TECHNICAL MANUAL

SL 5L

TM-50L — ADVANCED SOFTWARE ENGINEERING

Specialist Course Manual

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

26 MARCH 2026

DRAFT — UNOFFICIAL — NOT FOR OPERATIONAL USE

TM-50L — ADVANCED SOFTWARE ENGINEERING

Forward: SL 5L qualifies senior software engineers to lead MSS application development capability — designing platform architecture, enforcing security at scale, optimizing performance, and governing the technical practices of the USAREUR-AF SWE community. **Prereqs:** SL 4L (Software Engineer) — required. SL 4H (AI Engineer) and SL 4M (ML Engineer) — recommended for integration track engineers. Senior-level Python and TypeScript proficiency required; CONCEPTS_GUIDE_TM50L_SOFTWARE_ENGINEER_ADVANCED (read before this manual). *HQ USAREUR-AF · v1.0 · 2026 · DISTRIB: USG only · AUTH: C2DAO/UDRA v1.1*

WARNING

Infrastructure-level errors at SL 5L can corrupt shared platform state, disable access controls for multiple units, and produce data integrity failures that cascade through the entire USAREUR-AF data ecosystem. Engineering discipline at this level is not optional — it is a mission requirement. **CAUTION:** Platform SDK credentials with dataset write access are high-privilege operational secrets. Exposure constitutes a security incident. Report immediately to unit S6/G6 and C2DAO. Rotate immediately. **NOTE:** SL 5L tasks are not solo work. Always operate with a second qualified engineer as reviewer. For Ontology branch merges and CBAC policy changes, also require C2DAO sign-off.

CHAPTER 1 — INTRODUCTION AND SCOPE

1-1. Advanced Software Engineer Manual

BLUF: SL 5L qualifies senior software engineers to lead MSS application development capability — designing platform architecture, enforcing security at scale, optimizing performance, and governing the technical practices of the USAREUR-AF SWE community.

This manual builds directly on SL 4L. It does not repeat SL 4L content. If a concept was covered in SL 4L, this manual references it and extends it. Engineers who have not completed SL 4L will lack the foundational context required for SL 5L tasks.

SL 5L scope:

Domain	SL 4L Baseline	SL 5L Extension
Platform SDK	Dataset read/write, transaction management	Infrastructure integrations at scale, branch automation, bulk operations
Performance	Writing functional queries	Query profiling, caching architecture, indexing strategy, compute cost management
Security	CBAC in external apps, credential management	Multi-tenant isolation patterns, security assessment, OWASP in Foundry context
External integrations	REST APIs, webhooks	gRPC patterns, event streaming (Kafka/Kinesis), high-throughput ingestion architecture
DevSecOps	CI/CD for code repos	Full DevSecOps pipeline: Ontology CI, automated security scanning, ATO support
Leadership	Individual contributor	Architecture review, code standards, developer onboarding, platform governance

1-2. The SL 5L Role in USAREUR-AF

USAREUR-AF spans III Corps, V Corps, 21st TSC, 7th ATC, 10th AAMDC, 56th MDC-E, SETAF-AF, G2/G3/G6 staff, and coalition partners across the European and African AOR. MSS supports this entire enterprise. The SL 5L engineer is responsible not for a single application but for the health and capability of the entire SWE layer.

SL 5L responsibilities in the data chain:

```
MISSION REQUIREMENT
       |
       v
  C2DAO / G6                <- Architecture governance
  (Platform policy,
   CBAC schema,
   tenant boundaries)
       |
       v
  SL 5L SENIOR SWE          <- You are here
  (Platform architecture,
   reusable components,
   performance governance,
   security standards,
   SWE community leadership)
       |
       v
  SL 4L SWE TEAM            <- Your developers
  (OSDK apps, FOO, Workshop/OSDK-external,
   integrations, CI/CD)
       |
       v
```

```
SL 3 / SL 4G/H/M        <- Builders and analysts
(Ontology design,
 pipelines, AI/ML)
      |
      v
OPERATIONAL PRODUCT
(Commanders, staff, EUCOM)
```

SL 5L engineers are force multipliers. One senior engineer enabling ten SL 4L developers to build correctly produces ten times the correct output. One senior engineer who builds solo produces one application. Lead.

1-2A. Relationship to Other SL 5 Publications

Publication	Track	Key Overlap with SL 5L
SL 5G	ORSA Advanced	Platform infrastructure supporting analytical pipelines
SL 5H	AI Engineer Advanced	OSDK integration with AI systems (Chapters 2, 7)
SL 5M	ML Engineer Advanced	ML model-serving integrations; feature pipeline infrastructure
SL 5J	Program Manager Advanced	Platform engineering program coordination; SWE team structure
SL 5K	Knowledge Manager Advanced	Platform SDK patterns for KM system backends
SL 5L	Software Engineer Advanced	THIS DOCUMENT
SL 5N	UI/UX Designer Advanced	Frontend design collaboration; design system implementation
SL 5O	Platform Engineer Advanced	Platform/application boundary; DevOps collaboration; deployment targets

WFF Operational Consumer Note. The software infrastructure built by SL 5L engineers is the delivery layer for capabilities consumed by all six Warfighting Function (WFF) tracks: Intelligence (SL 4A), Fires (SL 4B), Movement and Maneuver (SL 4C), Sustainment (SL 4D), Protection (SL 4E), and Mission Command (SL 4F). WFF practitioners access MSS data through Workshop applications, OSDK dashboards, and API integrations that SL 5L engineers architect and govern. Platform reliability, security, and performance are not abstract engineering concerns — they directly determine whether WFF staff sections can access the operational picture they need at the moment they need it.

