CONCEPTS GUIDE

SL 5L

USAREUR-AF

CONCEPTS GUIDE — SL 5L: ADVANCED SOFTWARE 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 5L: ADVANCED SOFTWARE ENGINEER — MAVEN SMART SYSTEM (MSS)

Forward: SL 5L is not about writing more complex code than SL 4L. It is about taking ownership of the platform's code health, architectural patterns, and engineering standards — and shaping how the entire team writes, reviews, tests, and deploys software. **Purpose:** Extends mental models from TM-40L Concepts Guide to advanced software engineering on MSS. Prerequisite: TM-40L Concepts Guide and SL 4L qualification. *HQ USAREUR-AF · v1.0 · 2026 · DISTRIB: USG only*

SECTION 1 — FROM ENGINEER TO PLATFORM ENGINEER

1-1. The Transition

BLUF: SL 5L is not about writing more complex code than SL 4L. It is about taking ownership of the platform's code health, architectural patterns, and engineering standards — and shaping how the entire team writes, reviews, tests, and deploys software.

At SL 4L, the measure of success is whether the code works. At SL 5L, the measure is different: does the code work in a way that makes the platform stronger over time?

Dimension	SL 4L Focus	SL 5L Focus
Delivery	Build features that work	Build features that work sustainably
Standards	Follow team standards	Define and enforce team standards
People	Receive code review	Lead code review; develop others

1-2. Engineering Leadership Without the Title

Most SL 5L engineers will not hold a positional leadership role in the traditional Army sense. They exercise engineering leadership through every interaction with the codebase and the team: the patterns they establish, the code reviews they conduct, the technical decisions they document and defend.

This is influence without authority. The SL 5L engineer cannot order a teammate to write better code. They must demonstrate through example, explain through review, and create systems (checklists, templates, CI gates) that make the right approach the path of least resistance.

Engineering leadership at this level shapes the platform's trajectory. A SL 5L engineer who establishes a clean error-handling pattern creates a standard that dozens of future transforms will follow. One who lets bad patterns go in code review creates an equally durable standard — in the wrong direction.

1-3. The Floor-Raising Imperative

BLUF: The SL 5L engineer's job is to raise the floor on quality across the team, not just to build excellent systems personally.

A team with one excellent engineer and eight mediocre engineers produces mediocre outcomes at scale. The force multiplier is not individual output — it is the ability to make every engineer on the team more effective.

Mechanism	Application
Code review	Not just approving or rejecting — teaching. Every review comment is an opportunity to explain why a better approach exists.
Pattern establishment	When a common problem is solved well, extract the solution into a shared pattern, library, or template.
Documentation	Architecture decisions, gotchas, and production incident lessons become institutional knowledge instead of tribal knowledge held by one person.
CI/CD gates	Automate quality enforcement wherever possible. A linter catching a naming violation is better than a human reviewer who might miss it.

1-4. Vignette — The New Transform Pattern

A SL 5L SWE on the USAREUR-AF MSS platform team observes that four different engineers have independently implemented error handling in Foundry transforms in four different ways. None are wrong, but two lack adequate production logging, one suppresses exceptions silently, and the fourth is verbose but difficult to read.

The SL 4L response: fix your own transforms and move on. The SL 5L response: document all four approaches with tradeoffs, define a team standard, implement it as a shared utility or code template, and update the code review checklist. The one-time investment prevents the problem from recurring across the next fifty transforms the team writes.

SECTION 2 — PLATFORM ARCHITECTURE MENTAL MODEL — ADVANCED

2-1. The Spark Execution Model

BLUF: SL 5L engineers understand not just how to write transforms but how the Foundry platform executes them: Spark execution, dataset partitioning, compute resource allocation, and performance reasoning at scale. The diagnostic skill — moving from symptoms to root cause — is the advanced competency.

Spark processes data in two phases:

Transformation phase (lazy): When code calls `.filter()`, `.join()`, or `.select()`, Spark does not execute immediately — it builds a directed acyclic graph (DAG) of operations. This plan is optimized before any data is read. Implication: push filters as early as possible in the chain; avoid wide operations (shuffles, joins) until the dataset has been reduced.

Action phase (eager): When code calls `.write()` or `.count()`, Spark materializes the plan and executes it across worker nodes. Shuffle operations — data movement across the network during joins — are the primary performance bottleneck. Every wide operation is a potential problem. Design transforms to minimize shuffles: reduce data early, filter before joining, partition by the key downstream operations will use.

2-2. Dataset Partitioning

Foundry datasets are stored as partitioned files (typically Parquet). Partitioning determines how data is physically organized on disk and how Spark reads it.

