFT — UNOFFICIAL — NOT FOR OPERATIONAL USE

CONCEPTS GUIDE — SL 4O COMPANION — PLATFORM ENGINEER · MAVEN SMART SYSTEM (MSS)

Forward: The Platform Engineer builds the floor that every other track stands on. When the platform works, nobody notices. When it fails, everybody stops. **Purpose:** Develops mental models required to design, operate, and secure infrastructure for MSS. Read before beginning SL 4O task instruction. **Prereqs:** SL 3 REQUIRED (and SL 1 + SL 2 implied). Kubernetes and Linux administration experience recommended. *HQ USAREUR-AF · v1.0 · 2026 · DISTRIB: USG only*

SECTION 1 — THE PLATFORM ENGINEER'S ROLE ON MSS

BLUF: The Platform Engineer builds the floor that every other track stands on. When the platform works, nobody notices. When it fails, everybody stops.

MSS workforce tiers (infrastructure perspective):

Tier	Designation	Platform Relationship
SL 1	Maven User	End user — consumes applications running on the platform
SL 2	Builder	Builds on Workshop — platform provides the runtime
SL 3	Advanced Builder	Builds pipelines — platform provides compute and orchestration
SL 4L	Software Engineer	Writes application code — platform provides CI/CD, deployment, runtime
SL 4N	UI/UX Designer	Designs applications — platform defines performance/deployment constraints
SL 4H/M	AI/ML Engineer	Trains and deploys models — platform provides GPU compute, serving infrastructure
SL 4O	**Platform Engineer**	**Builds and operates the infrastructure all others depend on**

The Platform Engineer's distinguishing criterion: you own the systems that other people's code runs on. When a SWE's application fails, the SWE fixes the application. When the platform fails, every application fails. Platform failures are simultaneous, cross-cutting, and high-blast-radius.

Platform Engineer boundaries (owned by others): - Application code — SL 4L - Application design — SL 4N - Data modeling and pipeline logic — SL 3, SL 4H - ML model architecture — SL 4M - Foundry-native platform administration — Palantir (vendor-managed)

The boundary is clear: the Platform Engineer owns the infrastructure around Foundry, not Foundry itself. Kubernetes clusters, CI/CD pipelines, container registries, monitoring stacks, and the deployment toolchain — these are SL 4O territory.

SECTION 2 — PLATFORM-AS-PRODUCT MENTAL MODEL

BLUF: If you think of yourself as "the infrastructure person," you will build infrastructure. If you think of yourself as "the person whose product is the developer's experience," you will build a platform.

The product mindset shift:

Infrastructure Mindset	Platform-as-Product Mindset
"I configure servers"	"I build self-service capabilities for application teams"
"Developers submit tickets for what they need"	"Developers can provision what they need without a ticket"
"I optimize for system metrics (CPU, memory)"	"I optimize for developer productivity (deploy time, feedback speed)"
"Stability means nothing changes"	"Stability means changes are safe, fast, and reversible"
"Security is a gate at the end"	"Security is embedded in every stage"

Who is your user? The Platform Engineer's primary users are application developers (SL 4L, SL 3). Secondary users are data engineers (SL 4H, SL 4M) and the security team (ISSM). Design the platform to serve these users — measure success by their productivity, not by infrastructure uptime alone.

SECTION 3 — KUBERNETES MENTAL MODEL

BLUF: Kubernetes is a declarative system — you describe what you want, and the platform reconciles reality to match. Understand this contract or you will fight the system instead of using it.

The declarative contract:

```
You write:  "I want 3 replicas of this container, each with 512MB RAM"
K8s does:   Schedules pods, monitors health, restarts failures, scales as directed
```

```
You verify: Actual state matches desired state
```

Key mental models:

1. **Desired state vs. actual state:** You declare the desired state in YAML. Kubernetes controllers continuously work to make actual state match desired state. If a pod dies, the controller creates a new one. You do not restart pods — you declare how many should exist, and the system ensures it.
2. **Pods are cattle, not pets:** Pods are ephemeral. They are created, destroyed, and replaced automatically. Do not SSH into pods. Do not store state in pods. Do not give pods meaningful names. Design applications that survive pod replacement.
3. **Namespaces are boundaries:** Namespaces provide logical isolation — resource quotas, network policies, and RBAC are scoped to namespaces. One namespace per application per environment is the standard pattern.
4. **Labels are the API:** Kubernetes uses labels to select, group, and route to resources. Services find pods by label selectors. Network policies match by labels. Monitoring queries by labels. A consistent labeling strategy is foundational.

