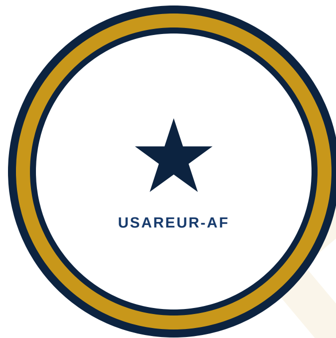
DRAFT — UNOFFICIAL — NOT FOR OPERATIONAL USE

CONCEPTS GUIDE

SL 4L



CONCEPTS GUIDE — SL 4L COMPANION — SOFTWARE ENGINEER · 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

CONCEPTS GUIDE — SL 4L COMPANION — SOFTWARE ENGINEER · MAVEN SMART SYSTEM (MSS)

Forward: The SWE is the engineer of record for production code on MSS. Everyone else depends on it running correctly, every time. **Purpose:** Develops mental models required to write, deploy, and maintain code on MSS. Read before beginning SL 4L task instruction. *HQ USAREUR-AF · v1.0 · 2026 · DISTRIB: USG only*

SECTION 1 — THE SOFTWARE ENGINEER'S ROLE ON MSS

BLUF: The SWE is the engineer of record for production code on MSS. Everyone else depends on it running correctly, every time.

MSS workforce tiers:

Tier	Designation	Primary Output	Code Level
SL 1	Maven User	Consumes data products, runs analyses	None
SL 2	Builder	Workshop apps, basic transforms, no-code pipelines	Minimal / drag-and-drop
SL 3	Advanced Builder	Complex pipelines, Contour datasets, data modeling	Light Python, no TypeScript
SL 4G	ORSA	Quantitative analysis, Commander products, statistical models	Python (analytical)
SL 4M	ML Engineer	ML models, validation, deployment	Python (ML)
SL 4H	AI Engineer	AIP Logic, Agents, LLM integration	Python + prompt engineering
SL 4L	**Software Engineer**	**Production transforms, TypeScript FOO, OSDK applications**	**Python + TypeScript (production)**

The SWE's distinguishing criterion: you write code that runs in production and that other people depend on. A SL 4G ORSA writes analytical Python; a SL 4M MLE writes model training code; a SL 4L SWE writes the infrastructure layer. When an ORSA notebook produces an incorrect result, the ORSA revises

DRAFT

it. When a production transform produces incorrect data, every analyst, every dashboard, and every staff section consuming that dataset works with bad information — and may not know it. SWE failures are silent, systemic, and operationally consequential.

SWE responsibilities on MSS: - Python transforms that execute on a schedule or trigger - TypeScript Functions on Objects (FOO) for computed Ontology properties - OSDK-backed applications for staff sections without direct Foundry access - Code repository governance: branch management, peer review, CI gates, promotion workflows - Technical coordination when builders and ORSAs need logic beyond their tier

SWE boundaries (owned by others): - Analysis notebooks — SL 4G - ML model hyperparameter tuning — SL 4M - AIP Agent chain design — SL 4H - Workshop application GUI builds — SL 3

The boundary is not about capability — it is about primary responsibility and production ownership.

NOTE — Army Data Plan SO 7 DevSecOps (TM-40L Section 1-5b): TM-40L Section 1-5b maps SWE responsibilities to Army Data Plan Strategic Objective 7 (DevSecOps), establishing the doctrinal basis for CI/CD pipeline standards, automated testing gates, and secure deployment practices on MSS.

NOTE — UDRA 6 Services Architecture (TM-40L Section 1-7): TM-40L Section 1-7 describes the UDRA six-services architecture (Discover, Access, Transport, Compute, Store, Govern) and how each service maps to SWE platform responsibilities. Reference when designing cross-service integrations.

NOTE — UDRA 15 Required Metadata Fields: SL 4L now documents the 15 metadata fields required by UDRA for all data assets. Ensure all production transforms and OSDK application outputs include compliant metadata before promotion.

