DRAFT — UNOFFICIAL — NOT FOR OPERATIONAL USE

TECHNICAL MANUAL

SL 4L

USAREUR-AF

TM-40L — 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-40L — MAVEN SMART SYSTEM (MSS)

Forward: SL 4L qualifies software engineers to build external applications, write integration code, develop TypeScript FOO logic, and deploy production-grade solutions on the MSS platform. This is a developer manual — it contains code, not just concepts. **Prereqs:** SL 1, Maven User; SL 2, Builder; SL 3, Advanced Builder (required); Python proficiency (intermediate or higher); TypeScript proficiency (intermediate or higher) *HQ USAREUR-AF · v1.0 · 2026 · DISTRIB: USG only · AUTH: C2DAO/UDRA v1.1*

WARNING: Code errors at SL 4L level can corrupt shared Ontology state, break downstream pipelines, and produce incorrect data products consumed by commanders making operational decisions. Apply engineering discipline. Test thoroughly. Coordinate governance before deploying to production. CAUTION: OSDK service account tokens and Foundry Platform SDK credentials are operational secrets. Loss or exposure of these credentials constitutes a security incident. Report immediately to unit S6/G6 and C2DAO.

CHAPTER 1 — INTRODUCTION: THE SOFTWARE ENGINEER ROLE IN MSS

1-1. Software Engineer Specialist Manual

BLUF: SL 4L qualifies software engineers to build external applications, write integration code, develop TypeScript FOO logic, and deploy production-grade solutions on the MSS platform. This is a developer manual — it contains code, not just concepts.

This manual provides task-based instruction for software engineers operating on the Maven Smart System (MSS). MSS is the USAREUR-AF enterprise AI/data platform built on Palantir Foundry. SL 4L personnel design and implement the technical components that advanced builders (SL 3) specify but cannot build without code.

SL 4L covers OSDK (Ontology SDK): authenticating, querying objects, executing actions, subscribing to changes from external applications; OSDK Health Dialog for error visibility and debugging; temporary media uploads via OSDK and Functions; Pilot for AI-assisted OSDK application scaffolding; Model Context Protocol (MCP) for connecting AI agents to the Ontology; Foundry Platform SDK (Python): reading and writing datasets, managing transactions, accessing file resources; TypeScript Functions on Objects (FOO): computed properties, bulk query patterns, performance optimization; actions with

complex validation: TypeScript validators, multi-step action flows, conditional logic; Slate: legacy custom HTML/CSS/JavaScript application development hosted on Foundry (documented for maintenance of existing Slate apps only — do not use for new development; see Chapter 7); CI/CD for Foundry: repository structure, automated testing, branch promotion workflows; security: CBAC in external apps, credential management, marking compliance in OSDK queries, audit trails; and integration patterns: REST APIs, webhooks, cross-system data flows connecting MSS to external Army systems.

SL 4L does NOT cover Workshop application design (no-code/low-code) — see SL 2, SL 3; Pipeline Builder visual UI — see SL 2, SL 3; basic Ontology modeling (UI-based) — see SL 3; PySpark transforms — see SL 4H (AI Engineer) for AI pipelines; SL 3 for design; AIP Logic configuration — see SL 3; or Agent Studio development — see SL 4H.

NOTE

SL 4L is peer to SL 4H (AI Engineer), SL 4M (ML Engineer), and SL 4G (ORSA). All four tracks require SL 3 as prerequisite. Each track owns a distinct technical domain. Coordinate across tracks — operational systems frequently require all four disciplines.

1-1a. The ASWF Problem-Solution Development Methodology

XVIII Airborne Corps published their Operational Data Team problem-solution development methodology in *Military Review* ("Fighting with Live Data," February 2026). The methodology is directly informed by the Army Software Factory (ASWF) and industry best practices. It represents one corps' approach — the specific timelines and phase gates below reflect XVIII ABC's pilot experience and may be adapted for different echelons, team sizes, and mission sets. The underlying principle — structured problem definition before solution development — is transferable regardless of organizational context.

The methodology prioritizes understanding and solving problems — not implementing preconceived solutions. The process begins with a staff element articulating a **problem**, not a desired solution, and progresses through phases punctuated by decision points (invest, divest, pivot):

Phase	Duration	SWE Role
Scoping	2 weeks	Assess technical feasibility; identify data source availability, platform constraints, integration requirements
Discovery	4 weeks	Prototype data access patterns; validate that required data exists and is accessible at required fidelity
Framing	4 weeks	Define MVP technical architecture; write implementation plan with SWE deliverables and dependencies
Development	8 weeks	Build MVP through iterative sprints; coordinate with DE (data pipelines), DS (models), UX (interfaces)

Phase	Duration	SWE Role
Handoff	2 weeks	Production hardening, documentation, CI/CD pipeline finalization, transition to sustainment

Total time to MVP: at least 5 months. The XVIII ABC BDA visualization capability followed this process: prototype in 3 months, MVP in 6, and full handoff to an Army enterprise program of record (Army Intelligence Data Platform, INSCOM) in 9 months.

During exercises: SWEs on MVP-phase products execute bug fixes and minor adjustments based on user feedback only. No major changes. Products in discovery/framing are held stable — the SWE focuses on gathering data and validating technical assumptions for future iterations.

> **NOTE**
>
> The ASWF trains its students in this methodology. SL 4L graduates who enter ODT assignments are expected to operate within this framework from day one. The PM (SL 4J) owns the process cadence; the SWE owns the technical execution within each phase.

Source: Forney, Herrmann, and Steele, "Fighting with Live Data," Military Review Online Exclusive, February 2026.

Supplementary: Adkins, "Achieving Decision Dominance," Military Review, Jan-Feb 2025, introduces the term "automated fighting products" (AFPs) — data visualization tools connected to live data pipelines that reduce staff burden. MSS applications built by SL 4L graduates are AFPs by this definition.

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

Prerequisite: SL 3 (Advanced Builder) is REQUIRED. Python proficiency (intermediate or higher) and TypeScript proficiency (intermediate or higher) are required independently of the TM series.

Advanced track: Upon completing SL 4L, qualified Software Engineers 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), and security compliance for operational software systems.

Peer specialist tracks: The Software Engineer implements the production code layer that all other specialist tracks depend on. Coordinate with SL 4H (AI Engineer) when AI workflow outputs require OSDK application surfaces for staff-section delivery. Coordinate with SL 4M (ML Engineer) when model deployment requires custom inference infrastructure beyond standard Foundry Transforms. Coordinate with SL 4K (Knowledge Manager) on knowledge pipeline implementation — the KM defines the architecture; the SWE implements pipelines requiring custom logic. The KM-SWE interface is a high-frequency coordination point: knowledge ontology design changes have direct impact on downstream application code.

WFF awareness: Software engineers on MSS build the application layer that WFF-qualified users (SL 4A through SL 4F — Intelligence, Fires, Movement and Maneuver, Sustainment, Protection, and Mission Command) interact with daily. An OSDK application degrading for a Fires (SL 4B) targeting workflow or a Movement and Maneuver (SL 4C) tracking tool has direct operational impact. Code quality, test coverage, and rollback procedures are not abstract engineering standards — they are WFF readiness factors.

1-3. The Software 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. Software engineers at SL 4L level are the technical implementers of the USAREUR-AF data ecosystem.

The SL 4L role in the data chain:

```
MISSION REQUIREMENT
        |
        v
   -30 BUILDER              <- Design + configure
   (Ontology, Workshop,
    Action design specs)
        |
        v
   -40L SOFTWARE ENGINEER   <- You are here
   (OSDK, external apps,
    FOO, CI/CD,
    integration code)
        |
        v
   OPERATIONAL PRODUCT
   (Unit tracking apps,
    SITREP automation,
    readiness dashboards,
    EUCOM integrations)
        |
        v
   -10/-20 USER
   (Consume + act on data)
```

Typical SL 4L deliverables in the USAREUR-AF context:

Deliverable	Description	Example
External application	Web or desktop app consuming MSS Ontology via OSDK	V Corps readiness dashboard consuming unit status objects
SITREP automation	App that reads Ontology objects and pushes formatted reports to external systems	Automated SITREP generation to EUCOM J3 portal

Deliverable	Description	Example
Integration service	Microservice bridging external Army system to MSS via REST/webhook	GCSS-Army feed into MSS equipment ontology
Slate app	Custom HTML/JS application hosted within Foundry (LEGACY — do not use for new development)	Interactive operational map with custom layers
FOO library	TypeScript computed properties enriching Ontology objects	Readiness score computed from component statuses
CI/CD pipeline	Automated testing and promotion workflow for Foundry code resources	Branch-gated promotion for production transforms

1-4. Prerequisites and Baseline Skills

Complete all of the following before beginning this manual:

Prerequisite	Verification
SL 1 (Maven User)	Can navigate MSS, consume data products, submit issues
SL 2 (Builder)	Can build basic Workshop apps, configure Object Types, create Actions via UI
SL 3 (Advanced Builder)	Can design full ontology models, advanced Pipeline Builder flows, multi-step Actions
Python proficiency	Can write, test, and debug Python scripts; understands async patterns, exception handling, type hints
TypeScript proficiency	Can write typed TypeScript; understands async/await, generics, module systems
Git proficiency	Understands branching, merging, pull requests, rebase workflows
REST API familiarity	Can read and write against REST APIs; understands auth patterns (OAuth2, bearer tokens)

NOTE

This manual does not teach Python or TypeScript. If you need to develop either skill, complete coursework before proceeding. Reference `learn-data.armydev.com` for approved training resources.

1-5. Governing References

Document	Relevance
USAREUR-AF C2DAO Guidance	Theater-level architecture standards; OSDK API enrollment requirements
NATO Architecture Framework v4 (NAFv4)	Coalition data architecture — applies to any integration with MPE-accessible objects
AR 25-2	Army Cybersecurity — credential handling, system authorization
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
NATO ADatP-34 / NISP	C3 interoperability compliance — applies to any MSS integration with coalition systems
STANAG 5643 (proposed) — MIM Governance Standard	NATO MIP Information Model governance — data model versioning, change proposals, national extensions
ADatP-5644 — Web Service Messaging Profile (WSMP)	NATO standard for MIM-formatted data exchange over networks — message structure, transport protocol

1-5a. Strategic Guidance

The following are strategic guidance documents — not doctrine — that inform MSS training design and operational context.

Document	Authority	Relevance
Army CIO Data Stewardship Policy (April 2, 2024)	Army CIO	Data stewardship hierarchy, governance chain, API access policy
UDRA v1.1 (February 2025)	Army Enterprise	Unified Data Reference Architecture — domain ownership, integration standards

DRAFT

Document	Authority	Relevance
DoD Data Strategy (2020)	OSD	VAULTIS-A framework (supersedes VAUTI) — Visible, Accessible, Understandable, Linked, Trusted, Interoperable, Secure, Auditable. 8 dimensions per DDOF Playbook v2.2 (Dec 2025); 85% weighted avg = DDOF Phase 3 quality gate.
NATO Digital Transformation Implementation Strategy (Oct 2024)	NATO	NATO digital transformation roadmap — MDO interoperability context for SWE deliverables

Reference: `learn-data.armydev.com` — authoritative reference for OSDK API versions, enrollment procedures, and approved integration patterns. Consult before beginning any new external application development.

1-5b. Army Data Plan SO 7 and DevSecOps Tempo

BLUF: Army Data Plan Strategic Objective 7 directs the Army to field a "Cloud, Data, DevSecOps Enabled Workforce & Leaders That Support Digital Operations." The DDOF Playbook mandates MVP delivery within 30 days. SWEs build the pipelines and applications that sustain this tempo.

SO 7 establishes the expectation that every data professional — including software engineers — operates within a DevSecOps culture. For SL 4L personnel this means:

- **Speed to value.** DDOF's 30-day MVP window is the planning constraint. Design pipelines, OSDK applications, and integration services for incremental delivery, not waterfall release cycles.
- **Shift-left security.** Embed automated security scanning and compliance checks in CI/CD pipelines (Chapter 8) before code reaches production.
- **Continuous feedback.** Production monitoring and user feedback loops feed the next sprint — not a separate sustainment phase.

NOTE — DevSecOps to DDOF Phase Mapping:

DevSecOps Activity	DDOF Phase	SWE Action
Build sprint	Phase 4 — Development	Write and unit-test pipeline/application code
Automated testing + security scan	Phase 5 — Test & Evaluation	Run CI test suite, SAST/DAST scans, marking validation
CI/CD deployment + monitoring	Phase 6 — Operations	Promote to production branch, configure alerting, verify data flow

Phases are sequential gates. Code that fails Phase 5 security scan does not promote to Phase 6. No exceptions.

1-6. The USAREUR-AF 5-Layer Data Stack — SWE Responsibilities

```
+-------------------------------------------------------------+
|  LAYER 5: ACTIVATION                                        |
|  External apps, SITREP automation, EUCOM integrations       |
|  --> SL 4L primary operating layer                         |
+-------------------------------------------------------------+
|  LAYER 4: ANALYTICS                                         |
|  External OSDK apps, custom dashboards, FOO-enriched objects |
|  --> SL 4L builds custom analytical applications           |
+-------------------------------------------------------------+
|  LAYER 3: SEMANTIC (ONTOLOGY)                               |
|  FOO computed properties, Action validators, OSDK queries   |
|  --> SL 4L writes code executing against this layer        |
+-------------------------------------------------------------+
|  LAYER 2: INTEGRATION                                       |
|  Platform SDK dataset operations, transaction management    |
|  --> SL 4L manages programmatic dataset access             |
+-------------------------------------------------------------+
|  LAYER 1: INFRASTRUCTURE                                    |
|  CBAC, marking, credential provisioning                     |
|  --> SL 4L consumes; C2DAO/G6 administers                  |
+-------------------------------------------------------------+
```

SL 4L engineers are the primary implementers of Layers 4 and 5 technical components. They write code that executes at Layer 3 (Ontology) and Layer 2 (datasets) but coordinate with C2DAO for any Layer 1 changes.

1-7. UDRA Service Architecture — SWE Implementation Map

BLUF: UDRA v1.1 defines six core services for Army data ecosystems. Software engineers implement or integrate with every one of them. The table below maps each UDRA service to concrete SWE responsibilities and MSS implementation patterns.

UDRA Service	SWE Responsibility	MSS Implementation
Production	Build data pipelines and transforms	Foundry pipeline code (Python transforms, PySpark)
Orchestration	Register, discover, and notify services	Data catalog integration, pipeline scheduling

UDRA Service	SWE Responsibility	MSS Implementation
Consumption	APIs, dashboards, analytics surfaces	Workshop apps, OSDK external apps, API endpoints
Access Management	ABAC/RBAC, zero-trust enforcement	Marking configuration, CBAC policy, access control rules
API Brokerage	Endpoint management and routing	API gateway configuration, OSDK enrollment
Computational Governance	Automated policy enforcement	`@check` decorators, validation rules, data quality gates

Source: UDRA v1.1 (February 2025)

Every OSDK application, pipeline, or integration service a SL 4L engineer builds touches at least three of these services. Design reviews must identify which services a deliverable engages and confirm the implementation satisfies UDRA requirements for each.

NOTE — UDRA Required Metadata (15 fields): Every data product registered in the Army data ecosystem must carry these metadata fields per UDRA v1.1: `apiEndpoint`, `authorizationReference`, `creationDateTime`, `custodian`, `description`, `disclosureAndReleasability`, `format`, `handlingRestrictions`, `identifier` (UUID), `name`, `originator`, `securityClassification`, `version`. SWEs must ensure pipelines populate these fields automatically — missing metadata blocks promotion past the Computational Governance gate. Validate metadata completeness as a CI/CD check (see Chapter 8).

WARNING: The `securityClassification`, `disclosureAndReleasability`, and `handlingRestrictions` fields are security-critical. Incorrect values can result in unauthorized data exposure. Populate these fields from authoritative source markings — never derive them from user input or default values.

CHAPTER 2 — OSDK FUNDAMENTALS

CODE EXAMPLES: Runnable OSDK and Ontology access patterns referenced in this chapter are available in the local development shim at `data_skills/13_foundry_patterns/ontology_modeling.py`. Transform and check patterns are in `python_transforms.py` and `foundry_checks.py`. Activate the venv: `source skills/data_skills/.venv/bin/activate`.

2-1. What Is the OSDK

BLUF: The Ontology SDK (OSDK) is the primary programmatic interface for external applications to query Ontology objects and execute Actions on MSS. It is the correct integration method for any external application consuming MSS data.

NOTE — Palantir Developers reference: *Product Launch: Rapid Software Development with OSDK 2.0* — Covers the OSDK 2.0 release and its developer experience improvements, including simplified client setup and improved TypeScript type generation. Relevant when configuring a new OSDK project or upgrading from an earlier version. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Building with Palantir AIP: the Ontology Software Development Kit* — Demonstrates OSDK in action for AIP-integrated applications, showing how external apps authenticate and query Ontology objects. Provides a practical walkthrough of the patterns covered in this chapter. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Foundry Reference Project | Structure* — Walks through the canonical Foundry Reference Project structure, covering how OSDK applications, Ontology layers, and pipelines are organized in a production-grade project. Recommended as a reference before setting up a new MSS application. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

The OSDK allows external Python or TypeScript applications to: - Query Ontology Object Types (with filtering, pagination, sorting) - Read and traverse Link Types (relationships between objects) - Execute Actions (triggering state changes in the Ontology) - Subscribe to real-time Ontology changes

The OSDK is not a direct database connection. It is a governed API that enforces CBAC, markings, and audit logging at every query. An external application using the OSDK cannot access data the authenticated user or service account is not authorized to see — this is the correct security behavior.

NOTE

The OSDK is the only approved method for external applications to interact with the MSS Ontology. Do not attempt to access Foundry datasets directly from external applications — use the OSDK for object data and the Platform SDK (Chapter 4) for dataset operations where approved.

NOTE — Palantir Defense OSDK Palantir offers a Defense OSDK with a pre-built Defense Ontology standardized to warfighting functions. Key properties: - **Consistency**: foundational data types aligned to WFF structure - **Adaptability**: API-like access deployable across environments (garrison, tactical, cloud) - **CJADC2 Compatibility**: designed for DoD interoperability requirements

Request access: palantir.com/request-defense-osdk | Defense SDK overview: palantir.com/defense/sdk

Source: Palantir Developer Community — Defense OSDK announcement

2-2. Authentication Architecture

CONDITIONS: You have been issued OSDK credentials (service account token or OAuth2 client credentials) through the C2DAO-approved process. You have a valid Foundry enrollment for your external application.