Scenario	Design Choice	Reasoning
High-frequency queries by unit + date	Partition by unit_id, date	Prunes irrelevant partitions on every query
Large, rarely filtered dataset used for joins	Partition by join key	Reduces shuffle size in downstream joins
Small reference dataset (<1M rows)	Single partition acceptable	Partitioning overhead not justified
Dataset with extreme cardinality skew	Bucket by hashed key	Distributes skewed data evenly

Bad partitioning — uneven partition sizes (skew) or partitioning along a dimension unrelated to common query patterns — is the most common cause of production performance failures. A dataset with millions of records in one partition and dozens in every other creates a bottleneck: one worker processes most of

the work while the rest sit idle.

2-3. Compute Resource Allocation

SL 5L engineers understand when and how to configure resource allocation — and, more importantly, when resource configuration is the wrong solution. The common mistake: a transform fails with an out-of-memory (OOM) error and the engineer requests more memory. More often, OOM errors indicate a design problem that more memory will not permanently fix.

Diagnostic reasoning for OOM errors: 1. What is the dataset size? Is it growing? If the dataset doubled, would the transform fail again? 2. Where in the execution plan does memory spike? Broadcasting a large reference dataset, collecting a large result to the driver, or an unskewed join are common culprits. 3. Can the operation be restructured to avoid materializing large intermediate datasets? Streaming aggregations, incremental processing, and pushdown filters often eliminate OOM errors without touching resource configuration.

Diagnostic reasoning for timeout errors: 1. Is the timeout from too much total work, or from a single slow step (Amdahl's Law — a slow serial step limits parallelism)? 2. Is the slow step a shuffle? What is the shuffle size? Can it be reduced? 3. Is the transform reprocessing data it has already processed? Incremental processing with watermarks eliminates reprocessing of historical data.

2-4. Symptoms-to-Root-Cause Reasoning

Apply this framework before changing resource configuration or requesting platform support:

Symptom	First Hypothesis	Diagnostic Step
OOM on worker node	Partition skew or large broadcast join	Check partition size distribution; check for `broadcast()` on large datasets
OOM on driver node	`collect()` or `toPandas()` on large dataset	Review code for driver-side data collection
Timeout on shuffle-heavy join	Excessive shuffle size	Profile shuffle read/write bytes; evaluate join order and filter placement
Transform slow at 10M rows, fails at 100M rows	Non-linear scaling due to cartesian product or unintended full scan	Review join conditions; check for missing join key filters
Correct output at small scale, wrong output at large scale	Non-deterministic operation or race condition in distributed execution	Review aggregation logic for non-associative/non-commutative operations
Build succeeds but downstream dataset is empty	Predicate pushdown filtered all data	Verify filter conditions against actual data distributions

2-5. Vignette — Diagnosing a Theater-Scale Transform Failure

The USAREUR-AF logistics integration team submits a new transform joining unit equipment readiness records against a theater-wide parts-availability dataset. At development scale (10K records), it runs in under two minutes. In production (40M equipment records × 800K parts records), it fails with an OOM error on the shuffle step.

The SL 5L SWE reviewing the PR does not recommend increasing memory. They trace the execution plan: the transform performs a join without a preceding filter on the equipment dataset, resulting in a full 40M × 800K shuffle. The fix: filter equipment records to the relevant AOR and date range before the join, reducing shuffle input by ~95%. The revised transform passes at production scale without any resource configuration change. Root cause was design, not resources.

SECTION 3 — ONTOLOGY DESIGN PATTERNS AT SCALE

3-1. Why Ontology Design Is an Engineering Concern

BLUF: At SL 5L level, the SWE contributes to Ontology design decisions that affect the entire platform. The critical competency is evaluating a design proposal for scalability before it is implemented — not after it fails under production load.

The Ontology is not a configuration exercise — it is a software architecture decision. Object Types, Properties, Links, and Actions define the data model that Workshop applications, OSDK integrations, and analytical pipelines consume. A design decision made today becomes a constraint on everything built on top of it. Retroactive changes are expensive: applications break, integrations require updates, and data must be migrated.

3-2. Object Type Inheritance Patterns

Foundry's Ontology does not support classical object inheritance. Two patterns model inheritance-like relationships:

Pattern	How It Works	Tradeoffs
Single Object Type with a type discriminator property	One Object Type (e.g., `Personnel`) with a `role` property distinguishing subtypes. All properties exist on every object; irrelevant properties are null.	Simpler structure, fewer Link Types, easier cross-subtype querying. Cost: property sparsity, schema confusion for consumers.
Separate Object Types per subtype	`ORSAPersonnel` and `SoftwareEngineer` Object Types	Clean separation of concerns, no sparsity, clear API contract per subtype. Cost:

Pattern	How It Works	Tradeoffs
with a shared parent link	each link to a `Personnel` parent. Type-specific properties on the subtype; shared properties on the parent.	complex Ontology graph, multi-hop traversal for cross-parent queries, higher maintenance burden.