SECTION 2 — THE FOUNDRY EXECUTION MODEL

BLUF: Transforms are not scripts. They are compute jobs called by the platform with typed inputs and a typed output. Understand this contract or your code will fail in production.

A script runs when you run it — file system, environment variables, network, local state all available. A Foundry transform is different: the platform calls it, provides the inputs, and receives the output. No local file system, no persistent state between runs, no side effects outside the defined output dataset. This design enables the platform to schedule, parallelize, cache, retry, and audit your code — but you must write code that fits the contract.

Execution contract:

```
Input datasets (provided by platform)
         |
         ▼
  [your transform function]
         |
```

```
        ▼
Output dataset (consumed by platform)
```

Your function receives DataFrames corresponding to declared inputs and returns a DataFrame. Everything else — environment variables, files, database connections, HTTP calls — is outside the contract and will fail in scheduled execution.

Script vs. transform behavior:

Pattern	In a script	In a Foundry transform
Read a file from disk	`open("data.csv")`	Not available. Data from input datasets only.
Write a file to disk	`open("out.csv", "w")`	Not available. Data to output datasets only.
Persist state between runs	Module-level variable	Not available. Use `@incremental` with watermarks.
Log to console	`print(...)`	Available; logs go to the platform build log.
Call an external API	`requests.get(...)`	Requires approved egress config. Not default.
Use a random seed	`random.random()`	Will differ each run unless seed is fixed.

Mental model: treat every transform as a pure function. Deviations that are operationally justified (reading a config parameter, calling an approved internal API) must be explicit, documented, and wrapped in error handling. Unjustified deviations are bugs.

Vignette — V Corps Equipment Readiness Transform

A SWE builds a scheduled transform that filters input by a date read from an environment variable. Works in local testing. In production, the environment variable is not set in the Foundry execution environment. The transform silently returns an empty DataFrame. Every downstream dashboard shows zero equipment at 0300. The fix: parameterize using Foundry dataset parameters or derive the filter from input data. The execution environment is the platform's, not yours.

SECTION 3 — IDEMPOTENCY AS A CORE DESIGN CONSTRAINT

BLUF: Same inputs, same logic, same outputs — regardless of how many times the transform has run. Foundry may re-run transforms on build, after failure, or on upstream changes. Non-idempotent transforms are operationally dangerous.

The idempotency test — ask before deploying any transform:

1. If this transform runs now and again in five minutes on the same input, are outputs identical?

2. If the output dataset is deleted and the transform re-runs from scratch, does it match what incremental updates would have produced?
3. If manually re-triggered after a pipeline failure, will there be duplicates, data loss, or corruption?

If the answer to any is "no" or "I'm not sure," redesign before production.

Common idempotency violations:

Violation	Description	Correct approach
Append without dedup	Re-runs create duplicates	Surrogate key + dedup on write, or `@incremental` with watermark
Timestamp-as-key	`datetime.now()` makes every run unique	Derive keys from input content, not execution time
Running counter	Counter increments each run	Store state in a dataset parameter or Foundry-managed watermark
Order-dependent logic	Assumes output was in a particular prior state	Reconstruct output from inputs; do not modify existing state

NOTE

APPEND transactions are not inherently idempotent. Re-runs produce duplicate rows without deduplication logic — typically via a content hash column and INSERT OR IGNORE pattern, or a surrogate key. For atomic full-dataset replacement, use SNAPSHOT transactions.

Incremental transforms corollary: The incremental output must be identical to what a full recompute would produce over the same data. Test explicitly: run incremental, wipe output, run full recompute, diff results. If they differ, there is an incremental logic bug. This test is not optional for transforms feeding readiness or operational data products.

Vignette — SITREP Aggregation Pipeline

An incremental SITREP aggregation transform uses a module-level Python `dict` that accumulates across runs. During a retry after build failure, the variable retains values from the failed run — double-counted submissions for two battalions. Correct design: every run reads only from declared input datasets, computes aggregations from scratch over the relevant window, writes complete consistent output. Module-level state is forbidden in production transforms.