1-3. Prerequisites

Prerequisite	Verification Method
SL 4L (Software Engineer)	Demonstrated proficiency: OSDK Python + TypeScript, Platform SDK, FOO, Workshop/OSDK-external apps, CI/CD, CBAC in apps

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. | SL 4H (AI Engineer) | Recommended — required for engineers in the integration/AI pipeline track | | SL 4M (ML Engineer) | Recommended — required for engineers supporting ML model serving integrations | | Senior Python | Async concurrency, profiling, memory management, packaging, type-checked codebases | | Senior TypeScript | Advanced generics, module federation, build optimization, strict mode | | Distributed systems | Event-driven architecture, eventual consistency, back-pressure, idempotency | | Security fundamentals | OWASP Top 10, secrets management, threat modeling, AR 25-2 familiarity |

1-4. Governing References

Document	Relevance
USAREUR-AF C2DAO Guidance	Theater architecture authority; multi-tenant policy; CBAC schema ownership
AR 25-2	Army Cybersecurity — ATO requirements, credential handling, incident reporting
DoD RMF (NIST SP 800-37)	Risk Management Framework — ATO process, security controls, continuous monitoring
NATO Architecture Framework v4 (NAFv4)	Coalition integration architecture — applies to any MPE-exposed objects
OWASP Application Security Verification Standard (ASVS)	Application security requirements — adapted for Foundry context in Chapter 7
DoD STIG for Application Security	Security Technical Implementation Guide — baseline for MSS application hardening
Army DIR 2024-03	Digital Engineering Policy — Army-wide digital engineering adoption directive
FM 3-12	Cyberspace Operations and Electromagnetic Warfare — cyberspace operations doctrine

Document	Relevance
DA PAM 25-2-5	Software Assurance — software security and assurance requirements
NATO ADatP-34 / NISP	C3 interoperability compliance — applies to coalition-facing platform integrations
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

1-4a. 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 chain, API governance, accountability for data products
UDRA v1.1 (February 2025)	Army Enterprise	Unified Data Reference Architecture — domain ownership, integration standards, canonical data flows
NATO Digital Transformation Implementation Strategy (Oct 2024)	NATO	NATO digital transformation roadmap — alignment target for enterprise SWE architecture

1-5. How to Use This Manual

Chapters 2–5 are technical reference and task chapters. Each section follows the TM format: Conditions, Standards, Equipment, Procedure. Read and understand the full chapter before executing tasks.

Chapter 6 (DevSecOps) and Chapter 7 (Security Assessment) require coordination with C2DAO before execution. Do not begin security assessment activities without written authorization.

Chapter 8 (Platform Leadership) is reference and guidance — not task-based. It describes how SL 5L engineers operate in the leadership role.

Appendices A and B are operational references used during code review and ATO support activities.

NOTE

Cross-references to SL 4H (AI Engineer) and SL 4M (ML Engineer) appear throughout. These are coordination points, not redundant content. When building systems that span multiple technical domains, engage the relevant SL 4-series engineers — do not attempt to own all domains as a single SWE.

CHAPTER 2 — PLATFORM SDK: INFRASTRUCTURE-LEVEL INTEGRATION

2-1. Scope of This Chapter

BLUF: The Platform SDK at SL 5L level goes beyond dataset read/write. This chapter covers infrastructure-level operations: bulk dataset management at scale, branch automation, transaction patterns for high-throughput ingestion, dataset lineage management, and the operational patterns required to manage a production MSS data environment programmatically.

NOTE — Palantir Developers reference: *Deep Dive: Code-Based AI Development with Ontology* — Covers code-based development patterns that extend traditional OSDK and Platform SDK usage into AI-integrated workflows, relevant to senior SWEs designing infrastructure that supports AI engineer and ML engineer deliverables. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

SL 4L covered Platform SDK fundamentals: authenticating, reading datasets, writing transactions, accessing file resources. This chapter extends to infrastructure-level use cases that arise when managing the platform at scale.

NOTE

Platform SDK operations at this level can affect shared infrastructure. Bulk deletes, branch merges, and large transaction writes should always be tested in a development or staging environment before execution against production. Coordinate all production operations with C2DAO.

2-2. Platform SDK Authentication at Scale

CONDITIONS: You are designing a system requiring multiple services, multiple service accounts, and potentially multiple Foundry enrollments connecting to MSS.