Design guidance: Use the discriminator pattern when subtypes share most properties and differ only in a few fields. Use the separate-type pattern when subtypes have fundamentally different property sets and are accessed primarily within their own type context. Never use the separate-type pattern for fine-grained distinctions producing more than 10 Object Types with overlapping semantics — this creates an unmaintainable Ontology.

3-3. Sparse vs. Dense Property Design

Dense design: All properties are expected to have values for all objects. A `Unit` Object Type with `name`, `UIC`, `parent_UIC`, `echelon`, `AOR`, `commanding_officer` is dense — these apply to every unit.

Sparse design: Most properties are null for most objects. A `PersonnelRecord` Object Type with 80 properties where any given person has 15–20 populated is sparse — a warning sign.

Sparse Ontology designs typically indicate the designer conflated multiple distinct Object Types, or added optional metadata as top-level properties instead of modeling them as linked objects or secondary datasets. Operational cost: Workshop widgets display null values for most properties, OSDK queries return mostly-empty property sets, and computed properties over sparse properties produce misleading aggregates.

3-4. Computed Property Performance Tradeoffs

Computed Property Type	Performance Profile	When to Use
Simple arithmetic on the same object's properties	Low cost, evaluates locally	Always acceptable
Aggregation over directly linked objects (1-hop)	Moderate cost, scales with link count	Acceptable for infrequent access; beware objects with thousands of links
Aggregation over 2+ hop traversals	High cost, potentially exponential with link counts	Avoid; materialize as a dataset property instead
Computation requiring external data lookup	Variable; can become a platform-wide bottleneck	Avoid in high-frequency-access Object Types

Rule: If a computed property will be accessed by Workshop tiles serving many simultaneous users, model the computation carefully. A property cheap for one user becomes expensive when ten Workshop instances request it simultaneously for large Object Sets.

3-5. Large-Scale Link Traversal Performance

Link traversal performance determines whether an application feels responsive or broken. The key variable is **fan-out**: a link from `Brigade` to `Unit` has low fan-out (bounded number of units per brigade). A link from `Personnel` to `TrainingRecord` may have high fan-out if each person has hundreds of records.

Design interventions for high-fan-out scenarios: 1. Specify traversal depth explicitly in OSDK queries; avoid open-ended traversals. 2. Materialize aggregate values (e.g., "number of training records per person") as dataset properties rather than traversing links at query time. 3. Paginate large linked sets; never load all linked objects for a large Object Set in a single query. 4. Evaluate link direction — traversal cost may differ by direction; profile both for high-fan-out links.

3-6. Pre-Implementation Review Framework

Before approving an Ontology design proposal, evaluate against:

Question	Threshold / Guidance
Scale projection	How many objects at deployment? In 12 months? In 36 months? Does the design hold at projected scale?
Access pattern fit	Do the property and link structure support the primary query patterns efficiently?
Property sparsity	What percentage of properties are expected to be null for the average object? If >30%, challenge the design.
Computed property cost	For every computed property, estimate cost at peak load. Could it be materialized instead?
Link fan-out	What is the expected maximum cardinality on each side of every Link Type? Flag any links where one side can have >1,000 related objects.
Change impact	If this design needs to change in six months, what will break and how much will it cost?

SECTION 4 — OSDK APPLICATION ARCHITECTURE

4-1. The Architecture Decision That Precedes Everything

BLUF: At SL 5L level, OSDK applications go beyond displaying Object data. Advanced patterns include caching strategies, real-time updates, multi-Object-Type design, and authentication/authorization for applications serving multiple roles with different access levels.

Before designing any advanced pattern, answer: what is the access pattern? How many simultaneous users? Query frequency? Data volume per query? Acceptable staleness? These answers determine every architectural decision that follows. Building the near-real-time pattern for a daily-reporting use case wastes resources. Building the historical-analysis pattern for a near-real-time exercise application produces failure when it matters most.

4-2. Caching Strategies for Frequently Accessed Object Sets

The Ontology is not a high-frequency transactional database. Applications that query the same Object Sets repeatedly without caching drive unnecessary platform load and produce inconsistent user experience.

Client-side caching: The OSDK client maintains an in-memory cache of loaded Objects; subsequent accesses return cached values without network round-trips. Understand the cache invalidation model: for near-real-time applications, a few minutes' staleness may be unacceptable; for historical analysis, hours of staleness is irrelevant.