STANDARDS: External application authenticates to MSS OSDK endpoint using approved credential type. No credentials appear in source code or version control. Token rotation is automated or procedurally managed per AR 25-2.

EQUIPMENT: Approved development environment; credential store (environment variables or secrets manager); OSDK Python or TypeScript package installed.

Authentication types supported:

Type	Use Case	Notes
Personal Access Token (PAT)	Developer testing and local development	Never use in deployed applications
Service Account Token	Deployed server-side applications	Provisioned by C2DAO; rotate per policy
OAuth2 Confidential Client	Web applications with server-side token exchange	Requires C2DAO enrollment; preferred for web apps
OAuth2 Public Client (PKCE)	Single-page applications; user-delegated access	User's own permissions apply; no elevation

CAUTION: Personal Access Tokens authenticate as you. A PAT committed to a repository gives anyone with repository access your full Foundry permissions. Treat PATs as passwords. Never commit them.

PROCEDURE — Configure OSDK authentication (Python, service account):

1. Install the OSDK package for your enrolled application:

```
pip install foundry-sdk-python
# Or for a specific application ontology SDK:
pip install myapp-osdk
```

1. Store the service account token in environment variables — never in source code:

```
# .env file (never commit this file; add to .gitignore)
FOUNDRY_TOKEN=your_service_account_token_here
FOUNDRY_URL=https://mss.usareur.army.mil
```

1. Initialize the authentication client:

```
import os
from foundry import FoundryClient
```

```
from foundry.auth import ConfidentialClientAuth

# Load credentials from environment — never hardcode
token = os.environ["FOUNDRY_TOKEN"]
base_url = os.environ["FOUNDRY_URL"]

# Initialize client with token auth (service account pattern)
client = FoundryClient(
    auth=token,
    hostname=base_url,
)
```

1. Verify connectivity before proceeding with any operational query:

```
def verify_connection(client: FoundryClient) -> bool:
    """
    Verify OSDK client can reach MSS before beginning operations.
    Returns True if healthy; raises on failure.
    """
    try:
        # Lightweight ping — does not pull operational data
        client.ontology.ontologies.list()
        return True
    except Exception as exc:
        raise RuntimeError(
            f"MSS OSDK connection failed. Check FOUNDRY_TOKEN and FOUNDRY_URL. "
            f"Contact C2DAO if credentials are expired. Error: {exc}"
        ) from exc
```

1. For TypeScript applications, initialize the OSDK client:

```
import { createClient } from "@osdk/client";
import { createConfidentialOauthClient } from "@osdk/oauth";

// Load credentials from environment — never hardcode
const foundryUrl = process.env.FOUNDRY_URL!;
const clientId = process.env.OSDK_CLIENT_ID!;
const clientSecret = process.env.OSDK_CLIENT_SECRET!;

const auth = createConfidentialOauthClient(
  clientId,
  clientSecret,
  foundryUrl,
);

const client = createClient(
  foundryUrl,
  "<your-ontology-rid>",
  auth,
);
```

NOTE

OAuth2 confidential client is the preferred pattern for deployed server-side TypeScript applications. It allows token refresh without re-deployment. Coordinate with C2DAO for client ID and secret provisioning.

2-3. Querying Objects — Fundamentals

CONDITIONS: OSDK client initialized and authenticated. Target Object Type is enrolled in your application's ontology. You have read access to the target Object Type.

STANDARDS: All queries include explicit pagination. Filter logic does not retrieve data beyond operational need (least-privilege data access). Error handling covers authentication failures, network errors, and empty result sets.

EQUIPMENT: Authenticated OSDK client (Python or TypeScript); target Object Type enrolled in application ontology; OSDK package installed in development environment.

PROCEDURE — Basic object query (Python):

```
from foundry import FoundryClient
from foundry.models import ObjectSet

def get_unit_status_records(
    client: FoundryClient,
    unit_id: str,
) -> list[dict]:
    """
    Retrieve unit readiness status objects for a given unit ID.
    Applies filter to unit_id property — does not pull full object set.

    Args:
        client: Authenticated OSDK client
        unit_id: UIC string (e.g., "W12345")

    Returns:
        List of unit status dicts for downstream processing
    """
    results = []

    # Filter at query time — never pull all objects and filter in Python
    object_set = (
        client.ontology
        .objects["UnitStatus"]
        .filter({"property": "unitId", "type": "eq", "value": unit_id})
        .take(200)  # Explicit page size — never unbounded
    )

    for obj in object_set:
        results.append({
```

```
            "rid": obj.rid,
            "unit_id": obj.properties.get("unitId"),
            "readiness_level": obj.properties.get("readinessLevel"),
            "as_of_dtg": obj.properties.get("asOfDtg"),
            "reporting_officer": obj.properties.get("reportingOfficer"),
        })

    return results
```

PROCEDURE — Basic object query (TypeScript):

```
import { client } from "./client"; // your initialized OSDK client
import { UnitStatus } from "@your-app/osdk";

interface UnitStatusRecord {
  rid: string;
  unitId: string | undefined;
  readinessLevel: string | undefined;
  asOfDtg: string | undefined;
}

async function getUnitStatusRecords(
  unitId: string,
): Promise<UnitStatusRecord[]> {
  const results: UnitStatusRecord[] = [];

  // Filter server-side — do not pull full object population
  for await (const obj of client(UnitStatus)
    .where(UnitStatus.unitId.eq(unitId))
    .asyncIter()) {
    results.push({
      rid: obj.$primaryKey,
      unitId: obj.unitId,
      readinessLevel: obj.readinessLevel,
      asOfDtg: obj.asOfDtg,
    });
  }

  return results;
}
```

DRAFT

2-4. Pagination

BLUF: Every OSDK query that returns more than one object must implement explicit pagination. Unbounded queries against large operational object populations will time out or degrade platform performance.

CONDITIONS: OSDK client initialized and authenticated. Target Object Type has more than one result in the expected query scope.

STANDARDS: All multi-object queries use explicit page sizes. No unbounded queries against operational Object Types. Application handles empty result sets and end-of-page conditions without error.

EQUIPMENT: Authenticated OSDK client (Python or TypeScript); target Object Type enrolled in application.

PROCEDURE — Paginated query with page token (Python):

```
from typing import Generator

def paginated_equipment_query(
    client: FoundryClient,
    status_filter: str,
    page_size: int = 100,
) -> Generator[dict, None, None]:
    """
    Generator-based paginated query for equipment objects.
    Yields one object dict at a time to avoid building large in-memory lists.

    Args:
        client:        Authenticated OSDK client
        status_filter: Equipment status string (e.g., "FMC", "NMC")
        page_size:     Objects per page (max 200; default 100)

    Yields:
        Equipment object dicts
    """
    page_token = None

    while True:
        # Build the query with page parameters
        query_params = {
            "filter": {
                "property": "equipmentStatus",
                "type": "eq",
                "value": status_filter,
            },
            "pageSize": page_size,
        }
        if page_token:
            query_params["pageToken"] = page_token

        page = client.ontology.objects["Equipment"].list(**query_params)

        for obj in page.data:
            yield {
                "rid": obj.rid,
                "bumper_number": obj.properties.get("bumperNumber"),
                "equipment_status": obj.properties.get("equipmentStatus"),
                "owning_unit": obj.properties.get("owningUnit"),
                "last_pmcs_dtg": obj.properties.get("lastPmcsDtg"),
            }

        # Advance page or exit
```

```
        page_token = page.next_page_token
        if not page_token:
            break
```

NOTE

Set `page_size` to 100 as a default for operational queries. Reduce to 25–50 if objects have many properties or linked objects. Never exceed 200 per page. Consult C2DAO guidance for Object Types marked as high-cardinality.

2-5. Filtering and Sorting

CONDITIONS: OSDK client initialized and authenticated. Target Object Type is enrolled in the application and accessible to the authenticated principal.

STANDARDS: Filters are applied server-side. OR filters are profiled against representative data before deployment. Query logic retrieves only data within operational need (least-privilege access).

EQUIPMENT: Authenticated OSDK TypeScript client; target Object Type definitions imported from generated OSDK package.

PROCEDURE — Compound filters and sorting (TypeScript):

```
import { UnitStatus, Equipment } from "@your-app/osdk";

// Compound filter: multiple conditions (AND logic)
async function getNonMissionCapableEquipment(
  unitId: string,
  minDaysSinceLastMaintenance: number,
): Promise<Equipment[]> {
  const cutoffDate = new Date();
  cutoffDate.setDate(cutoffDate.getDate() - minDaysSinceLastMaintenance);

  const results: Equipment[] = [];

  for await (const obj of client(Equipment)
    .where({
      $and: [
        Equipment.owningUnit.eq(unitId),
        Equipment.equipmentStatus.eq("NMC"),
        Equipment.lastPmcsDtg.lt(cutoffDate.toISOString()),
      ],
    })
    .orderBy({ lastPmcsDtg: "asc" }) // Oldest maintenance first
    .asyncIter()) {
    results.push(obj);
  }

  return results;
```

```
}

// Filter with OR logic — use sparingly on large populations
async function getCriticalOrNmcEquipment(
  unitId: string,
): Promise<Equipment[]> {
  const results: Equipment[] = [];

  for await (const obj of client(Equipment)
    .where({
      $and: [
        Equipment.owningUnit.eq(unitId),
        {
          $or: [
            Equipment.equipmentStatus.eq("NMC"),
            Equipment.priority.eq("CRITICAL"),
          ],
        },
      ],
    })
    .asyncIter()) {
    results.push(obj);
  }

  return results;
}
```

CAUTION: OR filters on high-cardinality Object Types execute full scans on the affected properties. Profile with representative data before deploying OR-heavy filters to production. Prefer multiple targeted AND queries over broad OR scans when object populations exceed 10,000.

2-6. Traversing Link Types

CONDITIONS: OSDK client initialized and authenticated. Parent Object Type and Link Type are enrolled in the application. Authenticated principal has read access to both sides of the link.

STANDARDS: Link traversal is batched where possible. Single-object link traversal is not used inside loops over large object populations. Result is bounded and does not load unbounded child sets into memory.

EQUIPMENT: Authenticated OSDK Python client; parent Object Type RID; Link Type name.

PROCEDURE — Follow links from parent to child objects (Python):

```
def get_unit_with_equipment(
    client: FoundryClient,
    unit_rid: str,
) -> dict:
    """
```

```
    Retrieve a unit object and its linked equipment via the
    UnitToEquipment link type. Returns a structured dict.

    Link traversal adds a secondary query — do not call this
    in a loop over large unit populations without batching.
    """
    # Fetch the parent unit object
    unit = client.ontology.objects["Unit"].get(unit_rid)

    # Traverse the link to child equipment objects
    equipment_list = []
    for equip in unit.links["UnitToEquipment"].get_linked_objects():
        equipment_list.append({
            "rid": equip.rid,
            "bumper_number": equip.properties.get("bumperNumber"),
            "status": equip.properties.get("equipmentStatus"),
        })

    return {
        "unit_rid": unit.rid,
        "unit_name": unit.properties.get("unitName"),
        "uic": unit.properties.get("uic"),
        "equipment": equipment_list,
        "equipment_count": len(equipment_list),
    }
```

NOTE

Link traversal is a separate API call per object. For bulk link traversal across many parent objects, use the batch link query pattern described in Chapter 3.

CHAPTER 3 — OSDK ADVANCED PATTERNS

NOTE — Palantir Developers reference: *Ontology SDK x Lennar for Quality Inspection* — A production case study of OSDK deployed for quality inspection workflows at scale. Illustrates real-world OSDK architecture decisions and the patterns covered in Chapters 2–3. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Functions | How to Locate and Edit Objects from your Ontology in Foundry* — Demonstrates how to programmatically locate and interact with Ontology Objects in a code context, reinforcing the object access patterns used in OSDK queries. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

3-1. Action Execution via OSDK

BLUF: Actions are the approved mechanism for external applications to write state changes back into the MSS Ontology. Do not write to Ontology datasets directly. Use Actions.

CONDITIONS: Action is defined and deployed in the MSS Ontology by a -30 builder or -40 developer. Your OSDK application enrollment includes the Action in its API scope. The authenticated service account has permission to execute the Action.

STANDARDS: Action execution includes parameter validation before submission. Responses are handled for both success and error cases. All Action executions are logged at the application level for audit purposes.

EQUIPMENT: Authenticated OSDK client (Python or TypeScript); Action name and parameter schema obtained from -30 builder or Ontology documentation; application-level logger configured.

PROCEDURE — Execute Action (Python):

```
import logging
from datetime import datetime, timezone
from foundry.models import ActionResponse

logger = logging.getLogger(__name__)

def submit_sitrep_entry(
    client: FoundryClient,
    unit_rid: str,
    readiness_level: str,
    personnel_strength: int,
    equipment_pct: float,
    reporting_officer: str,
    dtg: str | None = None,
) -> ActionResponse:
    """
    Submit a SITREP entry via the SubmitSitrep Action.
    Validates parameters before submission and logs result.

    Args:
        client:              Authenticated OSDK client
        unit_rid:            RID of the Unit object
        readiness_level:     C-rating string ("C1", "C2", "C3", "C4")
        personnel_strength:  Assigned strength as integer
        equipment_pct:       Equipment readiness as float 0.0-1.0
        reporting_officer:   Name/ID of submitting officer
        dtg:                 ISO8601 DTG string; defaults to now (UTC)

    Returns:
        ActionResponse from MSS

    Raises:
        ValueError: On invalid parameter values (pre-flight check)
        RuntimeError: On Action execution failure
    """
```

```
    # Pre-flight parameter validation — fail fast before hitting the API
    valid_c_ratings = {"C1", "C2", "C3", "C4"}
    if readiness_level not in valid_c_ratings:
        raise ValueError(
            f"Invalid readiness_level '{readiness_level}'. "
            f"Must be one of: {valid_c_ratings}"
        )

    if not 0 <= personnel_strength <= 10000:
        raise ValueError(
            f"personnel_strength {personnel_strength} out of range [0, 10000]"
        )

    if not 0.0 <= equipment_pct <= 1.0:
        raise ValueError(
            f"equipment_pct {equipment_pct} out of range [0.0, 1.0]"
        )

    # Default DTG to current UTC if not provided
    if dtg is None:
        dtg = datetime.now(timezone.utc).strftime("%Y%m%dT%H%MZ")

    # Execute Action — parameters must match Action parameter schema
    try:
        response = client.ontology.actions["SubmitSitrep"].apply(
            unit=unit_rid,
            readinessLevel=readiness_level,
            personnelStrength=personnel_strength,
            equipmentReadinessPct=equipment_pct,
            reportingOfficer=reporting_officer,
            reportingDtg=dtg,
        )

        logger.info(
            "SubmitSitrep action executed: unit=%s readiness=%s dtg=%s",
            unit_rid, readiness_level, dtg,
        )
        return response

    except Exception as exc:
        logger.error(
            "SubmitSitrep action failed: unit=%s error=%s",
            unit_rid, exc,
        )
        raise RuntimeError(
            f"SubmitSitrep action failed for unit {unit_rid}: {exc}"
        ) from exc
```

PROCEDURE — Execute Action (TypeScript):

```
import { SubmitSitrep } from "@your-app/osdk";

interface SitrepPayload {
  unitRid: string;
```

```
  readinessLevel: "C1" | "C2" | "C3" | "C4";
  personnelStrength: number;
  equipmentReadinessPct: number;
  reportingOfficer: string;
  reportingDtg?: string;
}

async function submitSitrepEntry(
  payload: SitrepPayload,
): Promise<void> {
  // TypeScript type system enforces readiness level at compile time
  // Runtime validation for numeric ranges
  if (payload.personnelStrength < 0 || payload.personnelStrength > 10000) {
    throw new RangeError(
      `personnelStrength ${payload.personnelStrength} out of range [0, 10000]`,
    );
  }

  if (payload.equipmentReadinessPct < 0 || payload.equipmentReadinessPct > 1) {
    throw new RangeError(
      `equipmentReadinessPct must be between 0.0 and 1.0`,
    );
  }

  const dtg =
    payload.reportingDtg ?? new Date().toISOString();

  await client(SubmitSitrep).applyAction({
    unit: { $primaryKey: payload.unitRid },
    readinessLevel: payload.readinessLevel,
    personnelStrength: payload.personnelStrength,
    equipmentReadinessPct: payload.equipmentReadinessPct,
    reportingOfficer: payload.reportingOfficer,
    reportingDtg: dtg,
  });
}
```

3-2. Object Subscriptions — Real-Time Change Notifications

BLUF: OSDK subscriptions allow external applications to receive push notifications when Ontology objects change, eliminating polling loops against operational data.

NOTE — Palantir Developers reference: *Product Launch: Media, Real-Time Updates, and Expressive Compute in OSDK | DevCon 2* — Covers WebSocket-based real-time object updates and expressive compute patterns introduced in OSDK at DevCon 2, directly extending the subscription model described in this section. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

CONDITIONS: Target Object Type supports subscriptions (confirm with C2DAO). Application server can maintain a persistent WebSocket connection to MSS. Network path from application server to MSS is stable.

STANDARDS: Subscription handlers are idempotent — duplicate events do not cause duplicate state changes. Subscription errors are caught and logged. Application recovers from dropped connections with backoff retry.

EQUIPMENT: Authenticated OSDK TypeScript client with subscription support; stable network path to MSS with WebSocket allowed; application-level logger and error handler configured.