STANDARDS: Each service authenticates with a dedicated, least-privilege service account. No shared credentials between services. Credential rotation is automated or procedurally enforced. All credentials stored in approved secrets management infrastructure.

EQUIPMENT: Foundry Platform SDK (Python); approved secrets manager (HashiCorp Vault, AWS Secrets Manager, or equivalent approved by C2DAO); environment-based configuration.

Multi-service credential architecture:

```
MSS PRODUCTION
     |
     +-- Service A (ingestion)     <- Service account A: write to raw datasets only
     |
     +-- Service B (transformation) <- Service account B: read raw, write processed
     |
     +-- Service C (API serving)   <- Service account C: read processed, no write
     |
     +-- Service D (audit/export)  <- Service account D: read audit logs, no write
```

Each service account is provisioned with the minimum CBAC permissions required for its function. A compromise of Service C (read-only API) cannot be used to write data or access raw datasets.

PROCEDURE — Centralized credential management for multi-service Platform SDK deployments:

1. Define a configuration schema that separates credential references from credential values:

```
# config.py — configuration schema, safe to commit
from pydantic_settings import BaseSettings
from pydantic import Field

class FoundryConfig(BaseSettings):
    """
    MSS Platform SDK configuration.
    All secrets loaded from environment — never hardcoded.
    """
    foundry_url: str = Field(..., env="FOUNDRY_URL")
    service_account_token: str = Field(..., env="FOUNDRY_SA_TOKEN")
    dataset_rid_raw: str = Field(..., env="DATASET_RID_RAW")
    dataset_rid_processed: str = Field(..., env="DATASET_RID_PROCESSED")
    branch_name: str = Field(default="master", env="FOUNDRY_BRANCH")

    class Config:
        env_file = ".env"
        env_file_encoding = "utf-8"
```

1. Initialize a shared client factory to avoid redundant authentication across service components:

```
# client_factory.py
from functools import lru_cache
from foundry_sdk import FoundryClient
from .config import FoundryConfig

@lru_cache(maxsize=1)
def get_client() -> FoundryClient:
    """
    Return a cached Platform SDK client instance.
    lru_cache ensures single client per process — avoids repeated auth overhead.
```

```
    Thread-safe for read-heavy workloads; use separate instances for parallel writers.
    """
    cfg = FoundryConfig()
    return FoundryClient(
        hostname=cfg.foundry_url,
        auth=cfg.service_account_token,
    )
```

1. For parallel writers, use a client pool rather than a single shared instance:

```
# client_pool.py
import threading
from queue import Queue
from foundry_sdk import FoundryClient
from .config import FoundryConfig

class FoundryClientPool:
    """
    Thread-safe pool of Platform SDK clients for parallel write operations.
    Each writer thread checks out a dedicated client to avoid contention.
    """

    def __init__(self, pool_size: int = 4):
        self._pool: Queue[FoundryClient] = Queue(maxsize=pool_size)
        cfg = FoundryConfig()
        for _ in range(pool_size):
            self._pool.put(
                FoundryClient(
                    hostname=cfg.foundry_url,
                    auth=cfg.service_account_token,
                )
            )

    def checkout(self) -> FoundryClient:
        """Block until a client is available."""
        return self._pool.get(block=True, timeout=30)

    def checkin(self, client: FoundryClient) -> None:
        """Return client to pool after use."""
        self._pool.put(client)
```

CAUTION

Do not share a single Platform SDK client across threads performing concurrent writes. Foundry transaction state is not thread-safe within a single client instance. Use the pool pattern for parallel writers.

2-3. Dataset Operations at Scale

CONDITIONS: You need to perform bulk dataset operations: large-scale reads, high-throughput writes, dataset schema migrations, or bulk deletes in a production MSS environment.

STANDARDS: All bulk operations are idempotent. Progress is checkpointed. Operations fail safely — partial failure does not corrupt dataset state. Bulk operations do not degrade platform performance for other users.

EQUIPMENT: Platform SDK (Python); Apache Spark or Pandas depending on data volume; checkpoint storage (approved local or dataset-backed).

PROCEDURE: 1. Select the appropriate write pattern for the operation type (see 2-3a through 2-3c below). 2. Implement checkpointing for all long-running bulk operations. 3. Validate all data batches before write — never write unvalidated data to MSS. 4. For schema migrations, follow the full migration procedure in 2-3b. 5. Register dataset lineage metadata after any programmatic transform (see 2-3c). 6. Coordinate all production bulk operations with C2DAO before execution.