Application-layer caching: For Object Sets queried frequently by many users, cache the result at the application layer and serve subsequent users from cache. Cache invalidation trigger: when underlying data changes, invalidate and reload.

WARNING

Caching that never invalidates is indistinguishable from stale data. An application that shows yesterday's force positions because its cache never refreshed is actively harmful operationally. Define cache TTL explicitly and document it in the application's runbook.

4-3. Real-Time Updates via Ontology Subscriptions

OSDK supports subscriptions — the application registers interest in an Object Set and receives updates when Objects in that set change, enabling near-real-time applications without polling.

Architecture considerations: - Subscriptions generate network traffic proportional to the rate of underlying data change. For slowly-changing data (daily equipment status updates), subscriptions are low overhead. For rapidly-changing data (event streams every 30 seconds), subscriptions may generate more traffic than polling. - Applications maintaining subscriptions across many Object Types simultaneously may hit platform subscription limits. Subscribe only to data actively displayed; unsubscribe when the user navigates away. - Subscription failures must be handled gracefully. An application that silently stops receiving updates is worse than one that displays an explicit "data may be stale" indicator.

4-4. Multi-Object-Type Application Design

Pattern	Application
Load-what-you-need	Do not load all data from all Object Types at startup. Load the minimum set needed to render the initial view; load additional data as users navigate or filter.
On-demand relationship navigation	When a user selects an object and the application needs related objects from a linked Type, load those on demand — not preemptively. Most users will not navigate to most relationships.
Consistent identity across Object Types	In applications displaying the same real-world entity through multiple Object Types, maintain consistent identity mapping through application state management — not separate queries.

4-5. Authentication and Authorization for Multi-Role Applications

Design principle: surfaces, not data. Show users only the data surfaces they are authorized to access. A planner and a commander using the same application may see different Object Types, different property sets, or different available Actions. These are authorization requirements, not cosmetic differences.

Implementation pattern: Query the current user's role from the authentication context at application load. Use the role to determine which sections to render, which Object Types to query, and which Actions to expose. Do not rely solely on the Ontology's enforcement layer to hide unauthorized data — the application should not request data it has no authorization to display. Defense in depth: the Ontology enforces; the application cooperates.

NOTE

An application that queries unauthorized data and renders nothing when the Ontology returns empty results looks broken to the user. Design explicit access-level awareness: if a user is not authorized to see a section, render a clear indicator ("Access to this section requires [role]") rather than an empty panel.

SECTION 5 — SECURITY ENGINEERING ON MSS

5-1. The Security Design Mindset

NOTE

SL 4L now addresses computational governance as code (Army Data Plan SO 7 DevSecOps), DDIL pipeline design for contested network environments, and zero trust architecture principles for MSS platform engineering. The security design patterns below assume familiarity with that foundation; review SL 4L Sections 1-5b and 1-7 before applying advanced security engineering at enterprise scale.

BLUF: Security is a design constraint applied from the beginning of every system, not a feature added at the end. At SL 5L level, the engineer evaluates every component for its security model before writing the first line of code.

SL 4L engineers learn to implement security controls: apply CBAC markings, use HTTPS, rotate credentials. SL 5L engineers design security in from the architecture phase. Before choosing a data storage pattern, integration approach, or application architecture, ask: what is the security model of this design, and does it satisfy the requirement?

Questions that precede every design decision: 1. **Authentication:** Who is allowed to access this component? How is their identity verified? What happens when authentication fails? 2. **Authorization:** Of authenticated users, who is allowed to perform which operations? How are permissions granted, revoked, and audited? 3. **Data markings:** What is the sensitivity classification of the data this component handles? Are markings enforced at ingestion, storage, and access? Can markings be accidentally stripped? 4. **Audit logging:** What operations need to be audited? Are audit logs tamper-resistant? Who can access them? 5. **Least privilege:** Does each component have only the permissions required for its function?

5-2. Column-Level Permissions in Datasets

Foundry supports column-level access controls — specific columns restricted to users with specific roles while the remainder of the dataset is broadly accessible.

Security engineering considerations: - Document which columns are restricted and under what conditions. A consumer receiving null values for restricted columns must know why. - Column restrictions do not prevent a consumer from knowing that restricted columns exist. If the existence of a column is itself sensitive, evaluate whether a separate dataset is the correct design. - Joins between datasets with column-level restrictions can leak information indirectly. If restricted column A is the join key producing dataset B, dataset B implicitly encodes information from column A. Evaluate derived datasets for indirect information exposure.