PROCEDURE — Subscribe to object changes (TypeScript):

```
import { UnitStatus } from "@your-app/osdk";

interface SubscriptionHandler {
  onUpdate: (obj: UnitStatus) => Promise<void>;
  onError: (error: Error) => void;
}

async function subscribeToUnitStatusChanges(
  unitIds: string[],
  handler: SubscriptionHandler,
): Promise<() => void> {
  /**
   * Subscribe to real-time UnitStatus object changes.
   * Returns an unsubscribe function — call it to clean up.
   *
   * Used by SITREP automation service to receive push updates
   * from MSS without polling.
   */
  const subscription = await client(UnitStatus)
    .where(UnitStatus.unitId.containsAnyTerm(unitIds))
    .subscribe({
      onChange: async ({ object, type }) => {
        if (type === "ADDED_OR_UPDATED") {
          try {
            // Handler must be idempotent — may receive same event twice
            await handler.onUpdate(object);
          } catch (err) {
            handler.onError(
              new Error(`Failed to process update for ${object.$primaryKey}: ${err}`),
            );
          }
        }
      },
      onOutOfDate: () => {
        // Subscription state is stale — re-query baseline then re-subscribe
        console.warn(
          "UnitStatus subscription out of date. Re-synchronizing baseline.",
        );
      },
      onError: (err) => {
        handler.onError(new Error(`Subscription error: ${err}`));
      },
    });

  // Return unsubscribe function for caller to invoke on shutdown
```

```
  return () => subscription.unsubscribe();
}
```

CAUTION: Subscription scope should be narrowed with filters when possible. An unfiltered subscription on a large operational Object Type (e.g., all Equipment in USAREUR-AF) will generate high event volume and can saturate application event queues. Filter to the AOR or unit population your application needs.

NOTE — Palantir Developers reference: *Product Launch: Edge Embedded Ontology* | *DevCon 2* — Covers a specialized deployment pattern where Ontology queries run at the edge without a round-trip to central infrastructure — relevant when designing OSDK applications that must operate in disconnected or low-latency environments. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

3-3. Bulk Object Operations

NOTE — Palantir Developers reference: *Code in Production: Lennar x MCP* | *DevCon 3* — Covers Model Context Protocol (MCP) integration with Foundry in a production deployment, an emerging pattern for enabling LLM tool use against the Ontology from external application contexts. Relevant to engineers building AI-integrated OSDK applications. Available on the Palantir Developers YouTube channel (@PalantirDevelopers). See **Chapter 10** for full MCP coverage.

BLUF: When processing multiple objects, use batch query patterns instead of per-object loops. Per-object queries (N+1 pattern) at operational scale will breach rate limits and degrade performance for all MSS users.

CONDITIONS: OSDK client initialized and authenticated. List of target object RIDs or filter criteria is known. Object population to process exceeds 10 objects (use single-object queries for smaller sets).

STANDARDS: No N+1 query pattern in production code. Batch size is bounded (50 RIDs per chunk default). Total result set is bounded before processing begins.

EQUIPMENT: Authenticated OSDK Python client; list of object RIDs or filter parameters.

PROCEDURE — Bulk load by RID list (Python):

```
from itertools import islice
from typing import Iterator

def chunked(iterable, size: int) -> Iterator[list]:
    """Split an iterable into fixed-size chunks."""
    it = iter(iterable)
    while chunk := list(islice(it, size)):
        yield chunk

def bulk_get_equipment_by_rid(
    client: FoundryClient,
    rids: list[str],
```

```
    chunk_size: int = 50,
) -> dict[str, dict]:
    """
    Retrieve multiple equipment objects by RID using chunked batch queries.

    The OSDK does not support arbitrary IN-clause queries by default.
    This pattern chunks the RID list and queries by filter in batches.

    Args:
        client:     Authenticated OSDK client
        rids:       List of equipment object RIDs
        chunk_size: RIDs per batch (keep <= 50 for safety)

    Returns:
        Dict mapping RID -> equipment property dict
    """
    result_map: dict[str, dict] = {}

    for chunk in chunked(rids, chunk_size):
        # Batch query using rid filter for this chunk
        for obj in client.ontology.objects["Equipment"].filter(
            {"property": "__rid", "type": "in", "value": chunk}
        ):
            result_map[obj.rid] = {
                "bumper_number": obj.properties.get("bumperNumber"),
                "equipment_status": obj.properties.get("equipmentStatus"),
                "owning_unit": obj.properties.get("owningUnit"),
            }

    return result_map
```

3-4. Error Handling and Retry Patterns

CONDITIONS: OSDK client initialized. Application must handle transient network errors and rate-limit responses without losing data or causing duplicate state changes.

STANDARDS: All OSDK calls are wrapped in error handling. Retries use exponential backoff. Authentication failures (401/403) are not retried — they are logged and raised immediately. Maximum retry attempts are bounded.

EQUIPMENT: Authenticated OSDK Python client; logging configured to capture retry attempts and final failures.

PROCEDURE — Retry with exponential backoff (Python):

```
import time
import logging
from typing import Callable, TypeVar

logger = logging.getLogger(__name__)
T = TypeVar("T")
```

```
def with_retry(
    func: Callable[[], T],
    max_attempts: int = 3,
    base_delay_seconds: float = 1.0,
    retryable_status_codes: set[int] | None = None,
) -> T:
    """
    Execute an OSDK call with exponential backoff retry.

    Retries on transient network errors and rate-limit responses (429).
    Does NOT retry on auth failures (401/403) or bad request (400).

    Args:
        func:                   Callable wrapping the OSDK operation
        max_attempts:           Maximum retry attempts (default 3)
        base_delay_seconds:     Initial retry delay; doubles each attempt
        retryable_status_codes: HTTP status codes to retry on (default: {429, 503})

    Returns:
        Result of func() on success

    Raises:
        Last exception after all retries exhausted
    """
    if retryable_status_codes is None:
        retryable_status_codes = {429, 503}

    last_exc: Exception | None = None

    for attempt in range(1, max_attempts + 1):
        try:
            return func()
        except Exception as exc:
            last_exc = exc

            # Check if this is a retryable error
            status_code = getattr(exc, "status_code", None)
            is_retryable = (
                status_code in retryable_status_codes
                or "timeout" in str(exc).lower()
                or "connection" in str(exc).lower()
            )

            if not is_retryable:
                logger.error("Non-retryable OSDK error: %s", exc)
                raise

            if attempt < max_attempts:
                delay = base_delay_seconds * (2 ** (attempt - 1))
                logger.warning(
                    "OSDK call failed (attempt %d/%d), retrying in %.1fs: %s",
                    attempt, max_attempts, delay, exc,
                )
                time.sleep(delay)
```

DRAFT

```
            else:
                logger.error(
                    "OSDK call failed after %d attempts: %s",
                    max_attempts, exc,
                )

    raise RuntimeError(
        f"OSDK operation failed after {max_attempts} attempts"
    ) from last_exc
```

3-5. OSDK Health Dialog — Error Visibility and Debugging

BLUF: OSDK errors now surface in the Health Dialog (Q1 2026). Use it as the first-line debugging tool for OSDK application failures before resorting to log analysis or network traces.

CONDITIONS: OSDK application is registered in the Foundry application enrollment. Health Dialog is accessible from the Foundry navigation bar (Administration > Health). Engineer has developer or admin permissions on the enrolled application.

STANDARDS: All OSDK application errors are reviewed through Health Dialog before escalating. Error patterns (repeated 403s, persistent timeouts, schema mismatches) are documented in the unit data ops log when they indicate systemic issues.

Background: Prior to Q1 2026, OSDK errors were only visible in application-side logs and network traces. Engineers had to correlate client-side HTTP status codes with server-side Foundry logs — a time-consuming process, especially for intermittent failures in production. The Health Dialog consolidates OSDK errors into a single, platform-native interface.

What the Health Dialog shows for OSDK applications:

Error Category	Information Displayed	Action
Authentication failures (401)	Token expiration time, token type, requesting application	Rotate token; verify enrollment is active
Authorization failures (403)	Object type, action, requesting principal, missing CBAC grant	Contact data steward; verify enrollment scope includes the object type
Schema mismatches	Expected vs. actual property types, object type version	Regenerate OSDK client against current Ontology; redeploy
Rate limit violations (429)	Request count, window, limit threshold	Reduce query frequency; implement batching (see 3-3)
Action validation failures	Validator name, validation message, input parameters	Review validator logic (see Chapter 6); check input data

Error Category	Information Displayed	Action
Connectivity / timeout errors	Endpoint, latency, failure count over time	Check network path; verify MSS platform status

PROCEDURE — Using Health Dialog to triage OSDK errors:

1. Navigate to **Administration > Health** in Foundry.
2. Select the enrolled OSDK application from the application list.
3. Review the error timeline — identify whether errors are transient (spikes) or persistent (flat elevated rate).
4. For each error category, expand the detail panel. The dialog provides the failing request context: object type, action name, requesting principal, and timestamp.
5. Cross-reference with application-side logs using the request timestamp and correlation ID (if your application logs correlation IDs per 9-4).
6. For 403 errors: verify the application enrollment includes the required Object Types and Actions. A common cause is an Ontology change that added a new Object Type consumed by the application but not yet included in the enrollment scope.
7. For schema mismatches: regenerate the OSDK client package against the current Ontology version and redeploy. Schema mismatches typically occur after an Ontology edit (property rename, type change) that the application has not yet adopted.

NOTE

Health Dialog is a platform feature — it does not require application code changes to enable. However, the quality of error context depends on proper OSDK client usage. Applications that swallow exceptions or retry silently may not surface errors to Health Dialog. Let errors propagate to the OSDK layer so the platform can capture them.

CAUTION

Health Dialog may display object type names, action names, and property names that are operationally sensitive. Access Health Dialog only from approved workstations. Do not screenshot or export Health Dialog data to uncontrolled systems.

3-6. Temporary Media Uploads via OSDK and Functions

BLUF: OSDK and Foundry Functions now support temporary media uploads (Q1 2026). Use this capability for user-submitted images, documents, and attachments that must be processed or attached to Ontology objects without requiring a permanent dataset file store.

CONDITIONS: OSDK client initialized and authenticated (TypeScript or Python). Target Action or Function accepts a media attachment parameter. Foundry enrollment includes the media upload capability for the application. File size does not exceed platform limits (check current limits in Foundry documentation — typically 100 MB per upload).

STANDARDS: Temporary media uploads are used for transient content that will be processed and either attached to an Ontology object or discarded. Do not use temporary uploads as a persistent storage mechanism — use Platform SDK dataset file resources (see 4-5) for permanent file storage. All uploaded media is validated for file type and size before submission. Sensitive media (CUI-marked images, classified documents) follows the same marking and handling requirements as any other MSS data.

Background: Prior to Q1 2026, attaching user-submitted files to Ontology objects required a multi-step workflow: upload to a staging dataset via Platform SDK, run a transform to extract metadata, then link to the target object. Temporary media uploads simplify this to a single OSDK call — the file is uploaded as part of an Action execution or Function invocation, and the platform handles temporary storage and cleanup.

Use cases in the USAREUR-AF context:

Use Case	Description	Object Type
SITREP photo attachments	Field units attach photos to SITREP submissions via external app	Sitrep
Equipment damage documentation	Maintenance teams upload damage photos during NMC reporting	Equipment
AAR supporting documents	After-action review submissions include scanned documents	AAREntry
ISR product ingestion	Imagery products attached to collection request objects	CollectionRequest

PROCEDURE — Temporary media upload with Action execution (TypeScript):

```
import { client } from "./foundryClient";
import { SubmitSitrepWithAttachment } from "@osdk/generated";

/**
 * Submit a SITREP with an attached photo via OSDK Action.
 *
 * The media file is uploaded as a temporary attachment — the platform
 * stores it for the duration of the Action processing window, then
 * the Action logic persists it to the appropriate object property.
 *
 * File validation occurs client-side before upload and server-side
 * in the Action validator.
 */
async function submitSitrepWithPhoto(
  unitId: string,
```

```
  sitrepData: SitrepPayload,
  photoFile: File,
): Promise<void> {
  // Validate file before upload — reject oversized or disallowed types
  const MAX_FILE_SIZE_MB = 25;
  const ALLOWED_TYPES = ["image/jpeg", "image/png", "application/pdf"];

  if (photoFile.size > MAX_FILE_SIZE_MB * 1024 * 1024) {
    throw new Error(
      `File exceeds ${MAX_FILE_SIZE_MB} MB limit: ${photoFile.name}`
    );
  }
  if (!ALLOWED_TYPES.includes(photoFile.type)) {
    throw new Error(
      `File type not allowed: ${photoFile.type}. Allowed: ${ALLOWED_TYPES.join(", ")}`
    );
  }

  // Execute Action with temporary media attachment
  await client(SubmitSitrepWithAttachment).applyAction({
    unitId,
    readinessLevel: sitrepData.readinessLevel,
    narrativeSummary: sitrepData.narrative,
    attachment: photoFile,  // Temporary media upload — platform handles storage
  });
}
```

PROCEDURE — Temporary media upload in a Foundry Function (TypeScript):

```
import {
  Function,
  Attachment,
  OntologyEditFunction,
} from "@foundry/functions-api";
import { Objects } from "@foundry/ontology-api";

/**
 * Process an uploaded equipment damage photo.
 * The Function receives the temporary media upload, validates it,
 * and attaches it to the Equipment object.
 */
export class EquipmentFunctions {
  @OntologyEditFunction()
  public async attachDamagePhoto(
    equipment: Objects.Equipment,
    damagePhoto: Attachment,
    description: string,
  ): Promise<void> {
    // Platform provides the Attachment handle for the temporary upload
    // Validate attachment metadata
    const metadata = await damagePhoto.getMetadata();
    if (metadata.sizeBytes > 25 * 1024 * 1024) {
      throw new Error("Attachment exceeds 25 MB limit");
    }
```

```
    // Attach to Equipment object — platform persists from temporary storage
    equipment.damagePhotoAttachment = damagePhoto;
    equipment.damageDescription = description;
    equipment.lastInspectionDtg = new Date().toISOString();
  }
}
```

NOTE

Temporary media uploads have a platform-defined retention window. If the Action or Function does not persist the attachment to an object property or dataset, the temporary file is automatically deleted after the retention period expires. Do not rely on temporary storage for any data that must be retained.

CAUTION

Uploaded media inherits the marking of the target object. If a user uploads an uncontrolled image to a CUI-marked Equipment object, the image becomes CUI upon attachment. Brief users on marking implications before enabling media upload in operational applications.

3-7. Pilot — AI-Assisted OSDK Application Development

BLUF: Pilot (launched March 2026) is an AI-powered tool that generates React application scaffolding, components, and OSDK integration code. Use Pilot to accelerate initial application development; do not use it as a substitute for engineering review and production hardening.

CONDITIONS: Pilot is enabled in the Foundry environment. Engineer has developer permissions and an active OSDK application enrollment. Engineer has working knowledge of React, TypeScript, and OSDK patterns (Chapters 2–3 of this manual).

STANDARDS: All Pilot-generated code undergoes the same review, testing, and CI/CD standards as manually written code (see Chapter 8). Pilot output is a starting point — not a deployable artifact. Engineers must validate generated OSDK queries for correctness, CBAC compliance, error handling, and performance before promotion.

What Pilot generates:

Output	Description	Engineer Review Required
React project scaffolding	Project structure, build config, dependency setup	Verify dependency versions; confirm no unnecessary packages
OSDK client initialization	Authentication setup, client configuration	Verify auth pattern matches deployment model (BFF vs. SPA); confirm credential handling

Output	Description	Engineer Review Required
Object query components	React components that query and display Ontology objects	Validate filters, pagination, error handling; confirm CBAC scoping
Action execution forms	Form components that execute Actions via OSDK	Validate input sanitization, validation logic, error messaging
Link traversal views	Components that navigate object relationships	Check for N+1 patterns; confirm batch loading (see 3-3)

When to use Pilot vs. manual development:

Scenario	Recommendation	Rationale
New application from scratch	**Use Pilot** for initial scaffolding	Saves hours on boilerplate; ensures standard project structure
Rapid prototype for stakeholder demo	**Use Pilot** for quick component generation	Accelerates time to visual output; prototype code is disposable
Adding features to existing production app	**Manual development**	Pilot does not understand existing app architecture, custom patterns, or team conventions
Complex multi-step Action workflows	**Manual development**	Pilot generates simple Action forms; multi-step flows (Chapter 6) require domain-specific validation logic
Security-critical integration code	**Manual development**	Credential handling, webhook verification, and CBAC enforcement require deliberate engineering (Chapter 9)
Performance-sensitive bulk operations	**Manual development**	Pilot may generate naive query patterns; bulk patterns (3-3) require intentional optimization

PROCEDURE — Using Pilot to scaffold an OSDK application:

1. Open Pilot from the Foundry Developer Console.
2. Describe the application in natural language — include the target Object Types, Actions, and the user-facing purpose. Example: *"Build a React app that displays Equipment objects filtered by owning unit, shows readiness status, and allows users to submit maintenance requests via the SubmitMaintenanceRequest Action."*
3. Review the generated project structure. Confirm it includes: OSDK client setup, authentication configuration, component hierarchy, and build scripts.
4. **Validate OSDK client initialization.** Confirm the auth pattern is correct for the deployment target (OAuth2 PKCE for SPAs; service account for server-rendered BFF). Pilot defaults may not match your deployment model.

5. **Validate generated queries.** Open each component that queries the Ontology. Confirm: explicit page size limits are set (see 2-4); filters match the intended data scope; error handling wraps all OSDK calls (see 3-4).
6. **Validate Action forms.** Confirm: input fields are validated and sanitized; Action parameters match the current Ontology Action definition; user feedback (success/error) is displayed.
7. **Add missing production requirements.** Pilot-generated code typically lacks: audit logging (see 9-4); marking display for CUI/coalition data (see 9-3); retry logic for transient errors (see 3-4); CBAC-aware error messaging (see 9-1).
8. Run all CI/CD checks (lint, type check, unit tests) before any branch promotion (see Chapter 8).

WARNING

Pilot generates code based on the current Ontology schema. If the Ontology changes after generation, Pilot-generated components may reference stale Object Types, properties, or Actions. Always regenerate the OSDK client and validate components after Ontology changes.

CAUTION

Do not input operational data, real unit names, or classified information into Pilot prompts. Use generic descriptions and placeholder data during generation. Replace with operational specifics only in the local development environment.

CHAPTER 4 — FOUNDRY PLATFORM SDK

4-1. What Is the Platform SDK

BLUF: The Foundry Platform SDK (Python) provides programmatic access to Foundry datasets, branches, transactions, and file resources.

NOTE — Palantir Developers reference: *Code Repositories | How to Write Data Transformations in Palantir Foundry* — Covers the core procedure for writing data transforms in Foundry Code Repositories, which underlies Platform SDK dataset write patterns. Relevant to anyone building pipeline logic that feeds MSS datasets. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Code Repositories | How to Create a Python Library in Palantir Foundry* — Demonstrates how to package shared transform logic as a reusable Python library within Foundry, applicable when building shared utilities for MSS data pipelines. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Foundry Reference Project | Data Pipeline* — Walks through the data pipeline layer of the Foundry Reference Project, showing how Platform SDK dataset operations fit into a production pipeline architecture. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Developer Deskside | Building Apps on Kafka Streaming Data in Palantir Foundry* — Covers patterns for building Foundry applications that consume Kafka streaming data, an advanced use case for Platform SDK integration in real-time operational pipelines. Available on the Palantir Developers YouTube channel (@PalantirDevelopers). Use it when you need to read or write tabular data, manage dataset transactions, or access files stored in Foundry. Use the OSDK (Chapters 2–3) for Ontology object access.