2-3a. High-Throughput Write Pattern

The default Platform SDK transaction pattern (open, write, commit) is appropriate for small-to-medium writes. For high-throughput ingestion, optimize the transaction batch size and use append mode to avoid full dataset rewrites.

```
import time
from typing import Iterator
import pandas as pd
from foundry_sdk import FoundryClient

def bulk_write_with_checkpointing(
    client: FoundryClient,
    dataset_rid: str,
    branch: str,
    data_iterator: Iterator[pd.DataFrame],
    checkpoint_interval: int = 10,
) -> dict:
    """
    Write large datasets in batches with progress checkpointing.

    Args:
        client: Authenticated Platform SDK client
        dataset_rid: Target dataset RID
        branch: Target branch (never write directly to master without C2DAO approval)
        data_iterator: Iterator yielding DataFrame chunks
        checkpoint_interval: Commit every N batches (tune for throughput vs.
durability)

    Returns:
        Summary dict with batch count, row count, elapsed time

    NOTE: This function appends to an existing transaction. The caller is responsible
    for opening the transaction and handling the final commit or abort.
    """
    batch_count = 0
    total_rows = 0
    start_time = time.monotonic()
```

```
    with client.datasets.transactions.start(
        dataset_rid=dataset_rid,
        branch=branch,
        transaction_type="APPEND",
    ) as txn:
        for batch_df in data_iterator:
            # Validate batch before write — never write unvalidated data
            if batch_df.empty:
                continue

            txn.write_pandas(batch_df)
            batch_count += 1
            total_rows += len(batch_df)

            # Periodic checkpoint log — helps diagnose failures in long-running jobs
            if batch_count % checkpoint_interval == 0:
                elapsed = time.monotonic() - start_time
                print(
                    f"[CHECKPOINT] Batches: {batch_count}, "
                    f"Rows: {total_rows}, "
                    f"Elapsed: {elapsed:.1f}s"
                )

    elapsed_total = time.monotonic() - start_time
    return {
        "batches": batch_count,
        "rows": total_rows,
        "elapsed_seconds": elapsed_total,
        "rows_per_second": total_rows / elapsed_total if elapsed_total > 0 else 0,
    }
```

NOTE

APPEND transactions are not inherently idempotent. Re-runs will write duplicate rows unless you implement deduplication (content hash + INSERT OR IGNORE, or surrogate key). Use SNAPSHOT for atomic full-dataset replacement.

2-3b. Schema Migration Pattern

Dataset schema changes in production require a coordinated migration pattern. Never alter schema in place on a dataset with active consumers.

```
SCHEMA MIGRATION PROCEDURE:

1. Create new dataset version (new RID or new branch)
2. Write migrated schema + backfilled data to new version
3. Validate new version against all downstream consumers (automated test suite)
4. Update all consumer configurations to point to new dataset RID/branch
5. Verify consumers are healthy on new version
6. Archive (do not delete) old dataset version — retain per data retention policy
7. Update dataset lineage metadata in Foundry
8. Notify C2DAO and downstream data stewards of completed migration
```

WARNING: Deleting a production dataset without confirming all consumers have migrated breaks downstream pipelines silently. Always archive, never delete, until all consumers are verified on the new version. Confirm with C2DAO before archiving.

2-3c. Dataset Lineage Management

SL 5L engineers are responsible for maintaining accurate lineage metadata for datasets they manage. Lineage is not optional — it is required for ATO continuous monitoring and for debugging data quality incidents.

```
def register_dataset_lineage(
    client: FoundryClient,
    output_dataset_rid: str,
    input_dataset_rids: list[str],
    transform_description: str,
) -> None:
    """
    Register lineage metadata for a dataset produced by a programmatic transform.
    Required for all datasets entering or passing through MSS production environment.

    This is a metadata operation — it does not modify the dataset itself.
    Coordinate with C2DAO for datasets that cross domain boundaries.
    """
    client.datasets.update_metadata(
        dataset_rid=output_dataset_rid,
        metadata={
            "lineage": {
                "inputs": input_dataset_rids,
                "transform": transform_description,
                "registered_at": time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime()),
            }
        },
    )
```

2-4. Branch Management Automation

CONDITIONS: You are managing a multi-environment MSS deployment (development, staging, production) or a multi-tenant environment requiring automated branch lifecycle management.

STANDARDS: Branch creation, promotion, and deletion are automated, audited, and gated. No manual branch promotions to production. All branch operations logged. Branch naming convention enforced programmatically.

EQUIPMENT: Platform SDK (Python); CI/CD pipeline (see Chapter 6); audit log storage.

PROCEDURE: 1. Define branch naming conventions using `BRANCH_PATTERNS` and enforce programmatically. 2. Create a `BranchPromotion` record for every promotion event, including ticket ID. 3. For promotions to master, verify ticket ID carries C2DAO prefix before proceeding. 4. Write audit log entry before executing the branch merge (fail-open: log first). 5. Execute branch merge via Platform SDK `branches.merge()`. 6. Verify downstream consumers are healthy after branch promotion.