5-3. Object Type-Level Access Controls in the Ontology

Principle	Application
Grant access to roles, not individuals	Individual grants create a maintenance burden and break when personnel rotate.
Apply minimum necessary access	An analyst needing to read equipment status Objects does not need write access.
Restrict Action execution separately from Object read access	A user can view an Object's properties while being unauthorized to execute Actions that modify it. Independent permissions.
Review access control schemas during rotation	Access appropriate for one incumbent may not be appropriate for all successors.

5-4. Action Authorization in Workshop Applications

Check	Question	Failure Mode
Actor authorization	Is the user executing this Action verified as authorized at the Ontology layer, not just the UI layer?	UI hides the button but the Action remains callable via API
Input validation	Are all Action inputs validated before execution?	Malformed input causes data corruption or exposes system behavior
Rollback capability	Can the effect of this Action be reversed?	Accidental or malicious Action execution with no recovery path
Audit trail	Is Action execution logged with actor identity, timestamp, and parameters?	No forensic record of who changed what and when
Scope limitation	Does the Action modify the minimum set of Objects required?	Over-scoped Action modifies objects outside the intended set

5-5. Security Review of a Code PR at SL 5L

Security review is integrated into code review on every PR — not a separate audit. Mandatory checks:

Check	Standard
Credential handling	Credentials stored in environment variables or the Foundry credential store. Any hardcoded credential — including test credentials for low-privilege accounts — blocks the PR.
Data flow tracing	Follow data from ingestion to output. At each step: does data carry its markings? Can markings be stripped or downgraded? Can restricted data be written to an unrestricted location?

Check	Standard
External connectivity	Is any external connection being opened? Is the external system on the approved integration list? Is the connection authenticated and encrypted? Is data being sent reviewed for sensitivity?
Logging review	Do log statements inadvertently capture PII, credentials, or sensitive data payloads? Log sanitized identifiers (record IDs, hashed values); do not log field values from sensitive datasets.
Error handling review	Do error conditions expose system internals (stack traces, schema information, connection strings) in messages returned to users? Describe the failure category; do not expose implementation details.

SECTION 6 — CI/CD AND AUTOMATED PROMOTION — ENGINEERING THE GOVERNANCE WORKFLOW

6-1. What CI/CD Enforces on MSS

BLUF: The SL 5L engineer designs and maintains the automated pipeline moving code from development to production. An effective CI/CD pipeline enforces governance requirements without creating bottlenecks that slow the team.

The MSS platform team's CI/CD pipeline enforces three categories:

Category	Examples	PR Behavior
Code quality	Automated tests pass, coverage thresholds met, static analysis clean, naming conventions compliant	Runs on every PR; blocks merge on failure
Security requirements	No hardcoded credentials (secret scanning), no unapproved external endpoints, dependency vulnerability scan passes	Non-negotiable blockers
Documentation requirements	ADRs exist for significant design changes, runbook updates included for operational procedure changes, CHANGELOG entries for user-facing changes	Runs on every PR; missing documentation generates warning or blocks depending on change classification

6-2. Pipeline Design Principles

Principle	Application
Fast feedback	PR authors should know within minutes whether changes pass quality gates. A 45-minute CI pipeline discourages frequent commits, leads engineers to batch changes, increases blast radius of failures. Parallelize tests; run fast checks first (lint before full test suite).
Fail fast, fail clearly	Failure messages must identify the specific check, the specific file, and a reference to the relevant standard. Generic "build failed" is not actionable.
Idempotent checks	Checks depending on external service availability, time-of-day, or random seeds produce flaky results that erode trust. Engineers who learn to ignore flaky failures will also ignore real failures.

6-3. Gate Progression

Severity	Action	Examples
Hard block	Merge prevented until resolved	Test failure, credential scan failure, security finding
Soft block	Warning logged; requires explicit override with justification	Coverage below threshold, missing documentation
Advisory	Informational only	Style suggestions, optional improvements

6-4. The Governance Bottleneck Problem

Governance requirements are operationally necessary — and a common source of bottlenecks. When governance checks are manual, they create a queue. When the queue backs up, engineers find workarounds.

SL 5L approach: automate everything that can be automated, and make automated checks informative rather than opaque. A naming convention check that tells the engineer "Dataset name 'unit_data_v2_final' violates USAREUR-AF naming standard — expected format: [domain]*[entity]*[version] (e.g., log_unit_readiness_v1)" enables immediate self-service correction. The automated check becomes a teaching tool.

Reserve manual governance review for decisions requiring human judgment: significant architecture changes, new external integrations, access control schema changes.