The Platform SDK is not a general-purpose data lake client. All access must be authorized by the C2DAO data steward for the target dataset. Do not use the Platform SDK to bypass CBAC — the SDK enforces the same access controls as the Foundry UI.

CAUTION: Writing to production datasets via the Platform SDK without coordination with the owning data steward is not authorized. Always write to development branches and promote through the standard review process. Chapter 8 covers promotion workflows.

4-2. Client Initialization and Authentication

CONDITIONS: Service account token and Foundry base URL are provisioned and stored in environment variables. Platform SDK Python package is installed in the development environment.

STANDARDS: Client is initialized from environment variables only. No credentials appear in source code. Connection is verified before any dataset operation begins.

EQUIPMENT: Approved development environment; `FOUNDRY_TOKEN` and `FOUNDRY_URL` environment variables set from C2DAO-approved credential store; Platform SDK Python package installed.

PROCEDURE — Initialize the Platform SDK client (Python):

```
import os
from foundry import FoundryClient

def build_platform_client() -> FoundryClient:
    """
    Build and return an authenticated Foundry Platform SDK client.
    Credentials loaded from environment — never hardcoded.

    Required environment variables:
        FOUNDRY_TOKEN:  Service account or PAT token
        FOUNDRY_URL:    MSS base URL
    """
    token = os.environ.get("FOUNDRY_TOKEN")
    base_url = os.environ.get("FOUNDRY_URL")
```

```
    if not token or not base_url:
        raise EnvironmentError(
            "FOUNDRY_TOKEN and FOUNDRY_URL must be set in environment. "
            "See team credential store for approved service account values."
        )

    return FoundryClient(auth=token, hostname=base_url)
```

4-3. Dataset Operations — Reading

CONDITIONS: Platform SDK client initialized and authenticated. Dataset RID is known. Authenticated principal has read access to the target dataset and branch. Data steward authorization obtained for production reads in automated pipelines.

STANDARDS: Large dataset reads use column selection or row limits. Full dataset reads are not performed on production datasets without data steward coordination. Schema is validated before downstream processing.

EQUIPMENT: Authenticated Platform SDK Python client; dataset RID; target branch name; `pandas` and `pyarrow` packages installed.

PROCEDURE — Read a Foundry dataset into a pandas DataFrame (Python):

```
import pandas as pd
from foundry import FoundryClient

def read_dataset(
    client: FoundryClient,
    dataset_rid: str,
    branch: str = "master",
    columns: list[str] | None = None,
) -> pd.DataFrame:
    """
    Read a Foundry dataset into a pandas DataFrame.

    Args:
        client:      Authenticated Platform SDK client
        dataset_rid: RID of the target dataset
        branch:      Branch to read from (default "master" = production)
        columns:     Column subset to read; reads all if None

    Returns:
        pandas DataFrame with dataset contents

    Notes:
        - For large datasets, use the row_limit or filter params
          to avoid loading full production datasets into memory.
        - Always confirm dataset schema with data steward before
          reading in production pipelines.
    """
```

```
    dataset = client.datasets.Dataset.get(dataset_rid)

    # Read via the Parquet/Arrow interface for performance
    df = dataset.read_table(
        branch_name=branch,
        columns=columns,
    )

    return df.to_pandas()


def read_dataset_with_limit(
    client: FoundryClient,
    dataset_rid: str,
    row_limit: int = 10000,
) -> pd.DataFrame:
    """
    Read a bounded row sample from a dataset.
    Use for validation, profiling, and dev/test workflows.
    Never use unlimited reads on large production datasets.
    """
    dataset = client.datasets.Dataset.get(dataset_rid)

    # Read first N rows only
    table = dataset.read_table(
        branch_name="master",
        start_transaction_rid=None,
        end_transaction_rid=None,
        row_limit=row_limit,
    )

    return table.to_pandas()
```

4-4. Dataset Operations — Writing with Transactions

BLUF: All writes to Foundry datasets use transactions.

NOTE — Palantir Developers reference: *Code Repositories* | *How to Write Incremental Data Transforms in Palantir Foundry* — Explains the incremental transform pattern for efficient data pipelines that only process new or changed data, reducing compute cost compared to full-dataset rewrites. Directly relevant to the APPEND vs. SNAPSHOT transaction choice described in this section. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Spark Basics* | *Partitions* — Covers Spark partition fundamentals, which govern how data is physically organized and processed in Foundry's Spark-based dataset layer. Relevant when optimizing write performance for large datasets. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Spark Basics | Shuffling* — Explains shuffle operations in Spark, the primary source of performance bottlenecks in distributed data transforms. Understanding shuffle is essential when diagnosing slow dataset write or transformation jobs. Available on the Palantir Developers YouTube channel (@PalantirDevelopers). A transaction is an atomic unit of work — either all writes in a transaction commit or none do. Always open a transaction before writing and close it explicitly.

Transaction types:

Type	Use Case	Behavior
`APPEND`	Add new rows to existing dataset	Does not modify existing data
`UPDATE`	Replace dataset contents	Replaces all rows (destructive — coordinate with steward)
`SNAPSHOT`	Replace full dataset atomically	Creates new snapshot; preferred for full-refresh pipelines
`DELETE`	Remove specific rows	Use sparingly; coordinate with data steward

CAUTION

APPEND transactions are NOT inherently idempotent. Each APPEND call adds data to the dataset without deduplication. To achieve idempotent writes, implement deduplication logic yourself using content hashes or surrogate keys before appending. Use SNAPSHOT transactions for full-dataset atomic replacement.

CONDITIONS: Platform SDK client initialized and authenticated. Target dataset RID is known. Authenticated principal has write access to the target branch. Data steward authorization obtained for any write to a shared staging or production dataset. Deduplication logic is implemented if APPEND transaction is used.

STANDARDS: All writes use transactions. Transactions are explicitly committed or aborted — no open transactions left on error. Writes target `develop` branch by default; writes to `master` require explicit steward coordination. APPEND transactions include deduplication logic to ensure idempotency.

EQUIPMENT: Authenticated Platform SDK Python client; dataset RID; target branch name; `pandas` and `pyarrow` packages installed.

PROCEDURE — Write to dataset with APPEND transaction (Python):

```
import pandas as pd
import pyarrow as pa
from foundry import FoundryClient

def append_sitrep_records(
    client: FoundryClient,
    dataset_rid: str,
    records: list[dict],
    branch: str = "develop",
) -> str:
```

```
    """
    Append SITREP records to a staging dataset via APPEND transaction.

    Writes to the 'develop' branch by default — never write directly
    to master without C2DAO coordination and steward approval.

    Args:
        client:      Authenticated Platform SDK client
        dataset_rid: RID of the target staging dataset
        records:     List of dicts matching the dataset schema
        branch:      Target branch (default "develop")

    Returns:
        Transaction RID for audit logging

    Raises:
        ValueError: If records list is empty
        RuntimeError: On transaction failure
    """
    if not records:
        raise ValueError("records list is empty — nothing to append")

    df = pd.DataFrame(records)
    table = pa.Table.from_pandas(df)

    dataset = client.datasets.Dataset.get(dataset_rid)

    # Open transaction — all writes are atomic within this context
    transaction = dataset.start_transaction(
        branch_name=branch,
        transaction_type="APPEND",
    )

    try:
        # Write the Arrow table to the transaction
        transaction.write_table(table)

        # Commit the transaction — data is now visible on the branch
        transaction.commit()

        return transaction.rid

    except Exception as exc:
        # Abort on any error — do not leave open transactions
        try:
            transaction.abort()
        except Exception:
            pass  # Best-effort abort
        raise RuntimeError(
            f"Dataset write failed, transaction aborted: {exc}"
        ) from exc


def snapshot_dataset(
    client: FoundryClient,
```

```
    dataset_rid: str,
    new_data: pd.DataFrame,
    branch: str = "develop",
) -> str:
    """
    Replace full dataset contents via SNAPSHOT transaction.
    Use for full-refresh pipelines that replace prior data each run.

    WARNING: This replaces ALL existing data on the target branch.
    Coordinate with the data steward before using on shared datasets.
    """
    table = pa.Table.from_pandas(new_data)
    dataset = client.datasets.Dataset.get(dataset_rid)

    transaction = dataset.start_transaction(
        branch_name=branch,
        transaction_type="SNAPSHOT",
    )

    try:
        transaction.write_table(table)
        transaction.commit()
        return transaction.rid
    except Exception as exc:
        try:
            transaction.abort()
        except Exception:
            pass
        raise RuntimeError(
            f"Snapshot transaction failed, aborted: {exc}"
        ) from exc
```

4-5. File Resources

NOTE — Palantir Developers reference: *Code Repositories* | *How to Consume a Library in Palantir Foundry* — Shows how to import and use a Foundry Python library in a transform or pipeline, the complement to the library creation pattern. Relevant when integrating shared utility code into Platform SDK-based pipelines. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

CONDITIONS: Platform SDK client initialized and authenticated. Target dataset RID is known. Authenticated principal has read or write access to the target dataset file store. File content is validated before upload.

STANDARDS: File uploads use APPEND transactions and are committed or aborted explicitly. File paths are well-defined logical paths — not arbitrary file system paths. Sensitive file content is not logged.

EQUIPMENT: Authenticated Platform SDK Python client; dataset RID; target branch name; file content as bytes.

PROCEDURE — Read and write file resources in Foundry (Python):

```
import io
from foundry import FoundryClient

def upload_report_file(
    client: FoundryClient,
    dataset_rid: str,
    file_content: bytes,
    file_path: str,
    branch: str = "develop",
) -> None:
    """
    Upload a file (PDF, CSV, JSON) to a Foundry dataset's file store.

    Used to attach generated reports, exports, or reference files
    to a dataset resource without writing tabular data.

    Args:
        client:       Authenticated Platform SDK client
        dataset_rid:  RID of target dataset
        file_content: Raw bytes of the file to upload
        file_path:    Logical path within the dataset file store
        branch:       Target branch
    """
    dataset = client.datasets.Dataset.get(dataset_rid)

    transaction = dataset.start_transaction(
        branch_name=branch,
        transaction_type="APPEND",
    )

    try:
        transaction.put_file(
            logical_path=file_path,
            file_data=io.BytesIO(file_content),
        )
        transaction.commit()
    except Exception as exc:
        try:
            transaction.abort()
        except Exception:
            pass
        raise RuntimeError(f"File upload failed: {exc}") from exc


def download_reference_file(
    client: FoundryClient,
    dataset_rid: str,
    file_path: str,
) -> bytes:
    """
    Download a reference file from a Foundry dataset file store.

    Returns:
        Raw bytes of the file
    """
```

```
    dataset = client.datasets.Dataset.get(dataset_rid)
    file_handle = dataset.get_file(logical_path=file_path)
    return file_handle.read()
```

4-6. Branch Management

CONDITIONS: Platform SDK client initialized and authenticated. Dataset RID is known. Authenticated principal has branch management permissions on the target dataset.

STANDARDS: Development branches are created from `master`. Branch names follow the naming convention in NAMING_AND_GOVERNANCE_STANDARDS. Branches are not left open indefinitely after development work completes.

EQUIPMENT: Authenticated Platform SDK Python client; dataset RID; desired branch name.

PROCEDURE — List branches and create a development branch (Python):

```
def get_or_create_dev_branch(
    client: FoundryClient,
    dataset_rid: str,
    branch_name: str,
) -> str:
    """
    Get an existing branch or create it from master.
    Standard pattern for setting up a development branch
    before writing to a dataset.

    Returns:
        Branch name (string) — usable in subsequent SDK calls
    """
    dataset = client.datasets.Dataset.get(dataset_rid)

    existing_branches = {b.name for b in dataset.list_branches()}

    if branch_name in existing_branches:
        return branch_name

    # Create from master — the development branch inherits production state
    dataset.create_branch(
        branch_name=branch_name,
        source_branch_name="master",
    )

    return branch_name
```

CHAPTER 5 — TYPESCRIPT FUNCTIONS ON OBJECTS (FOO)

5-1. What Are Functions on Objects

BLUF: Functions on Objects (FOO) are TypeScript functions deployed within Foundry that compute derived properties, aggregate data, or perform transformations directly on Ontology objects at query time.

NOTE — Palantir Developers reference: *Functions* | *Getting Started* — Introductory walkthrough of the Foundry Functions feature, covering repository structure, basic function authoring, and how Functions integrate with the Ontology. Recommended as the first reference before implementing FOO in this chapter. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Foundry Reference Project* | *Ontology* — Covers the Ontology layer of the Foundry Reference Project, showing how Functions, Object Types, and computed properties are organized in a production Ontology. Complements the repository structure described in Section 5-2. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

FOO is the correct mechanism for: - Computed properties on objects (e.g., readiness score derived from component statuses) - Aggregations across object populations (e.g., total equipment count by unit and status) - Bulk transformations that need access to linked objects - Custom search logic not expressible via standard OSDK filters

FOO is not appropriate for: - Long-running batch computations — use Pipeline Builder transforms - External API calls — FOO runs in a sandboxed environment - Stateful operations — FOO functions are stateless

NOTE

FOO functions are deployed as Foundry code resources in a TypeScript repository. They are subject to the same CI/CD and governance requirements as any production code resource. Chapter 8 covers repository and deployment discipline.

5-2. FOO Repository Structure

A standard FOO repository structure:

```
my-foo-functions/
├── src/
│   ├── index.ts            # Function exports — all functions registered here
│   ├── readiness/
│   │   ├── readinessScore.ts
│   │   ├── equipmentAggregate.ts
│   │   └── __tests__/
```

```
│   │       ├── readinessScore.test.ts
│   │       └── equipmentAggregate.test.ts
│   └── utils/
│       ├── ratingHelpers.ts
│       └── dateHelpers.ts
├── package.json
├── tsconfig.json
└── foundry.json            # Foundry project configuration
```

5-3. Computed Property Functions

CONDITIONS: TypeScript FOO repository is configured per Section 5-2. Target Object Type is registered in the Foundry Ontology. Computation cost is assessed as acceptable for on-demand execution (see decision framework in Concepts Guide Section 5).

STANDARDS: FOO function returns a defined type. Null/undefined property values are handled without throwing. Function is stateless — no module-level state accumulated across calls. Unit test coverage meets minimum per Section 8-4.

EQUIPMENT: TypeScript development environment; `@foundry/functions-api` and `@foundry/ontology-api` packages; Jest test framework.

PROCEDURE — Define a computed property FOO function (TypeScript):

```typescript
import { Function, Double, String } from "@foundry/functions-api";
import {
  UnitStatus,
  Equipment,
  MaintenanceRecord,
} from "@foundry/ontology-api";

// Computed property: overall unit readiness score (0.0 - 1.0)
// Factors: personnel fill rate, equipment FMC rate, maintenance current
export class ReadinessFunctions {
  @Function()
  computeUnitReadinessScore(unit: UnitStatus): Double {
    /**
     * Compute a composite readiness score for a unit.
     * Score = (personnel fill * 0.4) + (equipment FMC * 0.4) + (maint current * 0.2)
     *
     * Returns 0.0 if any required property is missing.
     * This score feeds the Readiness Dashboard Workshop application.
     */
    const personnelFill = unit.personnelFillRate ?? 0;
    const equipmentFmc = unit.equipmentFmcRate ?? 0;
    const maintCurrent = unit.maintenanceCurrentPct ?? 0;

    // Weighted composite score
    const score =
      personnelFill * 0.4 + equipmentFmc * 0.4 + maintCurrent * 0.2;
```

```
    // Clamp to [0.0, 1.0] — defensive against data anomalies
    return Math.max(0.0, Math.min(1.0, score));
  }

  @Function()
  computeReadinessRating(unit: UnitStatus): String {
    /**
     * Convert numeric readiness score to C-rating string.
     * Thresholds align with FM 7-0 readiness standards.
     */
    const score = this.computeUnitReadinessScore(unit);

    if (score >= 0.9) return "C1";
    if (score >= 0.75) return "C2";
    if (score >= 0.5) return "C3";
    return "C4";
  }
}
```

5-4. Bulk Query Patterns

BLUF: FOO bulk query functions execute server-side against the full object population. Design them to minimize object property access and avoid N+1 link traversals in hot paths.

CONDITIONS: Object Type is registered in the Foundry Ontology. ObjectSet passed to the function is pre-filtered by the caller to the relevant population. Computation is assessed as scalable against production object volumes.

STANDARDS: Function accesses only the properties it needs. No N+1 link traversal inside object iteration loops. Function is profiled against production-scale data volumes before deployment. Returns empty/zero result for empty ObjectSet without error.

EQUIPMENT: TypeScript development environment; `@foundry/functions-api` and `@foundry/ontology-api` packages; production-scale test data for profiling.