Branch lifecycle in multi-environment MSS:

```
DEVELOPMENT BRANCH
  (per-developer or per-feature)
        |
        v [automated tests pass]
STAGING BRANCH
  (integration testing, QA)
        |
        v [C2DAO review + approval]
MASTER / PRODUCTION BRANCH
  (live data, operational consumers)
```

```
from dataclasses import dataclass
from datetime import datetime
from foundry_sdk import FoundryClient

# Branch naming convention — enforced programmatically
BRANCH_PATTERNS = {
    "feature": "feature/{ticket_id}/{description}",
    "staging": "staging/{date}",
    "hotfix": "hotfix/{ticket_id}/{description}",
}

@dataclass
class BranchPromotion:
    source_branch: str
    target_branch: str
    promoted_by: str  # Service account or user ID
    ticket_id: str    # Required — must reference approved change request
    promoted_at: datetime = None

    def __post_init__(self):
        self.promoted_at = datetime.utcnow()


def promote_branch(
    client: FoundryClient,
    dataset_rid: str,
    promotion: BranchPromotion,
    audit_log_path: str,
) -> None:
    """
    Promote a branch to a target (e.g., staging -> master).

    Enforces:
    - Ticket ID required (change management compliance)
    - Audit log written before promotion (fail-open: log first, then promote)
    - Never promotes directly to master without explicit override

    COORDINATION: Promotion to master requires C2DAO approval documented in ticket_id.
    """
    if promotion.target_branch == "master" and not
promotion.ticket_id.startswith("C2DAO-"):
```

```
        raise ValueError(
            "Production promotion to master requires a C2DAO- prefixed ticket ID. "
            f"Got: {promotion.ticket_id}. Obtain C2DAO approval before retrying."
        )

    # Write audit log before executing — if promotion fails, log still exists
    _write_audit_log(audit_log_path, promotion)

    # Execute branch merge via Platform SDK
    client.datasets.branches.merge(
        dataset_rid=dataset_rid,
        source_branch=promotion.source_branch,
        target_branch=promotion.target_branch,
    )


def _write_audit_log(path: str, promotion: BranchPromotion) -> None:
    """Append branch promotion event to audit log."""
    import json
    entry = {
        "event": "branch_promotion",
        "source": promotion.source_branch,
        "target": promotion.target_branch,
        "promoted_by": promotion.promoted_by,
        "ticket_id": promotion.ticket_id,
        "timestamp": promotion.promoted_at.isoformat(),
    }
    with open(path, "a") as f:
        f.write(json.dumps(entry) + "\n")
```

NOTE

Branch automation does not replace C2DAO governance. The automation enforces policy (ticket requirement, audit logging) but the human decision to approve production promotion remains with C2DAO. Do not design systems that remove the human from production promotion decisions.

2-5. File Resource Management at Scale

CONDITIONS: MSS datasets include large file resources (imagery, documents, binary attachments). You need to manage bulk file ingestion, retrieval, and lifecycle at scale.

STANDARDS: File operations are streamed — never load large files fully into memory. File integrity is verified on ingest (hash check). File metadata is registered in dataset schema alongside binary RIDs.

EQUIPMENT: Platform SDK (Python); SHA-256 hash utility (Python `hashlib`); approved local staging directory for pre-ingest validation.

PROCEDURE: 1. Obtain the expected SHA-256 hash from the sending system or manifest before ingestion. 2. Compute the actual hash of the local file using streaming reads (do not load full file into memory). 3. Compare actual hash to expected hash. If mismatch, do not ingest — flag for security review.

4. Stream-upload the file to the Foundry dataset using the Platform SDK file upload API. 5. Register file RID and integrity metadata (hash, file name, ingest timestamp) in the dataset schema. 6. Log the ingest event with hash values for audit trail.

```
import hashlib
import io
from pathlib import Path
from foundry_sdk import FoundryClient

def ingest_file_with_integrity_check(
    client: FoundryClient,
    dataset_rid: str,
    branch: str,
    local_path: Path,
    expected_sha256: str | None = None,
) -> str:
    """
    Ingest a file resource into a Foundry dataset with SHA-256 integrity verification.

    Args:
        client: Authenticated Platform SDK client
        dataset_rid: Target dataset RID
        branch: Target branch
        local_path: Path to local file
        expected_sha256: If provided, verify hash before ingestion. Required for
                         files received from external systems.

    Returns:
        Foundry file RID

    SECURITY NOTE: Always verify hash for files received from external Army systems.
    An attacker who can modify a file in transit can inject malicious content into
MSS.
    """
    # Compute hash before upload
    sha256 = hashlib.sha256()
    with open(local_path, "rb") as f:
        for chunk in iter(lambda: f.read(65536), b""):
            sha256.update(chunk)
    actual_hash = sha256.hexdigest()

    if expected_sha256 and actual_hash != expected_sha256:
        raise ValueError(
            f"File integrity check FAILED for {local_path.name}. "
            f"Expected: {expected_sha256}, Got: {actual_hash}. "
            "File may have been modified in transit. Do not ingest."
        )

    # Stream upload — do not load full file into memory
    with open(local_path, "rb") as file_stream:
        file_rid = client.datasets.files.upload(
            dataset_rid=dataset_rid,
            branch=branch,
            file_path=local_path.name,
```

```
            file_stream=file_stream,
        )

    return file_rid
```

CHAPTER 3 — HIGH-PERFORMANCE FOUNDRY DEVELOPMENT

3-1. Scope of This Chapter

BLUF: Performance is a mission-critical requirement in MSS. Commanders cannot make decisions on stale or slow data. This chapter covers query optimization, caching strategies, indexing patterns, and compute cost management for Foundry applications at scale.