6-5. Promotion Workflow Design

Recommended gate sequence: 1. All CI checks pass (automated) 2. PR approved by one SL 5L-qualified reviewer (human) 3. Staging environment deployment and smoke test (automated) 4. C2DAO sign-off for changes affecting production Ontology or CBAC policies (human) 5. Production deployment with automated rollback on health check failure (automated)

Key principle: automate verification steps; reserve human review for approval decisions. Humans are slow and expensive at verification. They are not expensive at approval if verification has already been done automatically.

SECTION 7 — CODE REVIEW AS A TEAM QUALITY MECHANISM

7-1. The Four Dimensions of Code Review

BLUF: SL 5L engineers lead code reviews. Effective code review evaluates correctness, design, maintainability, security, and operational risk — and delivers feedback in a way that improves the reviewer without demoralizing them.

Dimension	What It Asks
Correctness	Does the code do what it is supposed to do? Does it handle edge cases — empty input, join with no matches, API timeout?
Design	Is this the right approach? Is there a simpler way to solve this? Will this design hold as requirements evolve? Are component responsibility boundaries clearly defined?
Maintainability	Will someone else understand this code in six months? Are variable and function names clear? Is complex logic commented? If debugged at 0300, can a qualified engineer understand it quickly?
Security and operational risk	Does this change introduce a vulnerability? What is the blast radius if it fails in production? Is the change reversible? Are monitoring and alerting hooks in place?

7-2. Blast Radius Analysis

Make blast radius assessment explicit in every code review:

Blast Radius	Description	Example
Local	Affects one user or one non-critical pipeline	A personal dashboard transform fails

Blast Radius	Description	Example
Unit	Affects one unit's data or workflows	A brigade readiness rollup transform produces wrong outputs
Platform	Affects a shared service or the Ontology schema	An OSDK change breaks a widely-used Object Type's API contract
Theater	Affects all USAREUR-AF users or mission-critical pipelines	A CBAC policy change removes access for all G3 users

Changes with theater-level blast radius require additional review steps, coordination with C2DAO, and a documented rollback plan before deployment. Code review identifies the blast radius and ensures the review process matches the risk.

7-3. Giving Effective Code Review Feedback

Principle	Application
Separate the code from the person	"This function is doing three unrelated things" — about the code. "You wrote a confusing function" — about the person. Review the code.
Explain the why, not just the what	"Change this to X because it handles the empty-input case that Y silently drops" teaches the reviewer something they carry forward. "Change this to X" leaves them guessing.
Distinguish blocking feedback from suggestions	Label explicitly: "Blocking: this creates a security vulnerability" versus "Suggestion: this could be simplified." Reviewers should not guess which feedback requires a change before merge.
Acknowledge good work	When a reviewer implemented a complex pattern correctly or handled an edge case you would not have thought of, say so. Establishes the standard for what "good" looks like.
Calibrate to the reviewer's level	A SL 3 engineer submitting their first complex transform needs different feedback than a SL 4L engineer who made an architectural misjudgment.

7-4. The Reviewer Development Obligation

SL 5L engineers have an obligation to develop the engineers whose code they review. Every code review is a mentorship opportunity. The cumulative effect of many well-executed code reviews is a team whose quality improves over time.

In practice: when a reviewer makes a design mistake you have seen before, discuss why the pattern matters after the review — do not just correct it in the PR. When a reviewer's code reflects a misunderstanding of how the platform works, point them to the relevant SL 4L or SL 5L section. When a

reviewer does something well you want to reinforce, be explicit about it.

SECTION 8 — ENGINEERING DOCUMENTATION AS OPERATIONAL INFRASTRUCTURE

8-1. What Must Be Documented

BLUF: At SL 5L level, documentation is an engineering artifact with the same lifecycle as code. Missing documentation is a defect. The operational cost becomes apparent when a system fails at 0300 and no one knows how to recover it.

Document Type	Content	Co-location
Architecture Decision Records (ADRs)	Context, options considered, decision made, rationale, consequences. Prevents future engineers from accidentally undoing decisions without understanding why they exist.	Repository directory the ADR pertains to
API contracts	Input parameters, output schema, authentication requirements, error responses, rate/throughput constraints. Must be versioned — when the API changes, documentation changes. Consumers must be notified of breaking changes.	API codebase; generated from code annotations where possible
Data schemas	Field names, types, descriptions, acceptable value ranges, known data quality issues.	Co-located with the transform that produces the dataset
Deployment runbooks	Restart procedures, scaling procedures, credential rotation, dependency update procedures. Written for a competent but unfamiliar engineer — not the engineer who built it. Test by having a second engineer follow without assistance.	Adjacent to the system component
Incident postmortems	What happened, timeline, root cause, changes being made to prevent recurrence. Not blame documents — engineering artifacts.	Team knowledge repository