SECTION 4 — THE ONTOLOGY AS YOUR API CONTRACT

BLUF: The Ontology is not a schema you own. It is the shared API contract for the entire MSS ecosystem. Treat changes with the same discipline as a breaking REST API version change.

Every Workshop query, Contour render, TypeScript FOO call, and OSDK application request goes through the Ontology. Adding a property extends the contract (non-breaking). Renaming, removing, or changing a property type breaks the contract — every application, module, and OSDK client referencing that property breaks.

Ontology change risk and required actions:

Change type	Risk	Required actions before change
Add new property	Low	Document in Object Type description; notify downstream builders
Rename property	**Breaking**	Audit all consumers (Workshop, Contour, FOO, OSDK); migrate in same branch; coordinate with data steward
Remove property	**Breaking**	Same as rename; confirm no active consumers before deletion
Change property type	**Breaking**	Same as rename; verify all ingestion transforms writing the property
Add new Object Type	Low	Document; coordinate naming with C2DAO governance standards
Add new Link Type	Low-medium	Audit if new link changes Object Set composition in Workshop
Change primary key	**Critical**	Full audit of all consumers; coordinate with C2DAO; treat as major version change

Consumer audit process — required before any breaking change:

1. Search all Workshop modules for references to the property or Object Type
2. Search all TypeScript Function code for references
3. Search all OSDK application code for references
4. Search all Contour datasets derived from the Object Type
5. Search all Python transforms that write to the Object Type

This is the SWE's responsibility, not the Ontology team's. Skipping this audit breaks production for users with no visibility into the change.

DRAFT

The branch/promote workflow provides a window to audit, migrate, and validate in dev before production impact. SWEs who bypass branch promotion under time pressure are the SWEs whose changes cause 0200 data outage calls.

Vignette — Equipment Object Type Property Rename

A SWE renames `eqStatus` to `equipmentReadinessStatus` on a Friday afternoon. Monday: G4 readiness dashboard is blank. A Workshop module and a TypeScript FOO function both reference `eqStatus` — both broken. A 10-minute consumer audit and a staged branch promotion would have caught both dependencies. Neither was done.

SECTION 5 — TYPESCRIPT FUNCTIONS — EXTENDING THE ONTOLOGY WITH LOGIC

BLUF: TypeScript Functions on Objects (FOO) add computed behavior to Ontology objects at read time. The primary design question: should this logic run on a schedule and store a result, or run on demand and compute live?

A FOO function executes when queried — not pre-computed. Every query that invokes it triggers a computation. Expensive FOO functions degrade query performance for all users of that Object Type.

Stored result vs. computed on read — decision framework:

Factor	Favor scheduled transform	Favor TypeScript FOO
Input data changes	On a schedule (daily, hourly)	Real time or continuously
Computation cost	High (expensive aggregation)	Low (simple arithmetic, lookup)
User freshness need	Scheduled freshness acceptable	Must reflect current state at query time
Consumer scope	Many consumers need the same result	Result is object-specific and contextual
Auditability required	Yes (stored result is auditable)	No (computed at read, no history)

FOO capabilities and limits:

Can do	Cannot do
Read properties of the called object instance	Write to the Ontology
Navigate Links to related objects	Make external API calls
Call other registered FOO functions	Access datasets directly
Return primitives, object refs, or arrays	Maintain state between invocations

Parameter design discipline: Minimum parameters needed to answer the business question. A FOO with five optional parameters and complex branching belongs in a transform.

Vignette — Equipment Readiness Calculation

Option A: Scheduled Python transform runs nightly, joins Equipment against Unit, computes readiness percentages, writes to `UnitReadiness` Object Type. Workshop queries `UnitReadiness` directly.

Option B: TypeScript FOO on `Unit` navigates `UNIT_HAS_EQUIPMENT` Link, iterates all Equipment objects, computes readiness on demand.

Option A is correct. Data changes daily at most, computation is moderately expensive, result must be auditable by G4. Option B recomputes on every Workshop query (performance degradation) and cannot support historical auditing. Option B is appropriate for real-time sensor data or calculations requiring current-moment precision.