NOTE — Palantir Developers reference: *Deep Dive: Interoperability at Scale with the Multimodal Data Plane | DevCon 5* — Covers cross-platform data interoperability at enterprise scale, including patterns for moving data between heterogeneous systems — directly relevant to senior SWEs architecting multi-system MSS integrations and performance-optimized data flows. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

The USAREUR-AF MSS instance serves hundreds of concurrent users across multiple commands and coalition elements. A poorly optimized query or a memory-inefficient transform does not just slow one user — it degrades the platform for the entire theater.

NOTE

Performance optimization work against production MSS must be coordinated with C2DAO. Profile in development/staging first. Document findings. Propose changes. Do not apply optimization changes directly to production without review.

3-2. Query Optimization for OSDK

CONDITIONS: An OSDK-based application or service is exhibiting slow query performance. You need to diagnose the root cause and implement optimized query patterns.

STANDARDS: OSDK queries against production MSS return results within defined SLA thresholds. Query patterns do not perform full object type scans when filtered queries are possible. Pagination is implemented for all result sets that may exceed 1,000 objects.

EQUIPMENT: OSDK (Python or TypeScript); profiling tools (cProfile, timing instrumentation); development or staging MSS environment.

PROCEDURE: 1. Identify the slow query or code path using profiling (see 3-6). 2. Check for N+1 query patterns — restructure to bulk link loading if found (see 3-2a). 3. Verify filters are pushed to the OSDK query layer, not applied in Python post-fetch (see 3-2b). 4. Apply property projection to fetch only required properties (see 3-2c). 5. Implement pagination for all result sets that may exceed 1,000 objects (see 3-2d). 6. Re-profile after changes to confirm improvement before promoting to production.

3-2a. The N+1 Query Problem in OSDK

The N+1 query problem is the most common performance anti-pattern in OSDK applications. It occurs when code fetches a list of objects and then makes a separate query per object to fetch linked objects.

Anti-pattern (N+1 — do not use):

```
# BAD: Fetches all units, then queries readiness for each individually
# N=1000 units -> 1001 queries to MSS
units = client.ontology.objects.Unit.all()
for unit in units:
    readiness = unit.links.readiness_report.get()  # 1 query per unit
    process(unit, readiness)
```

Correct pattern (bulk link loading):

```
# GOOD: Fetch units and readiness in bulk, join locally
# 2 queries total regardless of N
units = list(client.ontology.objects.Unit.all())
unit_rids = [u.rid for u in units]

# Bulk fetch linked readiness reports using filter on linked RID
readiness_reports = list(
    client.ontology.objects.ReadinessReport
    .where(ReadinessReport.unit_rid.is_in(unit_rids))
    .all()
)

# Build local index for O(1) lookup
readiness_by_unit = {r.unit_rid: r for r in readiness_reports}

for unit in units:
    readiness = readiness_by_unit.get(unit.rid)
    process(unit, readiness)
```

3-2b. Filter Push-Down

Always push filters to the OSDK query layer. Never retrieve all objects and filter in Python.

```
from datetime import datetime, timedelta

# BAD: Fetches all SITREPs, filters in Python
all_sitreps = client.ontology.objects.Sitrep.all()
recent = [s for s in all_sitreps if s.submitted_at > datetime.utcnow() -
timedelta(hours=24)]
```

```
# GOOD: Filter pushed to Foundry — only matching objects transferred
recent_sitreps = list(
    client.ontology.objects.Sitrep
    .where(Sitrep.submitted_at.gt(datetime.utcnow() - timedelta(hours=24)))
    .all()
)
```

3-2c. Projection — Fetch Only Required Properties

Fetching full objects when you only need two or three properties wastes network bandwidth and increases query latency. Use projection to fetch only what you need.

```
# BAD: Fetches all properties of all Unit objects
units = client.ontology.objects.Unit.all()
names = [u.unit_name for u in units]

# GOOD: Project to only the property needed
unit_names = [
    u.unit_name
    for u in client.ontology.objects.Unit
    .select(Unit.unit_name)  # Only transfer unit_name
    .all()
]
```

3-2d. Pagination and Cursor Management

Never assume a query returns all results in a single page. Always implement pagination for result sets that may be large.

```
from typing import Iterator, TypeVar
from foundry_sdk.ontology import OntologyObject

T = TypeVar("T", bound=OntologyObject)

def paginate_all(query, page_size: int = 500) -> Iterator[T]:
    """
    Generic paginator for OSDK object queries.
    Yields objects one at a time, fetching pages transparently.

    page_size: Tune based on object payload size.
      - Large objects (many properties): 100-200
      - Small objects (few properties): 500-1000
      - Never exceed platform-configured maximum page size
    """
    page = query.page(size=page_size)
    while True:
        for obj in page.data:
            yield obj
        if not page.next_page_token:
            break
        page = query.page(size=page_size, page_token=page.next_page_token)
```

3-3. Caching Strategies

CONDITIONS: High-frequency OSDK queries are driving unnecessary load on MSS. Reference data (unit rosters, equipment catalogs, organizational hierarchies) is queried repeatedly but changes infrequently.