ADR format (adapted for MSS):

```
ADR-[number]: [Short title]
Date: [YYYY-MM-DD]
Status: [Proposed / Accepted / Superseded]
Context: [What problem does this decision address?]
Decision: [What did we decide to do?]
Rationale: [Why did we choose this option over alternatives?]
Consequences: [What are the tradeoffs of this decision?]
```

8-2. Making Documentation Sustainable

Documentation that lives in a separate wiki detached from the code it describes will rot. Developers update code; the wiki is forgotten. Within six months, the wiki is misleading — worse than no documentation because readers may trust it.

Practice	Application
Co-location principle	Documentation lives as close to the code it describes as possible. ADRs in the repository directory they pertain to. Schema documentation in or adjacent to the producing transform.
Documentation as PR requirement	Documentation updates required as part of merge criteria for PRs that change system behavior. CI check: if a PR modifies a transform that produces a documented dataset, verify the schema documentation file was also updated.
Review cycle	Quarterly documentation review: pull up the runbook for each critical system and verify against current reality. Flag outdated sections; assign updates with the same priority as code maintenance.

8-3. The 0300 Test

When designing documentation, apply the 0300 test: if a production system fails at 0300 and a qualified but unfamiliar engineer needs to recover it, does the documentation exist to enable that recovery without waking up the system's original author?

If the answer is no, the documentation is incomplete — not a quality-of-life issue, but an operational readiness issue. On a platform supporting theater-level decision-making, a pipeline that cannot be recovered without calling one specific engineer who is on leave is a single point of failure.

Documentation that makes the 0300 test possible: - Runbook for each critical pipeline: what healthy looks like, how to diagnose common failure modes, how to restart or roll back - Dependency map: what does this system depend on, and what depends on it? - Contact escalation: if the runbook does not resolve the issue, who is the next call, and what information do they need beforehand?

SECTION 9 — ADVANCED FAILURE MODES — WHAT SL 5L ENGINEERS GET WRONG

BLUF: The failure modes at SL 5L level are not beginner mistakes — they are predictable errors made by capable engineers who have access to powerful tools and face genuine operational pressures.

Failure Mode	How It Manifests	Corrective Discipline
Over-engineering	Applying incremental processing, Ontology subscriptions, complex OSDK patterns, or multi-hop traversals when simpler approaches would serve the mission better. Causes: intellectual preference for elegant solutions, defensive engineering ("what if we need this later?"), status signaling. Systems become harder to understand, debug, and hand off.	Before adding complexity, ask what simpler approach was rejected and why. If the simpler approach has a concrete operational deficiency (too slow, too resource-intensive, too stale), complexity is justified. If it would have worked, refactor.
Underinvesting in observability	Observability instrumentation is not immediately functional — the investment pays off only when something goes wrong. Result: production failures are discovered by users reporting them, not by the engineering team detecting them.	Treat observability requirements as functional requirements: define what healthy looks like (metrics), how to know something is wrong (alerting thresholds), and how to diagnose what went wrong (logging structure). A system not meeting these requirements is not ready for production. Minimum: transform build alerting, data freshness monitoring, output quality checks, OSDK error rate monitoring, Action execution failure alerting.
Technical debt accumulation	Systems that "work" but are built on a fragile foundation accumulate debt silently. Failure point: a growth event — dataset doubles, new use case requires schema extension, or a new feature cannot be added without a rewrite. Even SL 5L engineers can build technically sophisticated systems with documentation gaps, hardcoded assumptions, or test coverage that exercises only happy paths.	Treat technical debt as an operational risk. Budget time for debt reduction each sprint. When building a new feature on top of an existing system, assess the system's debt first. Code review should explicitly identify debt created by a PR; the team decides whether to accept it or address it before merge.
Security as a compliance checkbox	Compliance mindset: goal is to pass the checklist, not to build a secure system. Engineers look for minimum evidence to satisfy each checklist item, not actual security assurance. A credential stored in an approved vault but never rotated, with overly broad permissions, shared across multiple systems, meets the "no hardcoded credentials" checklist requirement — and is still a security liability.	Security is a design constraint (Section 5), not a compliance requirement. The checklist verifies work that was already designed with security in mind — it is not a substitute for that design thinking.