PROCEDURE — Aggregate function across multiple objects (TypeScript):

```
import {
  Function,
  FunctionsMap,
  Integer,
  Double,
} from "@foundry/functions-api";
import { Equipment, ObjectSet } from "@foundry/ontology-api";

export class EquipmentAggregateFunctions {
  @Function()
  countEquipmentByStatus(
    equipment: ObjectSet<Equipment>,
```

```
  ): FunctionsMap<string, Integer> {
    /**
     * Count equipment objects grouped by status (FMC, NMC, PMC, etc.).
     * Returns a map of status -> count.
     *
     * Called from Workshop aggregation widgets and OSDK analytics queries.
     * Runs server-side — efficient for large equipment populations.
     */
    const counts = new Map<string, number>();

    for (const equip of equipment) {
      const status = equip.equipmentStatus ?? "UNKNOWN";
      counts.set(status, (counts.get(status) ?? 0) + 1);
    }

    return counts;
  }

  @Function()
  computeFleetFmcRate(
    equipment: ObjectSet<Equipment>,
  ): Double {
    /**
     * Compute fleet-level FMC rate for a given equipment ObjectSet.
     * ObjectSet is pre-filtered by caller (e.g., by owning unit or type).
     *
     * Returns 0.0 for empty sets to avoid division-by-zero.
     */
    let total = 0;
    let fmcCount = 0;

    for (const equip of equipment) {
      total++;
      if (equip.equipmentStatus === "FMC") {
        fmcCount++;
      }
    }

    if (total === 0) return 0.0;
    return fmcCount / total;
  }
}
```

5-5. FOO Performance Patterns

Critical rules for FOO performance:

Rule	Correct Pattern	Anti-Pattern
Avoid N+1 link traversal	Batch link loading where API supports it	Traversing links inside a loop over object population

Rule	Correct Pattern	Anti-Pattern
Minimize property access	Access only needed properties	Reading all properties on every object
Return early	Exit on null/missing data immediately	Processing through null checks deep in nested logic
Use ObjectSet filters	Let Foundry pre-filter before FOO receives objects	Receiving full population and filtering in TypeScript
Stateless design	Compute from object state only	Caching or accumulating state across function calls

PROCEDURE — Performance-optimized bulk link aggregation (TypeScript):

```
import { Function, FunctionsMap, Integer } from "@foundry/functions-api";
import { Unit, Equipment, ObjectSet } from "@foundry/ontology-api";

export class UnitEquipmentFunctions {
  @Function()
  countNmcEquipmentPerUnit(
    units: ObjectSet<Unit>,
  ): FunctionsMap<Unit, Integer> {
    /**
     * For each unit in the set, count NMC equipment via the
     * UnitToEquipment link.
     *
     * Performance note: Uses bulk link loading via the ObjectSet API
     * rather than per-object traversal. Reduces round-trips.
     */
    const result = new Map<Unit, number>();

    for (const unit of units) {
      // Link traversal here is batched by the Foundry runtime
      // when iterating an ObjectSet — this is more efficient than
      // calling unit.unitToEquipment.get() in a separate loop
      const nmcCount = unit.unitToEquipment
        .filter((e) => e.equipmentStatus === "NMC")
        .count();

      result.set(unit, nmcCount);
    }

    return result;
  }
}
```

5-6. Testing FOO Functions

NOTE — Palantir Developers reference: *Functions | Unit Testing Functions on Objects in Palantir Foundry* — Covers unit testing patterns for Functions on Objects, including mock object construction and Jest test setup — directly reinforces the test suite requirements in this section. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Code Repositories | How to Unit Test PySpark in Palantir Foundry* — Covers PySpark unit testing patterns, applicable when FOO functions interact with Spark-backed data or when testing Python transform logic alongside TypeScript FOO. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

CONDITIONS: FOO function is implemented per Section 5-3 or 5-4. Jest test framework is configured in the repository. Mock object stubs can be constructed from known property values without Foundry connectivity.

STANDARDS: At least one test validates correct output for a known input. Edge cases covered: null/undefined properties, empty object sets, boundary values for numeric computations. Test coverage meets 80% line minimum per Section 8-4. Tests run locally without Foundry connectivity.

EQUIPMENT: TypeScript development environment; Jest configured; `@foundry/ontology-api` mock types.

PROCEDURE — Unit test FOO functions (TypeScript, Jest):

```
import { ReadinessFunctions } from "../readiness/readinessScore";
import { UnitStatus } from "@foundry/ontology-api";

// Mock factory — create a minimal UnitStatus stub
function makeUnitStatusStub(
  personnelFillRate: number,
  equipmentFmcRate: number,
  maintenanceCurrentPct: number,
): UnitStatus {
  return {
    personnelFillRate,
    equipmentFmcRate,
    maintenanceCurrentPct,
  } as unknown as UnitStatus;
}

describe("ReadinessFunctions", () => {
  const fns = new ReadinessFunctions();

  describe("computeUnitReadinessScore", () => {
    test("returns 1.0 for fully capable unit", () => {
      const unit = makeUnitStatusStub(1.0, 1.0, 1.0);
      expect(fns.computeUnitReadinessScore(unit)).toBeCloseTo(1.0);
    });

    test("returns correct weighted composite score", () => {
```

```
      // 0.8 fill * 0.4 + 0.75 FMC * 0.4 + 0.9 maint * 0.2
      // = 0.32 + 0.30 + 0.18 = 0.80
      const unit = makeUnitStatusStub(0.8, 0.75, 0.9);
      expect(fns.computeUnitReadinessScore(unit)).toBeCloseTo(0.8);
    });

    test("returns 0.0 for fully degraded unit", () => {
      const unit = makeUnitStatusStub(0, 0, 0);
      expect(fns.computeUnitReadinessScore(unit)).toBeCloseTo(0.0);
    });

    test("clamps to 1.0 if properties exceed 1.0 (data anomaly)", () => {
      const unit = makeUnitStatusStub(1.5, 1.2, 1.1);
      expect(fns.computeUnitReadinessScore(unit)).toBeLessThanOrEqual(1.0);
    });
  });

  describe("computeReadinessRating", () => {
    test("assigns C1 for score >= 0.9", () => {
      const unit = makeUnitStatusStub(1.0, 1.0, 1.0);
      expect(fns.computeReadinessRating(unit)).toBe("C1");
    });

    test("assigns C4 for score < 0.5", () => {
      const unit = makeUnitStatusStub(0.2, 0.3, 0.1);
      expect(fns.computeReadinessRating(unit)).toBe("C4");
    });
  });
});
```

CHAPTER 6 — ACTIONS WITH COMPLEX VALIDATION

6-1. Action Validation Architecture

BLUF: Actions in Foundry can include TypeScript validation functions that run before the Action executes. Complex validation logic, multi-step workflows, and conditional action flows require TypeScript — this is a SL 4L responsibility, not SL 3.

The three validation layers for complex Actions:

Layer	Where	Who Writes	Purpose
Client-side (pre-submit)	External app or Workshop	-40L SWE	UX validation; catch obvious errors before API call
OSDK validation	Action parameter schema	Foundry runtime	Type enforcement; required fields

Layer	Where	Who Writes	Purpose
Server-side TypeScript validator	Foundry Function	-40L SWE	Business rules, cross-object validation, conditional logic

NOTE

Slate is Foundry's legacy application builder and is no longer the recommended path for new development. Use Workshop for internal Foundry applications, or OSDK-backed external applications for public-facing deployments. For client-side validation in new development, implement validation in your Workshop application or external OSDK application — not in a Slate app.

TypeScript server-side validators execute inside Foundry before the Action applies state changes. A validator returning an error prevents the Action from executing.

6-2. Writing TypeScript Action Validators

CONDITIONS: Action is defined in the Foundry Ontology by a -30 builder. Business rules for the Action are documented and reviewed by the responsible data steward. TypeScript FOO repository is configured per Section 5-2.

STANDARDS: Validation logic is separated from the Action function for testability. All business rules are enforced before the Action applies state changes. Throwing inside an `@ActionEditFunction` prevents execution. Unit test coverage meets 90% line minimum (validators are security-sensitive — higher bar per Section 8-4).

EQUIPMENT: TypeScript development environment; `@foundry/functions-api` and `@foundry/ontology-api` packages; Jest test framework; documented business rules from data steward.

PROCEDURE — TypeScript Action validator for SITREP submission:

```
import {
  ActionEditFunction,
  ActionInput,
  BooleanType,
  StringType,
  Edits,
} from "@foundry/functions-api";
import { UnitStatus, Sitrep } from "@foundry/ontology-api";

interface SitrepValidationInput {
  unit: UnitStatus;
  readinessLevel: string;
  personnelStrength: number;
  equipmentReadinessPct: number;
  reportingOfficer: string;
  reportingDtg: string;
```

```
}

interface ValidationResult {
  valid: boolean;
  errors: string[];
}

// Pure validation function — no Foundry decorators needed
// Separate from the Action itself for testability
function validateSitrepInput(
  input: SitrepValidationInput,
  currentStatus: UnitStatus | null,
): ValidationResult {
  /**
   * Validate SITREP submission parameters against business rules.
   *
   * Rules enforced:
   * 1. C-rating string must be C1/C2/C3/C4
   * 2. Personnel strength must not exceed 110% of assigned
   * 3. Equipment readiness must be between 0.0 and 1.0
   * 4. Reporting DTG must not be more than 24h in the past
   * 5. Reporting officer must be non-empty
   * 6. Cannot submit a SITREP for a unit in DEACTIVATED status
   */
  const errors: string[] = [];

  // Rule 1: C-rating validation
  const validRatings = new Set(["C1", "C2", "C3", "C4"]);
  if (!validRatings.has(input.readinessLevel)) {
    errors.push(
      `Invalid readiness level '${input.readinessLevel}'. ` +
        `Must be C1, C2, C3, or C4.`,
    );
  }

  // Rule 2: Personnel strength ceiling
  const assignedStrength = currentStatus?.assignedStrength ?? 0;
  if (assignedStrength > 0 && input.personnelStrength > assignedStrength * 1.1) {
    errors.push(
      `Personnel strength ${input.personnelStrength} exceeds 110% of ` +
        `assigned strength ${assignedStrength}. Verify before submitting.`,
    );
  }

  // Rule 3: Equipment readiness bounds
  if (input.equipmentReadinessPct < 0 || input.equipmentReadinessPct > 1.0) {
    errors.push(
      `Equipment readiness ${input.equipmentReadinessPct} out of range [0.0, 1.0].`,
    );
  }

  // Rule 4: DTG recency check (no more than 24h stale)
  const reportDtg = new Date(input.reportingDtg);
  const now = new Date();
  const ageHours = (now.getTime() - reportDtg.getTime()) / (1000 * 60 * 60);
```

```
  if (ageHours > 24) {
    errors.push(
      `Reporting DTG is ${Math.round(ageHours)} hours old. ` +
        `SITREPs older than 24 hours require commander approval.`,
    );
  }

  // Rule 5: Reporting officer required
  if (!input.reportingOfficer || input.reportingOfficer.trim().length === 0) {
    errors.push("Reporting officer is required.");
  }

  // Rule 6: Unit status check
  if (currentStatus?.unitActivationStatus === "DEACTIVATED") {
    errors.push(
      `Unit is in DEACTIVATED status. SITREPs cannot be submitted for ` +
        `deactivated units. Contact G1 to update unit status.`,
    );
  }

  return { valid: errors.length === 0, errors };
}

// The Action function — registered with Foundry runtime
export class SitrepActionFunctions {
  @ActionEditFunction()
  submitSitrep(
    @ActionInput("unit") unit: UnitStatus,
    @ActionInput("readinessLevel") readinessLevel: string,
    @ActionInput("personnelStrength") personnelStrength: number,
    @ActionInput("equipmentReadinessPct") equipmentReadinessPct: number,
    @ActionInput("reportingOfficer") reportingOfficer: string,
    @ActionInput("reportingDtg") reportingDtg: string,
    @Edits(Sitrep) sitrep: Sitrep,
  ): void {
    const validation = validateSitrepInput(
      {
        unit,
        readinessLevel,
        personnelStrength,
        equipmentReadinessPct,
        reportingOfficer,
        reportingDtg,
      },
      unit, // Pass current unit status for cross-field validation
    );

    if (!validation.valid) {
      // Throwing in an ActionEditFunction prevents the action from applying
      throw new Error(
        `SITREP validation failed:\n${validation.errors.join("\n")}`,
      );
    }

    // Validation passed — apply edits to the Sitrep object
```

```
    sitrep.unit = unit;
    sitrep.readinessLevel = readinessLevel;
    sitrep.personnelStrength = personnelStrength;
    sitrep.equipmentReadinessPct = equipmentReadinessPct;
    sitrep.reportingOfficer = reportingOfficer;
    sitrep.reportingDtg = reportingDtg;
    sitrep.submittedAtUtc = new Date().toISOString();
    sitrep.status = "SUBMITTED";
  }
}
```

6-3. Multi-Step Action Flows

BLUF: Multi-step Actions with conditional paths (e.g., submit → route for review → approve/reject) require a state machine pattern in TypeScript. Design the state machine explicitly — do not encode state in property naming conventions.

CONDITIONS: All state transitions and business rules are documented before implementation. Action definitions for each transition step are configured in the Foundry Ontology. Object Types for the primary entity and any audit/review records are defined.

STANDARDS: State machine transitions are defined explicitly as a constant — no ad hoc transition logic embedded in conditional branches. Invalid transitions throw before any edits are applied. Audit record is created for every state change. Unit tests cover all valid transitions and all invalid transition attempts.

EQUIPMENT: TypeScript development environment; `@foundry/functions-api` and `@foundry/ontology-api` packages; state machine design document; Jest test framework.

PROCEDURE — Multi-step EXORD processing action (TypeScript):

```
import {
  ActionEditFunction,
  ActionInput,
  Edits,
} from "@foundry/functions-api";
import { Exord, ExordReview } from "@foundry/ontology-api";

// Explicit state machine — all valid states and transitions defined here
type ExordStatus =
  | "DRAFT"
  | "SUBMITTED"
  | "UNDER_REVIEW"
  | "APPROVED"
  | "REJECTED"
  | "PUBLISHED";

const VALID_TRANSITIONS: Record<ExordStatus, ExordStatus[]> = {
  DRAFT: ["SUBMITTED"],
  SUBMITTED: ["UNDER_REVIEW"],
  UNDER_REVIEW: ["APPROVED", "REJECTED"],
```

```
  APPROVED: ["PUBLISHED"],
  REJECTED: ["DRAFT"], // Allow re-draft after rejection
  PUBLISHED: [], // Terminal state — no further transitions
};

function assertValidTransition(
  current: ExordStatus,
  next: ExordStatus,
): void {
  const allowed = VALID_TRANSITIONS[current] ?? [];
  if (!allowed.includes(next)) {
    throw new Error(
      `Invalid EXORD status transition: ${current} -> ${next}. ` +
        `Allowed transitions from ${current}: [${allowed.join(", ")}]`,
    );
  }
}

export class ExordActionFunctions {
  @ActionEditFunction()
  advanceExordStatus(
    @ActionInput("exord") exord: Exord,
    @ActionInput("targetStatus") targetStatus: string,
    @ActionInput("reviewerComment") reviewerComment: string,
    @ActionInput("reviewerName") reviewerName: string,
    @Edits(Exord) exordEdit: Exord,
    @Edits(ExordReview) review: ExordReview,
  ): void {
    const currentStatus = (exord.status ?? "DRAFT") as ExordStatus;
    const nextStatus = targetStatus as ExordStatus;

    // Validate transition — throws if invalid
    assertValidTransition(currentStatus, nextStatus);

    // Business rule: APPROVED requires a reviewer comment
    if (nextStatus === "APPROVED" && !reviewerComment?.trim()) {
      throw new Error(
        "Reviewer comment is required when approving an EXORD.",
      );
    }

    // Business rule: REJECTED requires a reviewer comment
    if (nextStatus === "REJECTED" && !reviewerComment?.trim()) {
      throw new Error(
        "Reviewer comment is required when rejecting an EXORD.",
      );
    }

    // Apply state transition
    exordEdit.status = nextStatus;
    exordEdit.lastModifiedUtc = new Date().toISOString();

    // Create review record for audit trail
    review.exord = exord;
    review.fromStatus = currentStatus;
```

```
    review.toStatus = nextStatus;
    review.reviewerName = reviewerName;
    review.reviewerComment = reviewerComment ?? "";
    review.reviewTimestampUtc = new Date().toISOString();
  }
}
```

CHAPTER 7 — SLATE APPLICATIONS (LEGACY)

NOTE — Palantir Developers reference: *Foundry Reference Project | Apps* — Covers the application layer of the Foundry Reference Project, demonstrating current recommended patterns for Workshop-based and OSDK-backed apps — useful context when planning migration away from Slate to current-generation application frameworks. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Developer Deskside | Creating Map Based Workflows with Workshop in Palantir Foundry* — Shows how to build map-based operational workflows in Workshop, the current-generation replacement for map-heavy Slate applications. Relevant to anyone maintaining or migrating a map-based Slate app. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Developer Deskside | Building a Ticket Framework in Foundry* — Demonstrates building a ticket/workflow application in Foundry — useful reference for migrating existing Slate operational workflow applications to Workshop. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

CAUTION

Slate is a legacy application builder. Do not use Slate for new development. Use Workshop for internal Foundry applications. For public-facing portals, build an external application using the OSDK or Platform SDK.

7-1. What Is Slate

BLUF: Slate is a legacy Foundry-hosted environment for building custom HTML/CSS/JavaScript applications. This chapter is retained for maintenance of existing Slate applications only. All new application development must use Workshop (for internal Foundry users) or OSDK-backed external applications (for users without Foundry access or public-facing portals).

CAUTION

Slate is a legacy application builder. Do not use Slate for new development. Use Workshop for internal Foundry applications. For public-facing portals, build an external application using the OSDK or Platform SDK.

Slate applications (legacy characteristics): - Run inside the Foundry UI frame (same auth session as the user) - Can call Foundry APIs including the OSDK, dataset query endpoints, and ontology endpoints - Support full HTML5/CSS3/JavaScript including modern frameworks (React, Vue if bundled) - Are deployed and version-controlled as Foundry code resources - Inherit the user's CBAC permissions — they cannot access data the user cannot see

Application type selection — current guidance:

Requirement	Current Approved Tool	Notes
Internal application for Foundry users	Workshop	No-code/low-code; see SL 2 and SL 3
Public-facing portal or app for non-Foundry users	External application backed by OSDK or Platform SDK	Hosted on external infrastructure with proper authentication (OAuth2, service account)
Existing Slate app (maintenance only)	Slate	Do not extend Slate apps; plan migration to Workshop or OSDK external app

DEPRECATED — When to use Slate vs. external application (archived reference for existing apps):

Factor	Slate (LEGACY)	External Application (CURRENT)
Auth model	Inherits user's Foundry session — no separate auth	Requires separate credential management (service account or OAuth)
Deployment	Hosted within Foundry — no external infrastructure	Requires external hosting (on-prem server, container)
Access	Users must have Foundry access	Can serve users without Foundry accounts
CBAC	Enforced automatically by Foundry session	Must be implemented explicitly in application
Integration	Direct Foundry API access	OSDK / Platform SDK over HTTPS

7-2. Slate Application Structure

CAUTION

Slate is a legacy application builder. Do not use Slate for new development. Use Workshop for internal Foundry applications. For public-facing portals, build an external application using the OSDK or Platform SDK.