SECTION 6 — OSDK — BUILDING APPLICATIONS AGAINST THE ONTOLOGY

BLUF: OSDK is the TypeScript API for applications that access MSS data from outside Foundry. Understand the Ontology first — OSDK is how you access it, not what you are accessing.

The OSDK is a generated TypeScript library exposing the MSS Ontology to external applications: V Corps G2 analytical portals, battalion maintenance trackers, readiness reporting tools in SharePoint. Staff sections without Foundry access consume MSS data through these OSDK-backed applications.

Mental model: An OSDK application is an Ontology client. It queries Object Types, navigates Links, and executes Actions defined in the Ontology. Foundry enforces access controls at the Ontology level — the OSDK application inherits whatever permissions the authenticated user or service account has on each Object Type and Action. The application cannot bypass Ontology restrictions. Design to work with Ontology permissions, not around them.

OSDK application architecture layers:

```
[ User / Staff Section ]
         │
         ▼
[ OSDK Application (TypeScript/React) ]
         │  queries Objects, navigates Links, calls Actions
         ▼
[ OSDK Client (generated TypeScript library) ]
         │  authenticated requests
         ▼
[ MSS Foundry Platform — Ontology API ]
         │  enforces CBAC, returns Objects
         ▼
```

DRAFT

```
[ Ontology Object Types, Links, Actions ]
          │  backed by
          ▼
[ Foundry Datasets (produced by transforms) ]
```

Each layer has distinct responsibility. Conflating layers — for example, writing directly to a dataset from the OSDK application layer rather than through an Action — bypasses the governance model.

Service account discipline:

Requirement	Standard
Storage	C2DAO-approved credential store only — never in code or config files
Scope	Least privilege — read access only to required Object Types
Rotation	Defined schedule; immediate rotation on suspected compromise
Audit	Log all service account activity

Service account credential violations are reportable security incidents, not coding convention failures.

Vignette — V Corps G3 Personnel Accountability Application

G3 needs a web application showing real-time personnel accountability. Personnel data lives in MSS as `Soldier` and `Unit` Object Types with `UNIT_HAS_SOLDIER` Links. The SWE builds a React frontend authenticating via individual SSO credentials. The application queries `Unit` objects and navigates `UNIT_HAS_SOLDIER` to retrieve accountability data.

Because access control lives at the Ontology level, staff members see only units they have been granted access to — no application-level permission logic needed. The Ontology is the permissions model.

The same application architecture scales to other MSCs — 10th AAMDC, 56th MDC-E, and SETAF-AF each deploy the identical React frontend against the same Ontology. Access control determines that a SETAF-AF personnel officer sees only SETAF-AF units, while a 10th AAMDC S1 sees only AAMDC formations. No MSC-specific code branches exist.

SECTION 7 — TESTING IN A PLATFORM ENVIRONMENT

BLUF: You cannot run Foundry locally, but you can test most transform logic locally with the right shim strategy. Every production transform must have at least one passing unit test before branch promotion.

Three-layer testing strategy:

Layer	Environment	What it catches
1 — Unit tests	Local, no Foundry	Logic errors, schema errors, error handling gaps

Layer	Environment	What it catches
2 — Integration tests	Foundry dev environment	Spark compatibility, API version issues, permission failures, live schema mismatches
3 — Pre-promotion smoke tests	Foundry branch	End-to-end behavior, downstream breakage

Layer 1 minimum bar for any production transform: - At least one test validating output schema (correct column names and types) - At least one test validating correct output for a known input - At least one test validating error handling for malformed input

TypeScript FOO testing pattern (Jest with local OSDK mock): 1. Construct a mock object instance with known property values 2. Call the FOO function against the mock 3. Assert returned value matches expected result

Cover edge cases: null/undefined properties, empty link sets, boundary values for numeric computations.

CI gate: The MSS CI pipeline runs unit tests on every pull request. A branch cannot be merged if tests fail. This gate is not optional and not bypassable. Tests that always pass without testing actual logic waste the CI gate and create false confidence.