Failure Mode	How It Manifests	Corrective Discipline
Failing to develop the next generation	Heavy operational workloads cause SL 5L engineers to deprioritize mentorship and code review quality because these activities feel less urgent than delivering the next system. Compounding failure: team capability stagnates, the SL 5L engineer becomes a quality bottleneck, and when they rotate out, institutional knowledge was never transferred.	At USAREUR-AF, operational data team personnel rotate. The team that built a system will not be the team that maintains it in two years. Treat engineer development as a mission requirement. Code review is a teaching opportunity — invest in it accordingly. The SL 5L engineer's measure of success is not the quality of systems they build alone — it is the quality of systems the team builds after they leave.

Vignette — The V Corps G3 OSDK Application Review

A SL 4L engineer submits a PR for a new OSDK application serving the V Corps G3 — a near-real-time force disposition display used during exercises. The application is functionally correct: it queries the right Object Types, displays the right data, and handles the primary use case correctly.

The SL 5L lead reviewing the PR identifies four issues not caught in functional testing: - The application loads all force disposition Objects at startup (4,200 at current exercise scale) rather than filtering to the G3's AOR. Long initial load times as exercise scale increases. - No cache invalidation logic — the application caches at load and never refreshes during a session. During a 12-hour exercise, displayed data will be up to 12 hours stale. - Actions available to the G3 are visible (though non-functional) to read-only roles due to UI logic that checks permissions server-side but renders the button before the check resolves. - No runbook for the application — no documentation of subscription TTL, how to force a refresh, or what to do if the Ontology subscription drops.

The SL 5L lead does not reject the PR. They document each finding with its operational reasoning, assign severity (blocking vs. advisory), and provide specific remediation guidance. The conversation that follows raises the SL 4L engineer's understanding of OSDK caching, authorization patterns, and production documentation requirements — knowledge they carry into the next five applications they build.

SUMMARY TABLE — SL 5L CONCEPTS AT A GLANCE

Section	Core Concept	Key Discipline
1. Platform Engineer	Engineering leadership without the title	Raise the floor, not just the ceiling

Section	Core Concept	Key Discipline
2. Platform Architecture	Spark execution model; symptoms-to-root-cause	Diagnose from evidence; fix the design, not the resources
3. Ontology at Scale	Design patterns for scalability	Evaluate before implementing; fan-out and sparsity as risk signals
4. OSDK Architecture	Caching, subscriptions, auth patterns	Access-pattern-first design; authorization defense in depth
5. Security Engineering	Security as design constraint	Trace every data flow; audit trail for every Action
6. CI/CD	Governance without bottlenecks	Automate verification; reserve humans for approval
7. Code Review	Four-dimensional review; developer mentorship	Blast radius awareness; feedback that teaches
8. Documentation	Documentation as operational infrastructure	The 0300 test; co-location; review cycles
9. Failure Modes	Over-engineering, observability debt, security as compliance	Design discipline; team development as mission requirement

PEER SL 5 CROSS-REFERENCES AND WFF INTEGRATION

Peer SL 5 Publications. Platform engineers build infrastructure consumed by all advanced specialist tracks. Coordinate with practitioners in these companion publications.

Publication	Track	Coordination Point
SL 5G	Advanced ORSA	Platform infrastructure supporting analytical pipelines
SL 5H	Advanced AI Engineer	OSDK integration with AI systems
SL 5M	Advanced ML Engineer	ML model-serving integrations; feature pipeline infrastructure
SL 5J	Advanced Program Manager	Platform engineering program coordination; SWE team structure
SL 5K	Advanced Knowledge Manager	Platform SDK patterns for enterprise KM system backends
SL 5N	Advanced UI/UX Designer	Frontend design collaboration; design system implementation

Publication	Track	Coordination Point
SL 5O	Advanced Platform Engineer	Platform/application boundary; DevOps collaboration; deployment targets

WFF Operational Consumer Note. The software platform built and governed by SL 5L engineers is the delivery mechanism for all capabilities consumed by the six Warfighting Function (WFF) tracks: Intelligence (SL 4A), Fires (SL 4B), Movement and Maneuver (SL 4C), Sustainment (SL 4D), Protection (SL 4E), and Mission Command (SL 4F). Platform reliability, security, and performance are not engineering abstractions — they determine whether a G2 analyst can pull an intelligence synthesis at 0300, whether a G4 planner can access logistics forecasts before a decision brief, and whether the G3's force disposition display is current when the commander needs it. The failure modes addressed in this guide — over-engineering, observability debt, security as compliance — have direct operational consequences for WFF practitioners.

This guide is a conceptual companion to SL 5L. It does not replace task-based training. Qualification requires demonstrated performance of SL 5L tasks under the conditions and to the standards specified in SL 5L.

USAREUR-AF Operational Data Team — MSS Platform Engineering | Version 1.0 — 2026