STANDARDS: Cache reads are bounded by TTL appropriate to data freshness requirements. Cache is invalidated on write. Cached data is never served past its TTL to operational consumers. Classification markings are preserved — cached data is not served to consumers who lack authorization for the original data.

EQUIPMENT: Python (in-process TTL cache or Redis for distributed deployments); Redis with TLS if multi-instance (see 3-3c); C2DAO approval for Redis deployment in any MSS-connected environment.

PROCEDURE: 1. Identify reference data candidates for caching — data queried frequently but updated infrequently. 2. Determine appropriate TTL based on data type using the guidelines in 3-3b. 3. Select cache implementation: in-process TTL cache (3-3a) for single-instance services; distributed Redis cache (3-3c) for multi-instance services. 4. Scope cache instances per authorization context — never share cache across tenant boundaries. 5. Implement cache invalidation on all write paths that update cached data. 6. Do not cache operational status data (SITREP, readiness) — always fetch live.

WARNING

Never cache data across authorization boundaries. A cache that serves data from a higher-classification source to a lower-classification consumer violates the MSS security model. Cache entries must be scoped to the authorization context of the requesting user or service.**

3-3a. In-Process TTL Cache for Reference Data

```
import time
from threading import Lock
from typing import Any, Callable, Generic, TypeVar

V = TypeVar("V")

class TTLCache(Generic[V]):
    """
    Simple in-process TTL cache for Foundry reference data.

    SECURITY: This cache is process-scoped. Do not share across users or
    authorization contexts. Each service account that needs cached data
    should maintain its own cache instance.

    Usage:
        cache = TTLCache(ttl_seconds=300)  # 5-minute TTL
        data = cache.get_or_fetch("unit_roster", lambda: fetch_unit_roster(client))
    """

    def __init__(self, ttl_seconds: int = 300):
```

```
        self._store: dict[str, tuple[Any, float]] = {}
        self._lock = Lock()
        self._ttl = ttl_seconds

    def get_or_fetch(self, key: str, fetch_fn: Callable[[], V]) -> V:
        with self._lock:
            if key in self._store:
                value, expires_at = self._store[key]
                if time.monotonic() < expires_at:
                    return value
            # Cache miss or expired — fetch fresh
            value = fetch_fn()
            self._store[key] = (value, time.monotonic() + self._ttl)
            return value

    def invalidate(self, key: str) -> None:
        """Explicitly invalidate a cache entry — call after any write that updates the
key."""
        with self._lock:
            self._store.pop(key, None)

    def invalidate_all(self) -> None:
        """Flush entire cache — call on application config change or security
event."""
        with self._lock:
            self._store.clear()
```

3-3b. Cache TTL Guidelines for MSS Data Types

Data Type	Recommended TTL	Rationale
Unit organizational hierarchy	15 minutes	Changes infrequently; stale hierarchy causes misrouting
Equipment catalog	30 minutes	Reference data; new equipment added rarely
Personnel roster	5 minutes	Assignments change; stale roster causes access errors
SITREP / operational reports	No cache	Time-sensitive operational data; always fetch live
Readiness status	2 minutes	Operational tempo; commanders need current data
Classification markings	No cache	Marking changes must propagate immediately
Ontology schema (object type definitions)	60 minutes	Schema changes are rare and coordinated

CAUTION: Caching operational status data (readiness, SITREP) can produce stale reads during rapidly evolving operational situations. When in doubt, do not cache. The cost of a stale read in a tactical decision is higher than the cost of an additional Foundry query.

3-3c. Distributed Cache with Redis (Multi-Instance Services)

For services running multiple instances (containerized deployments, see Chapter 6), in-process caches are not shared across instances. Use a distributed cache.

```
import json
import redis
from typing import Any, Callable
from .config import CacheConfig

class DistributedFoundryCache:
    """
    Redis-backed distributed cache for multi-instance MSS service deployments.

    SECURITY: Redis instance must be:
    - Deployed within the same network boundary as the service
    - Not exposed externally
    - TLS-encrypted in transit
    - Password-protected
    Consult AR 25-2 and C2DAO guidance before deploying Redis in any MSS-connected
environment.
    """

    def __init__(self, cfg: CacheConfig):
        self._redis = redis.Redis(
            host=cfg.redis_host,
            port=cfg.redis_port,
            password=cfg.redis_password,  # From environment — never hardcoded
            ssl=True,                     # TLS required
            decode_responses=True,
        )
        self._default_ttl = cfg.default_ttl_seconds

    def get_or_fetch(
        self,
        key: str,
        fetch_fn: Callable[[], Any],
        ttl: int | None = None,
    ) -> Any:
        cached = self._redis.get(key)
        if cached:
            return json.loads(cached)
        value = fetch_fn()
        self._redis.setex(
            name=key,
            time=ttl or self._default_ttl,
            value=json.dumps(value, default=str),
        )
        return value

    def invalidate(self, key: str) -> None:
        self._redis.delete(key)
```

3-4. Indexing and Search Optimization

CONDITIONS: Object searches across large Foundry datasets are slow. Users are experiencing latency on search-heavy workflows (e.g., unit lookup by name, equipment search by serial number, personnel search by rank/unit combination).