Vignette — Readiness Transform Test Coverage

A transform computing unit readiness scores from equipment status codes is submitted for branch promotion without unit tests. The CI gate blocks the merge. Writing the unit test overnight, the SWE discovers the readiness scoring function assigns the wrong weight to `NMC-P` status — scores 8–12% higher than correct. The notebook had not tested this case. The CI gate forced the discovery before any commander received an inflated readiness product.

SECTION 8 — CODE AS OPERATIONAL INFRASTRUCTURE

BLUF: Your code will be maintained by someone else after you leave. Write for the next person. Unmaintainable production code is not technical debt — it is a readiness risk.

USAREUR-AF operates under rotation and PCS cycles. The SWE who built a transform in January may be reassigned by September. The contractor may transition in six months. Their replacement inherits your code with no tribal knowledge of your decisions, edge cases handled, or upstream data quirks worked around. When they cannot understand the code: either they do not touch it (risk calculates) or they change it incorrectly and break production. Both are operational failures.

Maintainability checklist — apply before every pull request:

Item	Standard
Variable names	Descriptive, unambiguous — `unit_readiness_score`, not `urs` or `x`
Function names	Verb + noun, describe the action — `calculate_equipment_availability`, not `calc_ea`
Inline comments	Present on every non-obvious logic block; military-specific business rules especially
Module docstring	Present at top of every Python module and TypeScript file — purpose, inputs, outputs, caveats
README	Present in every code repository — architecture, dependencies, local run instructions, promotion steps
No magic numbers	Constants named at module level — `NMC_STATUS_CODES = ["NMC-M", "NMC-P"]`, not inline literals
No clever tricks	If a line requires a comment explaining why it is written that way, rewrite it
Dependency documentation	All external dependencies in `requirements.txt` or `package.json` with pinned versions

Comments on military-specific logic: MSS contains business logic specific to Army operations — readiness status codes (FMC, PMC, NMC-M, NMC-P), personnel accountability categories, SITREP submission timelines, CCIR thresholds. These are not self-evident to developers from outside the military context. Comment them every time. `# NMC-P: Not Mission Capable - Parts (awaiting parts, can be returned to service)` takes 30 seconds to write and may save an hour of confusion.

Apply the Army Writing Style parallel: code read at 0200 during a production incident by someone who did not write it is exactly analogous to a field manual read in poor conditions by a non-SME. Write for that reader.

Vignette — The Undocumented Pipeline

A senior contractor SWE builds a complex equipment readiness pipeline over six months — working correctly, undocumented, terse variable names, PMC counting logic buried four conditionals deep. The contractor transitions off. A new SWE changes the code to add a status category, subtly altering NMC-P counting. The shift is within noise for most units. Three months later, G4 identifies a discrepancy between MSS readiness scores and paper records. The investigation traces back to the undocumented logic change. Forensic analysis costs three weeks.

SECTION 9 — COMMON SWE FAILURE MODES ON MSS

BLUF: The most common SWE failures are predictable and preventable. Recognize them before replicating them.

#	Failure Mode	Root Cause	Prevention
1	Transforms that depend on external state	File reads, API calls, env vars not available in Foundry execution environment	Declare all inputs through Foundry dataset mechanism; no external state
2	Building without tests	Notebook verified visually, promoted without CI coverage	Unit tests in CI before every merge; CI gate blocks promotion on failure
3	Modifying shared Ontology without auditing consumers	Rename/remove/retype property without checking Workshop, FOO, OSDK, transforms	Consumer audit before any breaking change; branch/promote workflow for validation
4	Bypassing branch/promote under time pressure	Urgent fix made directly in production; secondary error not caught without CI	Branch/promote exists for urgent situations; exception requires data steward approval
5	Notebook logic that breaks when productionized	Notebook has implicit state (vars, imports, cached data) missing in scheduled execution	Run notebook from scratch with fresh kernel; verify idempotency; then convert to transform
6	Service account credential sprawl	Tokens in config files, version control, email, or plaintext deployment scripts	All credentials in C2DAO-approved store; zero in code or informal channels; violations are reportable incidents
7	Unscoped TypeScript Functions degrading performance	FOO aggregating large linked object sets works on test data, times out in production	Profile against production-scale data before deploying; expensive computations go in scheduled transforms
8	Implicit schema assumptions in transforms	Transform assumes specific columns/order; upstream schema change causes silent incorrect output	Validate input schema at top of every transform; fail fast and loudly on schema violations