SECTION 4 — INFRASTRUCTURE AS CODE MENTAL MODEL

BLUF: If you made a change and it is not in Git, it did not happen. If it happened and it is not in Git, it is drift — and drift kills.

Why IaC matters for MSS:

Without IaC	With IaC
"Who changed the firewall rule?" — nobody knows	Every change has a commit, an author, and a review
"Can we reproduce this environment?" — probably not	Clone the repo, apply the config, identical environment
"What is different between staging and prod?" — SSH and compare	`git diff staging prod`
"We need to roll back" — to what?	`git revert` to the previous known-good state

The GitOps loop:

```
Git repo (desired state)
     ↓ push
GitOps controller (detects change)
```

```
     ↓ reconcile
Cluster (actual state converges to desired)
     ↓ monitor
Drift detection (alert if actual ≠ desired)
     ↓ investigate
Fix: either update Git (intentional change) or revert cluster (unauthorized drift)
```

Critical rule: The Git repository is the single source of truth. If someone `kubectl apply`s a change directly to the cluster without committing to Git, the GitOps controller will eventually revert that change. This is a feature, not a bug — it prevents configuration drift.

SECTION 5 — CONTAINER SECURITY MENTAL MODEL

BLUF: A container is not a security boundary — it is a packaging format. Security comes from what is inside the container, how it is built, and where it runs.

The supply chain attack surface:

```
Developer workstation → Source code → Dependencies → Build environment
                                                          ↓
                                    Container image → Registry → Cluster
```

Every step is an attack surface. The Platform Engineer's job is to secure each step:

1. **Source:** Code review, branch protection, signed commits
2. **Dependencies:** Pinned versions, vulnerability scanning, license compliance
3. **Build:** Ephemeral build environments, no cached credentials, reproducible builds
4. **Image:** Hardened base (Iron Bank), minimal attack surface, non-root, scanned
5. **Registry:** Access-controlled, audit-logged, signed images only
6. **Cluster:** Admission policies, network policies, runtime security monitoring

The Iron Bank principle: Start with a known-good, DoD-hardened base image. Add only what you need. Scan the result. Sign it. Pin it by digest. Never trust a tag — tags can be overwritten; digests cannot.

SECTION 6 — CI/CD PIPELINE MENTAL MODEL

BLUF: The pipeline is a trust machine. Code enters one end untrusted. Each stage adds a layer of verification. What exits the other end has been built, tested, scanned, signed, and deployed — or rejected.

The trust gradient:

```
UNTRUSTED                                                    TRUSTED
   |                                                            |
   Source → Build → Unit Test → SAST → SCA → Image Scan → Sign → Deploy
   |         |         |          |      |        |          |      |
   "I wrote  "It      "It does   "No    "No     "No CVEs   "It's   "It's
    code"    compiles"  what I    known   known   in the     proven   running"
                       intended"  vulns   vulns   image"     safe"
                                  in my   in my
                                  code"   deps"
```

Each stage gates the next. If any stage fails, the pipeline stops. The artifact does not advance. This is the mechanism that lets you deploy with confidence — not because you trust the developer, but because you trust the process.

SECTION 7 — AIR-GAPPED AND DDIL MENTAL MODEL

BLUF: Air-gapped environments are islands. Everything the island needs must be shipped there before it is needed. You cannot download a dependency during a deployment.

The island model:

```
CONNECTED ENVIRONMENT              AIR GAP              DISCONNECTED ENVIRONMENT
(hub / data center)                 |||                  (edge / classified)
                                    |||
  Build artifacts  ——— bundle —→  transfer  —→  unpack ——— deploy
  Container images                 media              Internal registry
  Config bundles                 (approved             Internal mirror
  Dependency mirrors              process)             Local CA/PKI
```

Key constraints: 1. **No runtime downloads:** Every container image, every npm package, every Python wheel must be pre-bundled. If the build needs it, it is in the bundle or it does not exist. 2. **Version skew:** The air-gapped environment may be days or weeks behind the connected environment. Design for graceful version differences. 3. **Transfer approval:** Moving data across the air gap requires process approval. You cannot "quickly push a fix" — plan for transfer windows.

DDIL vs. air-gapped: - Air-gapped = permanently disconnected; everything pre-staged - DDIL = intermittently connected; must handle both connected and disconnected states - DDIL is harder to design for because the system must gracefully transition between states

Design for DDIL: Assume disconnection is the default state. Connection is a bonus used for synchronization. If the system requires constant connectivity to function, it will fail in the AOR.