Standard Slate project structure (maintenance reference only):

```
my-slate-app/
├── index.html              # Entry point — rendered by Foundry Slate runtime
├── styles/
│   └── main.css
├── scripts/
│   ├── main.js             # Application entry
│   ├── api.js              # Foundry API abstraction layer
│   ├── components/
│   │   ├── readinessDashboard.js
│   │   └── sitrepForm.js
│   └── utils/
│       ├── dtgFormatter.js
│       └── ratingColors.js
├── assets/
│   └── usareur_seal.png
└── foundry.json
```

7-3. Foundry API Integration from Slate

CAUTION

Slate is a legacy application builder. Do not use Slate for new development. Use Workshop for internal Foundry applications. For public-facing portals, build an external application using the OSDK or Platform SDK.

CONDITIONS: Existing Slate application is in maintenance (not new development). Foundry Slate runtime is accessible. `window.foundryContext` is populated by the Slate runtime with base URL and auth token. Target Object Type is accessible to the authenticated user.

STANDARDS: Auth tokens are used only in JavaScript scope — never rendered in HTML or logged. All user inputs are sanitized before inclusion in API filter parameters. Error messages exposed to users do not contain raw API error details or schema information.

EQUIPMENT: Existing Slate application codebase; Foundry development environment; browser developer tools for local debugging.

PROCEDURE — Query Ontology objects from Slate JavaScript (maintenance reference only):

```
// api.js — Foundry API abstraction for Slate application

/**
 * Query UnitStatus objects filtered by AOR.
 * Uses the Foundry Slate context API for auth — no separate credentials needed.
 *
 * @param {string} aorFilter - AOR code string (e.g., "EUROPE-NORTH")
 * @returns {Promise<Array>} Array of unit status objects
 */
async function getUnitStatusByAor(aorFilter) {
  // Slate context provides the base URL and auth headers automatically
  const ontologyApiBase = `${window.foundryContext.baseUrl}/api/v1/ontologies`;
  const ontologyRid = window.foundryContext.ontologyRid;

  const response = await fetch(
    `${ontologyApiBase}/${ontologyRid}/objects/UnitStatus` +
      `?filter=${encodeURIComponent(
        JSON.stringify({
          type: "eq",
          field: "aorCode",
          value: aorFilter,
        }),
      )}&pageSize=100`,
    {
      method: "GET",
      headers: {
        Authorization: `Bearer ${window.foundryContext.authToken}`,
        "Content-Type": "application/json",
      },
    },
  );

  if (!response.ok) {
    throw new Error(
      `UnitStatus query failed: ${response.status} ${response.statusText}`,
    );
  }

  const data = await response.json();
  return data.data ?? [];
}

/**
 * Execute a SubmitSitrep Action from Slate.
 * Uses user's session token — Action enforces CBAC for the current user.
 *
 * @param {Object} params - SITREP parameters
 * @returns {Promise<void>}
 */
async function submitSitrepAction(params) {
  const ontologyApiBase = `${window.foundryContext.baseUrl}/api/v1/ontologies`;
  const ontologyRid = window.foundryContext.ontologyRid;
```

```
  const response = await fetch(
    `${ontologyApiBase}/${ontologyRid}/actions/SubmitSitrep/apply`,
    {
      method: "POST",
      headers: {
        Authorization: `Bearer ${window.foundryContext.authToken}`,
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        parameters: {
          unit: { objectTypeApiName: "UnitStatus", primaryKey: params.unitRid },
          readinessLevel: params.readinessLevel,
          personnelStrength: params.personnelStrength,
          equipmentReadinessPct: params.equipmentReadinessPct,
          reportingOfficer: params.reportingOfficer,
          reportingDtg: params.reportingDtg,
        },
      }),
    },
  );

  if (!response.ok) {
    const errorBody = await response.json().catch(() => ({}));
    throw new Error(
      `SubmitSitrep action failed: ${response.status} — ` +
        `${errorBody.errorCode ?? "Unknown error"}: ${errorBody.message ?? ""}`,
    );
  }
}

export { getUnitStatusByAor, submitSitrepAction };
```

7-4. Slate Security Model

CAUTION

Slate is a legacy application builder. Do not use Slate for new development. Use Workshop for internal Foundry applications. For public-facing portals, build an external application using the OSDK or Platform SDK.

Critical security requirements for Slate applications (maintenance reference only):

Requirement	Implementation	Notes
Never expose auth tokens in DOM	Use `window.foundryContext.authToken` only in JS scope	Do not render tokens in HTML elements or log them

Requirement	Implementation	Notes
Validate all user inputs	Sanitize before including in API requests	Prevent injection into filter parameters
Error messages must not leak schema	Catch API errors and display safe user-facing messages	Do not render raw API error responses
Content Security Policy	Set CSP headers in Slate app configuration	Prevents XSS; required for all Slate apps in production
No external fetch from Slate	All API calls target the same Foundry origin	External fetch is blocked by CSP; do not attempt to circumvent

PROCEDURE — Secure input sanitization for Slate filter params:

```
/**
 * Sanitize user-provided filter string for use in Foundry API filter parameter.
 * Prevents injection through encoded or special-character filter strings.
 *
 * @param {string} input - Raw user input
 * @param {number} maxLength - Maximum allowed length
 * @returns {string} Sanitized string safe for filter use
 */
function sanitizeFilterInput(input, maxLength = 50) {
  if (typeof input !== "string") {
    throw new TypeError("Filter input must be a string");
  }

  // Strip characters not allowed in filter values (alphanumeric, hyphen, underscore,
space)
  const sanitized = input.replace(/[^a-zA-Z0-9\-_ ]/g, "").trim();

  if (sanitized.length === 0) {
    throw new Error("Filter input is empty after sanitization");
  }

  if (sanitized.length > maxLength) {
    throw new Error(
      `Filter input too long: ${sanitized.length} chars (max ${maxLength})`,
    );
  }

  return sanitized;
}
```

CHAPTER 8 — CI/CD AND CODE REPOSITORY DISCIPLINE

NOTE — Palantir Developers reference: *Code Repositories* | *Development Process and Pipeline Craftsmanship in Palantir Foundry* — Covers the full development lifecycle for Foundry code resources — branching strategy, code review, pipeline hygiene, and promotion discipline. A comprehensive reference for the practices described in this chapter. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

8-1. Repository Structure for Foundry Code Resources

BLUF: Foundry code resources — TypeScript FOO functions, Action validators, Slate apps — are developed in Git repositories with the same engineering discipline as any production software system. No code deploys to production outside of the approved CI/CD workflow.

Standard repository layout for a Foundry code resource project:

```
my-foundry-project/
├── .github/
│   └── workflows/
│       ├── ci.yml              # Lint, type-check, unit test on PR
│       └── promote.yml         # Branch promotion to Foundry on merge
├── src/
│   ├── functions/              # TypeScript FOO and Action functions
│   ├── slate/                  # Slate application code
│   └── tests/                  # All tests co-located with source
├── scripts/
│   ├── deploy_dev.sh           # Deploy to Foundry dev branch
│   └── promote_prod.sh         # Promote Foundry branch to production (gated)
├── .env.example                # Template — never commit .env
├── .gitignore                  # Must include .env, node_modules, dist/
├── foundry.json                # Foundry project configuration
├── package.json
├── tsconfig.json
└── README.md
```

CAUTION: `.env` files containing tokens must be in `.gitignore` before the repository's first commit. Retroactive removal from Git history does not invalidate the exposed credential — rotate the token immediately if committed. Report to unit S6/G6.

8-2. Foundry Branch Workflow

NOTE — Palantir Developers reference: *Code Repositories* | *Best Practices for Creating Pull Requests in Palantir Foundry* — PR best practices in Foundry code repositories, directly reinforcing the C2DAO code review workflow and the PR standards described in this chapter. Available on the Palantir

Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Code Repositories | Reviewing Code and Best Practices* — Covers code review standards and best practices within Foundry repositories, complementing the peer review requirements in Section 8-5. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

Standard branch strategy for Foundry code resources:

```
main (production)
  |
  +-- develop (integration)
        |
        +-- feature/TASK-123-readiness-score-foo
        +-- feature/TASK-124-sitrep-validator
        +-- fix/TASK-125-pagination-bug
```

Branch protection rules (configure in GitHub/GitLab and Foundry):

Branch	Required Reviews	Required Checks	Push Direct?
`main` (production)	2 (tech lead + C2DAO rep)	All CI checks pass	No
`develop` (integration)	1 (peer review)	Lint + unit tests pass	No
`feature/*`	0 (WIP)	None (developer discretion)	Yes (developer)
`fix/*`	1 (peer review)	Lint + unit tests pass	No

Foundry-side branch workflow mirrors Git workflow:

```
Foundry: develop branch  <-->  Git: develop branch
Foundry: master branch   <-->  Git: main branch
```

All Foundry code resource changes go to the `develop` branch first. Promotion to `master` (production) requires the Git `main` branch merge AND a Foundry branch review/merge through the Foundry UI or Foundry CI integration.

8-3. CI/CD Pipeline Configuration

CONDITIONS: Git repository is configured with branch protection rules per Section 8-2. GitHub Actions (or equivalent CI system) is available for the repository. ESLint, TypeScript, and Jest are configured in `package.json`.

STANDARDS: CI runs on every pull request targeting `develop` or `main`. Lint, type-check, and unit test steps are all required to pass before merge is permitted. Secret scanning step is included to catch committed credentials. Coverage threshold is enforced (80% line minimum for unit tests).

EQUIPMENT: GitHub Actions; Node.js 20; `package.json` with `lint`, `type-check`, and `test` scripts configured; `.gitignore` with `.env` listed.

PROCEDURE — Configure GitHub Actions CI pipeline:

```
# .github/workflows/ci.yml
name: CI — Lint, Type Check, Test

on:
  pull_request:
    branches: [develop, main]
  push:
    branches: [develop]

jobs:
  ci:
    runs-on: ubuntu-latest
    timeout-minutes: 15

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: "20"
          cache: "npm"

      - name: Install dependencies
        run: npm ci

      - name: Lint
        run: npm run lint
        # ESLint with @typescript-eslint — no errors allowed on PR

      - name: Type check
        run: npm run type-check
        # tsc --noEmit — type errors block merge

      - name: Unit tests
        run: npm run test -- --coverage --coverageThreshold='{"global":{"lines":80}}'
        # 80% line coverage minimum — enforced as merge gate
        env:
          CI: true

      - name: Check for committed secrets
        run: |
          # Fail if .env files or token patterns appear in committed files
          if git diff HEAD~1 --name-only | xargs grep -lE \
            'FOUNDRY_TOKEN|_SECRET|_PASSWORD|Bearer [a-zA-Z0-9]{20,}' \
            2>/dev/null; then
            echo "ERROR: Potential credential committed. Review and rotate."
```

DRAFT

```
            exit 1
          fi
```

8-4. Automated Testing Requirements

NOTE — Palantir Developers reference: *Code Repositories* | *How to Use Data Expectations in Palantir Foundry* — Covers Foundry's built-in Data Expectations feature for defining data quality assertions within a pipeline — directly relevant to the deployment checklist and automated testing requirements for datasets and transforms. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

Testing requirements by code resource type:

Resource Type	Required Test Types	Coverage Minimum	Notes
FOO functions	Unit tests	80% line	Test all edge cases including null/missing properties
Action validators	Unit tests	90% line	Validators are security-sensitive — higher bar
Slate JS	Unit tests for business logic	70% line	DOM-dependent code tested via jsdom
Python pipelines	Unit + integration	80% line	Integration tests run against Foundry dev branch
OSDK clients	Unit tests with mocks	75% line	Mock OSDK client — do not test against production

PROCEDURE — Mock the OSDK client for unit tests (TypeScript):

```
// tests/mocks/osdkClientMock.ts
import { vi } from "vitest";

/**
 * Create a mock OSDK client for unit testing.
 * Prevents any actual calls to MSS during test runs.
 *
 * Usage:
 *   const mockClient = createMockOsdkClient({ unitStatusData: [...] });
 *   const result = await getUnitStatusRecords(mockClient, "W12345");
 */
export function createMockOsdkClient(fixtures: {
  unitStatusData?: Record<string, unknown>[];
}) {
  return {
    ontology: {
      objects: {
```

```
        UnitStatus: {
          filter: vi.fn().mockReturnValue({
            asyncIter: async function* () {
              for (const record of fixtures.unitStatusData ?? []) {
                yield record;
              }
            },
          }),
        },
      },
      actions: {
        SubmitSitrep: {
          apply: vi.fn().mockResolvedValue({ type: "SUCCESS" }),
        },
      },
    },
  };
}
```

8-5. Branch Promotion Workflow

CONDITIONS: All CI checks pass on the Git `develop` branch. Integration tests have been run against the Foundry dev environment. Peer review is complete. For Ontology or dataset changes, C2DAO data steward coordination is scheduled.

STANDARDS: All seven promotion steps are completed in sequence. No step is bypassed, including C2DAO review for schema/CBAC/marking changes. Post-promotion smoke test is executed within 30 minutes. Promotion is documented in the unit data ops log.

EQUIPMENT: Git repository access (`develop` and `main` branches); Foundry UI access for branch management; C2DAO ticketing system access; operations log.

PROCEDURE — Promote Foundry develop branch to master (production):

```
STEP 1: Verify CI passes on Git develop branch
  - All lint, type-check, and unit tests green
  - No open critical findings in code review

STEP 2: Conduct integration test against Foundry develop branch
  - Deploy code to Foundry develop branch (automated or manual)
  - Run integration test suite against Foundry dev environment
  - Verify all FOO functions and Action validators execute correctly
  - Spot-check at least three real Ontology objects

STEP 3: Peer review in Foundry code repository (UI)
  - Open branch review in Foundry repository viewer
  - Reviewer must be a different engineer than the author
  - Reviewer confirms: no hardcoded credentials, no debug logging of object data

STEP 4: Merge Git develop -> main (requires 2 approvals per branch rules)
```

```
STEP 5: C2DAO data steward review (for any changes affecting production datasets or
Ontology)
  - Submit Foundry branch merge request via C2DAO ticketing system
  - C2DAO confirms: CBAC settings unchanged, no marking changes, UDRA alignment intact

STEP 6: Foundry develop -> master branch promotion (Foundry UI)
  - Performed only after Steps 1-5 complete
  - Promotes all code resources, FOO functions, and Action validators atomically
  - Monitor for 30 minutes post-promotion: check application health, query latency

STEP 7: Post-promotion verification
  - Execute smoke test suite against production
  - Confirm readiness dashboard loads and displays correct data
  - Confirm SubmitSitrep action executes end-to-end
  - Document promotion in unit data ops log
```

WARNING: Never bypass the C2DAO review step (Step 5) for promotions that modify Ontology schemas, CBAC assignments, or data markings. These changes affect every user and downstream system. Unauthorized schema changes in production are a reportable incident.

NOTE

CWIX is NATO's annual interoperability validation event (~3,000 participants, 40 nations, ~25,000 tests). Foundry pipelines supporting coalition operations should target CWIX validation profiles for their domain.

NOTE — When to Use Compute Modules Compute Modules provide containerized compute for workloads that exceed Code Repositories and Functions capabilities: - **GPU workloads**: satellite imagery, large model inference, embedding computation - **Custom runtimes**: languages/frameworks not natively supported in Foundry - **Spiky demand**: auto-scaling for unpredictable computational loads - **Existing code**: 60% of Compute Modules run code authored outside Foundry

Decision guide: Use **Functions** for lightweight ontology operations. Use **Code Repositories** for Python/Java transforms. Use **Compute Modules** for containerized, GPU, or external code deployment.

Source: Palantir Developer Community — Why We Built It: Compute Modules

CHAPTER 9 — SECURITY AND COMPLIANCE

9-1. CBAC in External Applications

BLUF: Context-Based Access Control (CBAC) in Foundry ensures users only see data they are authorized to access. External applications consuming MSS via OSDK must preserve this access control — they must not aggregate, cache, or expose data beyond the authorization of the requesting user.

Core CBAC rules for external application developers:

Rule	Requirement	Violation Example
No elevation	Service account cannot grant end users access beyond their Foundry permissions	App queries OSDK as service account and returns full object set to any web user
No caching beyond session	Cached OSDK responses expire with user session	Caching sensitive operational data in shared Redis without TTL
No aggregation bypass	Aggregate only over objects the user can see	Computing fleet-wide readiness from objects filtered to user's authorization
Audit log preservation	Application-level audit log must capture who queried what	Missing logging on sensitive object access
Marking propagation	If source object is marked CUI, display must indicate marking	Stripping CUI markings in report output

9-2. Credential Management

NOTE — Palantir Developers reference: *Platform Administration* | *Setting up SSO in Palantir Foundry* — Covers SSO configuration for the Foundry platform, relevant to senior SWEs and administrators managing authentication architecture for MSS external applications. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

CONDITIONS: External application requires authenticated access to MSS OSDK or Platform SDK. C2DAO has provisioned the appropriate credential type. AR 25-2 requirements are understood and being applied.

STANDARDS: All credentials are stored in approved storage patterns (preference order listed below). No credentials appear in source code, Dockerfile, docker-compose, log files, or version control history. Token rotation schedule is documented and executed.

EQUIPMENT: C2DAO-provisioned credentials; approved secrets manager (CI/CD secrets vault or Army-approved HashiCorp Vault); `.env` file (local dev only, in `.gitignore`); unit security log.

PROCEDURE — Credential management for deployed external applications:

```
APPROVED credential storage patterns (in order of preference):

1. Foundry OAuth2 Confidential Client (server-side token exchange)
   - Best for web applications with a server component
   - Token stays server-side; never exposed to browser
   - C2DAO provisions client_id and client_secret

2. Environment variable injection (CI/CD secrets)
```

```
   - Service account token stored in CI/CD secrets vault
   - Injected at deploy time as environment variable
   - Never appears in source code or container image

3. Army-approved secrets manager
   - HashiCorp Vault or equivalent approved by G6
   - Application reads secret at startup from secrets API
   - Requires AR 25-2 compliant implementation

NOT APPROVED:

- Hardcoded tokens in source code
- Tokens in Dockerfile or docker-compose.yml
- Tokens in configuration files committed to version control
- Tokens in log files or debug output
- Tokens shared over NIPR email
```

PROCEDURE — Token rotation discipline:

```
Token rotation schedule (align with AR 25-2 and local G6 guidance):

- Service account tokens: rotate every 90 days minimum
- PATs (developer use only): rotate every 30 days
- OAuth2 client secrets: rotate every 180 days minimum

Rotation procedure:
1. Provision new credential through C2DAO process
2. Update secrets manager / CI/CD secrets vault with new value
3. Redeploy application to pick up new credential
4. Verify application functions correctly with new credential
5. Revoke old credential in Foundry token management
6. Document rotation in unit security log
```

NOTE — Palantir Developers reference: *Cipher* | *How to Encrypt Data in Foundry with Cipher* — Covers Foundry's Cipher tool for field-level encryption of sensitive data, directly relevant to data classification and security requirements for objects carrying CUI or higher markings. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Security* | *How to Debug a User's Access to a File* — Diagnostic procedure for investigating access control issues in Foundry — essential for developers troubleshooting CBAC problems in external OSDK applications. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

9-3. Marking Compliance in OSDK Queries

BLUF: Foundry data markings (CUI, coalition restrictions) are enforced at the OSDK layer. However, your application is responsible for propagating these markings in display and output.