SUMMARY — PRINCIPLES FOR THE MSS SOFTWARE ENGINEER

1. **Your code runs in production.** Other people depend on it. Act accordingly.
2. **Transforms are pure functions called by the platform.** No external state. No side effects.

3. **Idempotency is not optional.** Two runs on the same data must produce the same output.
4. **The Ontology is a shared API.** Breaking changes break everyone. Audit before changing. Use the branch workflow.
5. **TypeScript Functions run at query time.** Expensive computations belong in transforms, not FOO.
6. **OSDK is an Ontology client.** Understand the Ontology first. Access controls live at the Ontology layer.
7. **Test before you promote.** Unit tests in CI. Integration tests in dev. Smoke tests before production.
8. **Write for the next person.** Descriptive names, inline comments, README, no clever tricks.
9. **Common failures are predictable.** Recognize the failure modes in Section 9 before replicating them.

GOVERNING REFERENCES (ADDITIONAL)

Document	Relevance
Army DIR 2024-03	Digital Engineering Policy — Army-wide digital engineering adoption directive
FM 3-12	Cyberspace Operations and Electromagnetic Warfare — cyberspace operations doctrine
DA PAM 25-2-5	Software Assurance — software security and assurance requirements
DA PAM 25-1-1	Army IT Implementation Instructions — data management and IT governance procedures

CURRICULUM NOTES

Prerequisite: SL 3 (Advanced Builder) is REQUIRED — not recommended. Python proficiency (intermediate or higher) and TypeScript proficiency (intermediate or higher) are required independently of the TM series.

Advanced track: SL 4L graduates should pursue **SL 5L (Advanced Software Engineer)** for advanced topics including large-scale OSDK application architecture, Foundry platform extension patterns, CI/CD pipeline hardening, coalition data integration (NAFv4 compliance), and security compliance for operational software systems supporting classified environments.

Peer specialist cross-references: - **SL 4K (Knowledge Manager):** High-frequency coordination point. The KM designs knowledge ontology and pipeline architecture; the SWE implements pipelines requiring custom code. Breaking ontology changes affect knowledge pipeline code — coordinate before any schema change. - **SL 4H (AI Engineer):** AI workflows delivered through OSDK applications require SWE implementation of the application layer. Coordinate on CBAC compliance, credential management, and

application architecture before build begins. - **SL 4M (ML Engineer):** Coordinate on production pipeline implementation when model deployment requires custom inference infrastructure beyond standard Foundry Transforms, and on OSDK application surfaces for model-backed properties. - **SL 4G (ORSA):** ORSA analytical products sometimes require OSDK delivery interfaces for commander-facing applications. Coordinate with ORSA on output format and human-review-gate design before committing to application architecture.

WFF awareness: Software engineers on MSS build the production code layer that WFF-qualified users (SL 4A through SL 4F — Intelligence, Fires, Movement and Maneuver, Sustainment, Protection, and Mission Command) interact with daily. A production code failure is not an engineering metric — it is a WFF operational impact. Every application in production has a WFF customer. Know who they are, what their decision cycle is, and what degraded service means for their mission.

NOTE — New Doctrine Content in SL 4L: SL 4L now includes Army Data Plan SO 7 DevSecOps alignment with DDOF phases (section 1-5b), the UDRA 6-service architecture mapped to SWE responsibilities (section 1-7), and UDRA required metadata fields that pipelines must populate automatically.

Continue to SL 4L for task-based instruction in OSDK development, TypeScript Functions, Python transforms, CI/CD workflows, and security compliance.

END OF CONCEPTS GUIDE — SL 4L COMPANION

DRAFT