CONDITIONS: External application retrieves OSDK objects that may carry data markings (CUI, coalition restrictions). Application presents data to end users in a UI or report output.

STANDARDS: Markings are extracted from every OSDK object response. Highest marking in a result set is determined and displayed as a page/section banner. No marking is stripped before display. CUI data displayed without markings constitutes a spillage event.

EQUIPMENT: Authenticated OSDK Python client; UI or report output layer; knowledge of applicable marking categories (CUI, REL TO USA/NATO).

PROCEDURE — Check and propagate markings (Python):

```
from dataclasses import dataclass, field
from enum import Enum

class DataMarking(Enum):
    UNCLASSIFIED = "U"
    CUI = "CUI"
    CUI_REL_NATO = "CUI//REL TO USA, NATO"

@dataclass
class MarkedObject:
    """Wrapper for OSDK object with marking metadata preserved."""
    rid: str
    properties: dict
    markings: list[str] = field(default_factory=list)
    highest_marking: DataMarking = DataMarking.UNCLASSIFIED

def extract_object_with_markings(osdk_object) -> MarkedObject:
    """
    Extract object properties and markings from an OSDK object.
    Markings must be preserved — do not strip or ignore them.

    Applications displaying this data must show the appropriate
    marking banner based on highest_marking.
    """
    markings = [
        m.get("displayName", "")
        for m in (osdk_object.markings or [])
    ]

    # Determine highest marking level for display banner
    highest = DataMarking.UNCLASSIFIED
    if any("REL TO" in m or "NATO" in m for m in markings):
        highest = DataMarking.CUI_REL_NATO
    elif any("CUI" in m for m in markings):
        highest = DataMarking.CUI

    return MarkedObject(
        rid=osdk_object.rid,
        properties={
            k: v for k, v in osdk_object.properties.items()
        },
        markings=markings,
```

```
        highest_marking=highest,
    )

def format_display_banner(marking: DataMarking) -> str:
    """Return the correct classification banner for UI display."""
    banners = {
        DataMarking.UNCLASSIFIED: "UNCLASSIFIED",
        DataMarking.CUI: "CONTROLLED UNCLASSIFIED INFORMATION",
        DataMarking.CUI_REL_NATO:
            "CONTROLLED UNCLASSIFIED INFORMATION // REL TO USA, NATO",
    }
    return banners[marking]
```

WARNING: Applications that strip or do not display data markings are in violation of Army data handling policy (Army CIO Data Stewardship Policy, April 2024) and AR 25-2. CUI data displayed without markings constitutes a spillage event. Report to unit S2/S6 immediately.

9-4. Audit Trail Requirements

CONDITIONS: External application executes OSDK queries or Action calls against sensitive Object Types. C2DAO has identified which Object Types require audit logging. Approved log destination is configured (SIEM, approved log aggregator, or local audit log file for dev).

STANDARDS: Every query against a sensitive Object Type is logged with user_id, object_type, filter parameters, and result count. Every Action execution is logged with user_id, action name, target RID, and success/failure. Object property values are NOT logged in audit records. Audit log is separate from application log.

EQUIPMENT: Python `logging` module; approved log destination (file or SIEM); C2DAO list of Object Types requiring audit logging.

PROCEDURE — Application-level audit logging (Python):

```
import logging
import json
from datetime import datetime, timezone
from typing import Any

# Configure audit logger — write to separate audit log file, not application log
audit_logger = logging.getLogger("audit")
audit_logger.setLevel(logging.INFO)

# Audit log handler — write to approved log destination
# In production: configure to write to SIEM or approved log aggregator
audit_handler = logging.FileHandler("/var/log/mss_app/audit.log")
audit_handler.setFormatter(
    logging.Formatter("%(asctime)s %(message)s")
)
audit_logger.addHandler(audit_handler)
```

```
def audit_log_query(
    user_id: str,
    object_type: str,
    filter_params: dict[str, Any],
    result_count: int,
    application_name: str,
) -> None:
    """
    Log an OSDK query to the audit trail.

    Required for all queries against sensitive Object Types.
    Consult C2DAO for the list of Object Types requiring audit logging.

    DO NOT log object property values in the audit record —
    log only the query parameters and result count.
    """
    record = {
        "event": "OSDK_QUERY",
        "timestamp_utc": datetime.now(timezone.utc).isoformat(),
        "user_id": user_id,
        "application": application_name,
        "object_type": object_type,
        "filter_params": filter_params,   # Query params only — no property values
        "result_count": result_count,
    }
    audit_logger.info(json.dumps(record))


def audit_log_action(
    user_id: str,
    action_name: str,
    target_object_rid: str,
    success: bool,
    error_message: str | None,
    application_name: str,
) -> None:
    """
    Log an Action execution to the audit trail.
    Required for all Action executions in external applications.
    """
    record = {
        "event": "OSDK_ACTION",
        "timestamp_utc": datetime.now(timezone.utc).isoformat(),
        "user_id": user_id,
        "application": application_name,
        "action_name": action_name,
        "target_rid": target_object_rid,
        "success": success,
        "error_message": error_message,
    }
    audit_logger.info(json.dumps(record))
```

9-5. Integration Security — REST APIs and Webhooks

CONDITIONS: External application exposes an inbound webhook endpoint or calls outbound REST APIs. HMAC shared secret is provisioned through C2DAO-approved credential store. TLS certificates are in place on all endpoints.

STANDARDS: All inbound webhook payloads are signature-verified before processing. Timestamp check enforces maximum request age (default 5 minutes) to prevent replay attacks. All outbound HTTP calls enforce connect and read timeouts. Rate limiting is implemented. Egress is restricted to approved MSS endpoints.

EQUIPMENT: Inbound webhook endpoint with HTTPS; HMAC shared secret in environment variable; Python `hmac` and `hashlib` modules; approved firewall egress configuration from G6.

Security requirements for REST/webhook integrations:

Requirement	Standard	Notes
TLS 1.2+ required	All HTTP communication over TLS 1.2 minimum	TLS 1.3 preferred for new integrations
Webhook signature verification	Verify HMAC signature on all inbound webhooks	Reject requests with invalid or missing signatures
Input validation at boundary	Validate all inbound data against expected schema	Do not trust inbound JSON field types
Rate limiting	Implement rate limiting on integration endpoints	Prevents abuse; coordinate with C2DAO for approved limits
Timeout enforcement	Set connect and read timeouts on all outbound calls	Prevent hung connections from blocking application threads
Allowlist egress	Restrict outbound connections to approved MSS endpoints	Implemented at firewall level; verify with G6

PROCEDURE — Webhook signature verification (Python):

```
import hmac
import hashlib
import time

def verify_webhook_signature(
    payload_bytes: bytes,
    signature_header: str,
    secret: str,
    timestamp_header: str,
    max_age_seconds: int = 300,
) -> bool:
    """
    Verify HMAC-SHA256 signature on an inbound webhook payload.
```

```
    Rejects requests where:
    - Signature does not match (tampered payload or wrong secret)
    - Timestamp is older than max_age_seconds (replay attack prevention)

    Args:
        payload_bytes:    Raw request body bytes
        signature_header: X-Foundry-Signature header value
        secret:           Shared HMAC secret (from environment variable)
        timestamp_header: X-Foundry-Timestamp header value
        max_age_seconds:  Max acceptable age of request (default 5 min)

    Returns:
        True if valid; False if invalid (do not process payload if False)
    """
    # Replay attack check — reject stale requests
    try:
        request_timestamp = int(timestamp_header)
    except (ValueError, TypeError):
        return False

    age = int(time.time()) - request_timestamp
    if age > max_age_seconds or age < 0:
        return False

    # Compute expected signature: HMAC-SHA256(timestamp + "." + body)
    message = f"{timestamp_header}.".encode() + payload_bytes
    expected_signature = hmac.new(
        secret.encode(),
        message,
        hashlib.sha256,
    ).hexdigest()

    # Constant-time comparison — prevents timing attacks
    return hmac.compare_digest(
        f"sha256={expected_signature}",
        signature_header,
    )
```

CHAPTER 10 — MODEL CONTEXT PROTOCOL (MCP) INTEGRATION

NOTE — Palantir Developers reference: *Code in Production: Lennar x MCP* | *DevCon 3* — Production case study of MCP integration with Foundry. Demonstrates connecting an LLM agent to the Ontology using MCP for tool-use workflows. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

10-1. What Is MCP

BLUF: The Model Context Protocol (MCP) is an open standard for connecting AI agents to external tools and data sources. Foundry released MCP as an application-level feature in January 2026. MCP enables AI agents to query the Ontology, execute Actions, and retrieve object data through a standardized protocol — without custom OSDK integration code for each agent.

MCP defines a client-server architecture: the AI agent (client) discovers available tools from an MCP server, then invokes those tools during reasoning. In the Foundry context, the MCP server exposes Ontology Object Types, Actions, and Functions as tools that the agent can call. The agent does not access the Ontology directly — all access flows through the MCP server, which enforces CBAC and audit logging.

Why MCP matters for SL 4L engineers:

Without MCP	With MCP
Each AI agent requires custom OSDK integration code	Agent discovers Ontology capabilities through standard protocol
Agent tool definitions are hardcoded and brittle to Ontology changes	Tool definitions are dynamically generated from current Ontology schema
Adding a new Object Type to an agent requires code change + redeploy	Adding a new Object Type to the MCP server makes it immediately available to all connected agents
Each agent implementation duplicates CBAC enforcement, pagination, error handling	MCP server handles CBAC, pagination, and error handling centrally

MCP is NOT a replacement for OSDK. OSDK remains the approved SDK for building external applications that present data to human users (dashboards, forms, portals). MCP is specifically for enabling AI agents to interact with the Ontology programmatically during autonomous or semi-autonomous reasoning workflows. If the consumer is a human user, use OSDK. If the consumer is an AI agent, evaluate MCP.

Scope boundary: MCP server configuration and Ontology exposure are application-level concerns owned by the SL 4L engineer. The AI agent logic itself (prompt engineering, reasoning chains, model selection) is owned by the SL 4H (AI Engineer). Coordinate across tracks when deploying MCP-enabled workflows.

10-2. MCP Architecture in Foundry

CONDITIONS: Foundry environment has MCP enabled (application-level feature, January 2026). Engineer has developer permissions on the target Ontology project. MCP server is provisioned through the Foundry Developer Console.

STANDARDS: MCP servers expose only the Object Types and Actions required by the consuming agent — no blanket Ontology exposure. CBAC is enforced on every MCP tool invocation. All MCP tool calls are audit-logged.

Architecture:

```
AI Agent (LLM)
     |
     | MCP Protocol (JSON-RPC over stdio or HTTP+SSE)
     v
MCP Server (Foundry-hosted)
     |
     | OSDK (internal)
     v
MSS Ontology
  - Object Types (query, get)
  - Actions (execute)
  - Functions (invoke)
     |
     v
CBAC Enforcement + Audit Log
```

MCP server components:

Component	Description	SWE Responsibility
Tool definitions	Schema describing each Ontology capability (Object Type queries, Actions, Functions) the agent can invoke	Configure which Object Types, Actions, and Functions are exposed; define parameter schemas
Transport layer	Communication channel between agent and MCP server (stdio for local; HTTP+SSE for remote)	Select transport based on deployment model; configure endpoint security
Authentication	Identity under which MCP tool calls execute against the Ontology	Configure service account or user-delegated auth; follows same patterns as OSDK (see 2-2)
CBAC enforcement	Access control on every tool invocation	Inherits from Foundry CBAC — no additional configuration, but verify enrollment scope covers exposed tools
Audit logging	Record of every tool call: who, what, when, parameters, result summary	Automatic for Foundry-hosted MCP servers; verify logs are accessible in Health Dialog (see 3-5)

Transport options:

Transport	Use Case	Security Considerations
stdio (standard I/O)	Local development; agent and MCP server run on same machine	No network exposure; suitable for development and testing
HTTP + Server-Sent Events (SSE)	Production deployment; agent and MCP server on different hosts	Requires TLS 1.2+; authenticate agent-to-server connection; restrict to approved network paths

10-3. Configuring an MCP Server for Ontology Access

CONDITIONS: MCP feature enabled in Foundry. Target Object Types and Actions are defined in the Ontology. OSDK application enrollment exists for the MCP server (or a new enrollment will be created). Service account or OAuth2 credentials are provisioned.

STANDARDS: MCP server configuration follows least-privilege: expose only the Object Types, Actions, and Functions the consuming agent requires. Configuration is version-controlled alongside application code. Changes to exposed tools require C2DAO review when they affect production Ontology access.

EQUIPMENT: Foundry Developer Console access; OSDK application enrollment; service account token (for server-to-server) or OAuth2 configuration (for user-delegated).

PROCEDURE — Configure a Foundry MCP server:

1. **Create or select an OSDK application enrollment.** The MCP server uses an enrollment to determine which Object Types, Actions, and Functions it can expose. Use an existing enrollment or create a new one scoped to the agent's requirements.
2. **Define the MCP server configuration.** The configuration specifies which Ontology capabilities are exposed as MCP tools:

```
{
  "mcpServer": {
    "name": "mss-readiness-agent-tools",
    "description": "MCP tools for the V Corps readiness monitoring agent",
    "transport": "http_sse",
    "authentication": {
      "type": "service_account",
      "tokenEnvVar": "MCP_SERVICE_ACCOUNT_TOKEN"
    },
    "tools": {
      "objectTypes": [
        {
          "apiName": "Unit",
          "operations": ["query", "get"],
          "description": "Query and retrieve unit objects with readiness status"
        },
        {
```

```
          "apiName": "Equipment",
          "operations": ["query", "get"],
          "description": "Query equipment objects by unit and status"
        }
      ],
      "actions": [
        {
          "apiName": "FlagReadinessAnomaly",
          "description": "Flag a unit readiness anomaly for staff review"
        }
      ],
      "functions": [
        {
          "apiName": "computeUnitReadinessScore",
          "description": "Compute aggregate readiness score for a unit"
        }
      ]
    }
  }
}
```

1. **Provision credentials.** For server-to-server deployment (most common for autonomous agents), use a service account token stored in the environment variable specified in the configuration. For user-delegated scenarios (agent acting on behalf of a logged-in user), configure OAuth2 per section 2-2.
2. **Deploy the MCP server.** For Foundry-hosted MCP servers, deploy through the Developer Console. For self-hosted MCP servers (e.g., running alongside an external agent), deploy the MCP server application to your infrastructure and configure the agent to connect to it.
3. **Validate tool discovery.** Connect a test agent (or use the MCP inspector tool) and verify the agent can discover the exposed tools. Confirm tool schemas match the current Ontology definitions.
4. **Validate CBAC enforcement.** Attempt a tool call for an Object Type the service account does not have access to. Confirm the MCP server returns an authorization error, not data.

10-4. MCP Use Cases in the USAREUR-AF Context

Operational use cases where MCP provides value over direct OSDK integration:

Use Case	Agent Description	MCP Tools Exposed	Track Coordination
Readiness monitoring	Autonomous agent monitors unit readiness, flags anomalies	Unit (query), Equipment (query), FlagReadinessAnomaly (action)	SL 4H (agent logic), SL 4L (MCP config)
SITREP analysis	Agent reads submitted SITREPs, extracts trends, generates summary	Sitrep (query), Unit (query), computeTrendScore (function)	SL 4H (analysis logic), SL 4L (MCP config), SL 4A (Intel validation)

Use Case	Agent Description	MCP Tools Exposed	Track Coordination
Data quality audit	Agent scans Object Types for data quality issues (missing fields, stale records)	Multiple Object Types (query), FlagDataQualityIssue (action)	SL 4H (audit logic), SL 4L (MCP config), SL 4K (KM oversight)
Maintenance prediction	Agent analyzes equipment maintenance history, predicts upcoming failures	Equipment (query), MaintenanceRecord (query), computeFailureProbability (function)	SL 4H (model logic), SL 4M (ML model), SL 4L (MCP config)

PROCEDURE — Connecting an AI agent to MCP (Python example using the MCP Python SDK):

```
from mcp import ClientSession, StdioServerParameters
from mcp.client.stdio import stdio_client

async def connect_to_foundry_mcp() -> None:
    """
    Connect an AI agent to the Foundry MCP server and
    discover available Ontology tools.

    This is the agent-side connection pattern. The agent
    discovers tools, then invokes them during reasoning.
    """
    # Configure connection to the MCP server
    server_params = StdioServerParameters(
        command="foundry-mcp-server",
        args=["--config", "mcp_config.json"],
        env={
            "MCP_SERVICE_ACCOUNT_TOKEN": os.environ["MCP_SERVICE_ACCOUNT_TOKEN"],
        },
    )

    async with stdio_client(server_params) as (read, write):
        async with ClientSession(read, write) as session:
            # Initialize the MCP session
            await session.initialize()

            # Discover available tools — these map to Ontology capabilities
            tools = await session.list_tools()
            for tool in tools:
                print(f"Tool: {tool.name} — {tool.description}")

            # Example: invoke a query tool
            result = await session.call_tool(
                "query_Unit",
                arguments={
                    "filter": {
                        "property": "readinessLevel",
                        "type": "lt",
                        "value": "C3",
                    },
                    "pageSize": 50,
```

```
        },
    )

    # Process results — agent uses these in reasoning
    for unit in result.content:
        print(f"Unit: {unit}")
```

NOTE

The MCP Python SDK (`mcp` package) is the reference client implementation. For TypeScript agents, use the `@modelcontextprotocol/sdk` package. Both implement the same protocol specification.

WARNING

MCP gives AI agents the ability to execute Actions — state-modifying operations on the Ontology. Scope Action exposure carefully. An agent with access to a delete or modify Action can alter operational data autonomously. Use read-only tool exposure for monitoring agents; add write Actions only for agents with explicit human-in-the-loop approval gates.

CAUTION

MCP tool calls consume the same OSDK rate limits as direct OSDK queries. An agent making rapid, unbounded tool calls can exhaust rate limits and degrade performance for all MSS users. Implement agent-side rate limiting or configure the MCP server with call frequency bounds.

APPENDIX A — OSDK QUICK REFERENCE

A-1. Python OSDK Common Patterns

Operation	Pattern	Notes
List objects (filtered)	`client.ontology.objects["Type"].filter({...}).take(N)`	Always set explicit limit
Get single object by key	`client.ontology.objects["Type"].get(primary_key)`	Raises if not found
Execute action	`client.ontology.actions["ActionName"].apply(**params)`	Validate params before calling

Operation	Pattern	Notes
Traverse link	`obj.links["LinkName"].get_linked_objects()`	Paginate if link has many children
Paginated list	`page = ...list(pageSize=N); page.next_page_token`	Loop until next_page_token is None
Subscribe to changes	OSDK subscriptions — use TypeScript SDK for subscriptions	Python subscription support varies by SDK version

A-2. TypeScript OSDK Common Patterns

Operation	Pattern	Notes
Iterate all matching objects	`for await (const obj of client(Type).where(...).asyncIter())`	Handles pagination automatically
Fetch single object	`await client(Type).fetchOne(primaryKey)`	Throws if not found
Execute action	`await client(Action).applyAction({...})`	Return value varies by action type
Subscribe	`client(Type).where(...).subscribe({onChange, onError})`	Returns unsubscribe function
Compound filter	`.where({ $and: [Type.prop.eq(v1), Type.prop2.lt(v2)] })`	$and and $or supported

A-3. OSDK Error Codes

Error Code	Meaning	Action
400 Bad Request	Invalid filter, bad parameter format	Check query parameters and object type schema
401 Unauthorized	Token expired or invalid	Rotate token; check FOUNDRY_TOKEN env var
403 Forbidden	No CBAC access to object type	Contact data steward; verify enrollment scope
404 Not Found	Object type or object does not exist	Verify object type API name; check enrollment
429 Too Many Requests	Rate limit exceeded	Implement exponential backoff; reduce query frequency
503 Service Unavailable	MSS platform degraded	Retry with backoff; check MSS status page

APPENDIX B — SWE SECURITY CHECKLIST

Complete this checklist before any code promotion to production. Document completion in the unit data ops log.

B-1. Credential Hygiene

- ☐ No credentials, tokens, or passwords in any file committed to version control
- ☐ `.env` and any credential files are in `.gitignore`
- ☐ Service account credentials loaded from environment variables or approved secrets manager
- ☐ PATs are not used in deployed applications (developer use only)
- ☐ Token rotation schedule is documented and scheduled

B-2. CBAC and Authorization

- ☐ External application does not elevate user permissions beyond their Foundry CBAC
- ☐ Service account queries are scoped to the minimum required Object Types
- ☐ OSDK application enrollment includes only Object Types and Actions needed (no overage)
- ☐ Application has been reviewed by C2DAO for CBAC compliance before go-live

B-3. Data Handling

- ☐ OSDK query results are not cached beyond the user's session scope
- ☐ Data markings (CUI, coalition) are displayed in application UI
- ☐ Application audit log captures all queries and action executions against sensitive objects
- ☐ No operational data in application debug logs or error messages exposed to end users

B-4. Input Validation

- ☐ All user-supplied inputs are validated and sanitized before use in API queries
- ☐ Inbound webhook payloads are signature-verified before processing
- ☐ Application enforces timeout on all outbound HTTP calls

B-5. CI/CD and Code Quality

- ☐ All unit tests pass in CI pipeline

DRAFT

- ☐ Code coverage meets minimums (validators ≥90%, FOO ≥80%, other ≥75%)
- ☐ No outstanding lint or type-check errors
- ☐ Peer review completed by engineer other than author
- ☐ C2DAO data steward review completed (for Ontology or dataset changes)

B-6. Post-Deployment

- ☐ Smoke test executed against production after promotion
- ☐ No new errors in application logs within 30 minutes of deployment
- ☐ Readiness dashboards and SITREP functions verified end-to-end
- ☐ Rollback plan documented and communicated to team

APPENDIX C — INTEGRATION PATTERNS REFERENCE

C-1. Pattern: SITREP Automation Service

Use case: Automated SITREP generation — reads Unit objects from MSS Ontology, formats reports, pushes to EUCOM J3 portal.

Architecture:

```
EUCOM J3 Portal <-- SITREP Service --> MSS (OSDK)
                        |
                   Scheduler
                   (cron / task queue)
```

Key implementation decisions: - Auth: Service account token (server-to-server, no user session needed) - Trigger: Scheduled task every 6 hours OR subscription-triggered on UnitStatus change - Output: Formatted report (PDF or JSON) pushed to external system REST API - Error handling: Dead letter queue for failed submissions; alert to operations team - CBAC: Service account scoped to read UnitStatus objects only; write to external system only

C-2. Pattern: Readiness Dashboard (External)

Use case: V Corps readiness dashboard served to users who do not access Foundry directly.

Architecture:

```
Browser --> React Frontend --> Node.js BFF --> MSS (OSDK via OAuth2)
                                    |
```

```
                                    Session Store
                                    (Redis / memory)
```

Key implementation decisions: - Auth: OAuth2 PKCE (user-delegated) — each user authenticates with their own Foundry identity - CBAC: Enforced automatically — user sees only objects their CBAC allows - Caching: OSDK responses cached per user session only; TTL 5 minutes; invalidated on logout - Markings: Highest marking of any object in the response displayed in page header banner - Audit logging: Every query logged with user_id, object_type, and result count

C-3. Pattern: External System Feed (Inbound to MSS)

Use case: GCSS-Army equipment status feed → MSS Equipment Ontology (daily sync).

Architecture:

```
GCSS-Army --> Feed Processor --> Foundry Platform SDK (dataset write)
                  |                          |
             Transform + validate      Staging dataset
                                            |
                                  Pipeline Builder transform
                                            |
                                  Equipment Ontology objects
```

Key implementation decisions: - Write target: Staging dataset (via Platform SDK APPEND transaction) — never write directly to Ontology. CAUTION: APPEND is not inherently idempotent; use surrogate keys to deduplicate before appending. - Transform: Pipeline Builder transform (designed by -30 builder) reads staging → curated → Ontology - Idempotency: Each record has a source system ID; INSERT OR IGNORE pattern prevents duplicates - Error handling: Records failing validation written to error dataset; operations team alerted - Schema validation: Inbound feed validated against expected schema before any write

C-4. Pattern: Webhook Integration (MSS Outbound)

Use case: MSS triggers webhook to external alerting system when unit drops below C3 readiness.

Architecture:

```
MSS Ontology (Action) --> Webhook Endpoint --> External Alert System
         |                        |
   Action validator         Signature verify
                            Rate limit check
                            Payload validate
```

Key implementation decisions: - Trigger: SubmitSitrep Action includes webhook call when readiness < C3 - Security: HMAC-SHA256 signature on every outbound webhook; verified by receiver - Retry: Exponential backoff (3 attempts); failures logged and alerted to operations - Payload: Minimal payload —

object RID and event type only; receiver queries OSDK for full data - Rate limiting: Maximum one webhook per unit per hour to prevent alert flooding

APPENDIX D — PROFESSIONAL READING LIST

Curated articles from Army professional journals and military publications. These supplement doctrinal references with contemporary operational perspectives.

Source	Title	Date	Relevance
Army AL&T	"The Software Advantage"	2024-25	Software modernization
Army AL&T	"Commoditizing AI/ML Models"	2024-25	Platform engineering for AI/ML
Field Artillery Bulletin	"The New Digital Kill Chain"	2025	Software-defined fires
Army Communicator	"Leading in Data Centricity"	Spring 2025	Platform data centricity

GLOSSARY

Action (Foundry) — A defined operation that modifies Ontology object state. Actions enforce validation, authorization, and audit logging. External applications execute Actions via OSDK. Not to be confused with a direct dataset write.

Action Validator — TypeScript function deployed as a Foundry code resource that enforces business rules before an Action applies state changes. Written by -40L engineers; configured by -30 builders.

APPEND Transaction — A dataset write transaction type that adds rows to an existing dataset without modifying existing data. The standard write pattern for incremental data loads. APPEND transactions are NOT inherently idempotent — each call appends data without deduplication. Implement deduplication logic (content hashes or surrogate keys) before appending if idempotency is required. Use SNAPSHOT transactions for full-dataset atomic replacement.

AOR (Area of Responsibility) — Geographic and functional boundary within which a command exercises authority. Relevant to OSDK query filtering — most operational queries scope to a specific AOR.

AR 25-2 — Army Regulation 25-2, Cybersecurity. Governs credential handling, system authorization, access control, and security incident reporting.

DRAFT

Audit Trail — Application-level log capturing who queried what data and when. Required for all queries against sensitive Object Types and all Action executions in external applications.

BFF (Backend for Frontend) — Server-side component of a web application that handles authentication and API calls on behalf of the browser frontend. The correct pattern for web apps consuming MSS via OSDK — keeps tokens server-side.

Branch (Foundry) — An isolated development environment within a Foundry project. Code changes are developed on a non-production branch and promoted to master (production) after review.

CBAC (Context-Based Access Control) — Foundry's access control model. Governs which users can read, write, and execute Actions on specific Object Types and datasets based on roles and markings. External applications must preserve CBAC — they cannot elevate user permissions.

C2DAO (Command and Control Data Architecture Office) — USAREUR-AF theater-level authority for MSS data architecture, OSDK enrollment, API access governance, and production promotion approvals.

CI/CD (Continuous Integration / Continuous Deployment) — Automated pipeline that runs tests, lint, and type checks on every commit and gates deployments on passing checks. Required for all Foundry code resources.

CUI (Controlled Unclassified Information) — An information handling category requiring specific access and marking controls. CUI markings on MSS objects must be displayed in any application that presents that data.

DTG (Date-Time Group) — Military date-time format: DDHHMMZMMMYYYY (e.g., 101435ZMAR2026). Used in SITREP timestamps and operational records throughout MSS.

EUCOM (United States European Command) — Geographic Combatant Command to which USAREUR-AF is the ASCC (along with USAFRICOM). MSS integrations with EUCOM systems are a primary SL 4L use case.

EXORD (Execute Order) — A command directive executing a plan. Referenced in multi-step Action workflows that manage EXORD state machine transitions.

FMC (Fully Mission Capable) — Equipment status indicating the item is capable of performing all assigned missions. A key metric in equipment readiness queries.

FOO (Functions on Objects) — TypeScript functions deployed within Foundry that compute derived properties or aggregations against Ontology objects at query time. Server-side, stateless, sandboxed.

Foundry Platform SDK — Python SDK providing programmatic access to Foundry datasets, branches, transactions, and file resources. Distinct from the OSDK (which accesses the Ontology).

GCSS-Army (Global Combat Support System — Army) — Army logistics management system. A common source for equipment and supply data ingested into MSS.

G2 — Intelligence staff section (corps/division/brigade level). A primary consumer of ISR and all-source analysis data products on MSS.

G6 — Signal/communications staff section. Responsible for Army network infrastructure and cybersecurity implementation. Relevant to credential provisioning and integration approvals.

Health Dialog (Foundry) — Platform-native interface that displays OSDK application errors, including authentication failures, authorization failures, schema mismatches, and rate-limit violations. First-line debugging tool for OSDK applications (Q1 2026). See section 3-5.

HMAC (Hash-based Message Authentication Code) — Cryptographic signature mechanism used to verify webhook payload integrity and authenticity.

ISR (Intelligence, Surveillance, Reconnaissance) — The collection, processing, and dissemination of information. ISR tracking applications are a common SL 4L deliverable.

Link Type (Foundry Ontology) — A defined relationship between two Object Types. Traversing link types from external applications requires additional OSDK calls — use bulk patterns to avoid N+1 performance issues.

Marking — A metadata tag on a Foundry object indicating its handling category (e.g., CUI, coalition restriction). Enforced by CBAC; must be propagated in application display.

MCP (Model Context Protocol) — Open standard for connecting AI agents to external tools and data sources. Released as a Foundry application-level feature in January 2026. Enables AI agents to query the Ontology, execute Actions, and invoke Functions through a standardized protocol without custom OSDK integration code. See Chapter 10.

MPE (Mission Partner Environment) — Network environment enabling data sharing between U.S. forces and coalition partners. Objects accessible on MPE require NAFv4 compliance review before SL 4L integration development.

MSS (Maven Smart System) — The USAREUR-AF enterprise AI/data platform built on Palantir Foundry. The platform against which all SL 4L development occurs.

N+1 Query Problem — A performance anti-pattern where an application executes one query to retrieve a list of objects, then one additional query per object to retrieve related data. Produces O(N) API calls. Solved with batch query patterns.

NAFv4 (NATO Architecture Framework version 4) — Coalition data architecture standard. Required for any integration accessible by coalition partners on MPE.

NMC (Non-Mission Capable) — Equipment status indicating the item cannot perform any assigned mission. Drives maintenance prioritization queries.

Object Type (Foundry Ontology) — A typed entity in the Foundry Ontology (e.g., Unit, Equipment, Sitrep). The primary unit of data access via the OSDK.

OSDK (Ontology SDK) — The approved programmatic interface for external applications to query Ontology objects and execute Actions. Available in Python and TypeScript. Enforces CBAC and markings on every query.

OAuth2 PKCE — OAuth 2.0 authorization flow using Proof Key for Code Exchange. Used for user-delegated authentication in single-page applications where secrets cannot be stored safely on the client.

PAT (Personal Access Token) — Developer credential for Foundry access. Authenticates as the issuing user. Approved for local development only — never in deployed applications.

Pilot (Foundry) — AI-powered tool for generating React OSDK application scaffolding, components, and integration code (launched March 2026). Accelerates initial development but does not replace engineering review. All Pilot-generated code must meet the same CI/CD and security standards as manually written code. See section 3-7.

Pipeline Builder — Foundry's visual ETL tool for building data transformation workflows. SL 3 scope for design; Platform SDK provides programmatic access to datasets produced by pipelines.

PMC (Partially Mission Capable) — Equipment status indicating the item can perform some but not all assigned missions.

Promotion (Foundry branch) — The process of merging a non-production branch (develop) to the production branch (master) after completing all review and testing gates. Requires C2DAO coordination for changes affecting production Ontology or datasets.

Service Account — A non-human identity provisioned for deployed applications. Service account tokens are the approved credential type for server-side external applications. Provisioned by C2DAO.

SITREP (Situation Report) — A formatted report summarizing the current status of a unit or operation. SITREP automation (reading from MSS, formatting, pushing to external systems) is a primary SL 4L use case.

Slate — A legacy Foundry environment for building custom HTML/CSS/JavaScript applications hosted within the Foundry platform. Applications inherit the user's Foundry session and CBAC. **Slate is deprecated — do not use for new development.** Use Workshop for internal Foundry applications. For public-facing portals, build an external application using the OSDK or Platform SDK.

Snapshot Transaction — A dataset write transaction type that replaces all existing data in a dataset atomically. Used for full-refresh pipelines. Requires data steward coordination before use on shared datasets.

Temporary Media Upload — OSDK and Foundry Functions capability (Q1 2026) enabling user-submitted files (images, documents, attachments) to be uploaded as part of Action execution or Function invocation without requiring a permanent dataset file store. Temporary uploads are retained for a platform-defined window and must be persisted to an object property or dataset by the processing logic. See section 3-6.

TLS (Transport Layer Security) — Cryptographic protocol securing HTTPS connections. TLS 1.2 minimum required for all MSS integrations; TLS 1.3 preferred for new integrations.

Transaction (Foundry Platform SDK) — An atomic unit of work for writing to a Foundry dataset. All writes must be wrapped in a transaction. Transactions are committed (visible) or aborted (rolled back) — there is no partial commit.

TypeScript — A typed superset of JavaScript. The required language for FOO functions, Action validators, and Slate application logic. SL 4L prerequisite skill.

UDRA v1.1 — Unified Data Reference Architecture, version 1.1 (February 2025). Defines domain ownership, federated governance, and integration standards for MSS. All SL 4L integrations must align.

USAREUR-AF — United States Army Europe and Africa. The Army Service Component Command to USEUCOM and USAFRICOM, headquartered in Wiesbaden, Germany. The operational context for all SL 4L development.

III Corps — Third Corps, recently realigned under USAREUR-AF. Major consumer of MSS readiness data products.

V Corps — Fifth Corps, the primary warfighting corps HQ in Europe. Major consumer of MSS readiness data products.

VAUTI — Visible, Accessible, Understandable, Trustable, Interoperable. Legacy 5-dimension data quality framework from AR 25-1 (2019). Superseded by VAULTIS (DoD Data Strategy, 2020) and extended to VAULTIS-A (DDOF Playbook v2.2, December 2025). See VAULTIS-A.

VAULTIS-A — Visible, Accessible, Understandable, Linked, Trusted, Interoperable, Secure, Auditable. The current 8-dimension data quality framework per DDOF Playbook v2.2 (December 2025). Supersedes VAUTI (AR 25-1) and VAULTIS (DoD Data Strategy 2020). All MSS data products and integrations must achieve an 85% minimum weighted average across all 8 dimensions (DDOF Phase 3 quality gate). Proponent: T2COM C2DAO / HQDA CIO/G-6 / SAIS-ADD.

Webhook — An HTTP callback that allows a system to push event notifications to an external receiver. Used in MSS integration patterns to trigger downstream processing on Ontology state changes.

SL 4L, Maven Smart System (MSS), Software Engineer Technical Manual Headquarters, United States Army Europe and Africa, Wiesbaden, Germany, 2026 Distribution authorized to U.S. Government agencies and their contractors only.

DoD and Army Strategic References:

- **DoDI 5000.87, Software Acquisition Pathway (October 2020)** — Establishes the software acquisition pathway for rapid, iterative software delivery
- **DoD Software Modernization Strategy (February 2022)** — DoD-wide strategy for software modernization, DevSecOps, and platform engineering
- **Army Directive 2024-02, Agile Software Development (December 2024)** — Army policy for agile software development practices and delivery