STANDARDS: Search queries execute within defined SLA thresholds. Foundry Object Storage V2 (OSv2) indexes are correctly configured for high-frequency search properties. Full-text search is used only where appropriate — property equality and range filters use indexed paths.

EQUIPMENT: OSDK (Python); profiling/timing tools; access to C2DAO for index change requests; development MSS environment for query analysis.

PROCEDURE: 1. Profile the slow search queries using the analysis tool in 3-4a. 2. Map query patterns (equality, range, full-text, geospatial) to the appropriate index type using the table below. 3. Identify missing indexes based on profiling results (queries flagged as exceeding 500ms). 4. Document the business requirement and performance impact before requesting index changes. 5. Submit index change request to C2DAO — do not modify shared Object Type indexes independently. 6. Re-profile after index changes are applied to confirm improvement.

Foundry indexing principles:

Index Type	Use Case	Notes
Property index (equality)	Filter/search by exact value (unit_id, serial_number)	Low-cost; use for all primary lookup properties
Property index (range)	Filter by date range, numeric range	Enables efficient time-windowed queries
Full-text search index	Free-text search across string properties	Higher cost; use only for user-facing search, not programmatic filters
Geospatial index	Location-based queries (unit positions, AOR boundaries)	Required for any map-based operational application

Coordinate with C2DAO to add or modify indexes on shared Object Types. Index changes affect all consumers. Document the business requirement before requesting index changes.

3-4a. Query Pattern to Index Mapping

Before requesting index changes, map your query patterns to determine which indexes are missing:

```
# Query analysis tool — run against development environment only
# Identifies slow queries and suggests index candidates

def analyze_query_performance(
    client,
    object_type_name: str,
    sample_queries: list[dict],
```

```
) -> list[dict]:
    """
    Run sample queries and measure execution time.
    Use results to identify missing indexes.

    sample_queries format:
    [
        {"filter_property": "unit_id", "filter_value": "1-1-CAV", "type": "equality"},
        {"filter_property": "submitted_at", "filter_value": ("2026-01-01", "2026-02-
01"), "type": "range"},
    ]

    Returns list of results with timing and recommendations.
    """
    results = []
    for q in sample_queries:
        start = time.monotonic()
        # Execute sample query (implementation depends on OSDK version)
        elapsed = time.monotonic() - start
        results.append({
            "property": q["filter_property"],
            "type": q["type"],
            "elapsed_ms": elapsed * 1000,
            "needs_index": elapsed > 0.5,  # Flag queries exceeding 500ms
        })
    return results
```

3-5. Compute Cost Management

CONDITIONS: Your transforms, FOO functions, or OSDK services are consuming excessive compute resources on MSS, generating cost overruns or degrading platform performance for other tenants.

STANDARDS: Transforms are scheduled at appropriate intervals — not more frequently than their source data update rate. FOO computed properties use incremental computation where possible. Large analytical queries are batched, not executed on-demand per user request.

EQUIPMENT: Foundry transform environment; platform usage/cost monitoring (C2DAO-provided); MSS development environment for testing optimized patterns.

PROCEDURE: 1. Identify high-cost patterns using the principles table below. 2. For full-scan transforms, convert to `@incremental` with watermarks — see SL 4H. 3. For on-demand aggregations, pre-materialize results in a scheduled transform (see example below). 4. Identify and consolidate redundant transforms computing the same output. 5. Switch transform schedules from fixed-interval to event-triggered where source data changes are detectable. 6. Verify compute cost reduction with C2DAO platform monitoring before closing the optimization ticket.

Compute cost principles:

Pattern	Impact	Mitigation
Full dataset scan on each transform run	High CPU/IO	Use @incremental transforms with watermarks — see SL 4H
FOO property recomputed on every object fetch	High CPU per query	Pre-materialize frequently-used computed properties
On-demand aggregation across millions of objects	High memory/CPU	Pre-aggregate in scheduled transform; serve pre-aggregated results
Redundant transforms (multiple pipelines recomputing the same output)	Wasted compute	Identify shared compute; consolidate to single canonical transform
Transforms running at fixed schedule regardless of source data changes	Wasted compute	Trigger transforms on source data change events, not clock schedule

```
# Pattern: Pre-aggregated readiness summary
# Instead of computing readiness per user request, materialize daily/hourly summaries

# transform: compute_readiness_summary.py
# Runs hourly; outputs pre-aggregated by echelon
# Consumers query the summary dataset — not raw readiness objects

from transforms.api import transform_df, Input, Output
import pyspark.sql.functions as F

@transform_df(
    Output("/usareur/readiness/readiness_summary_hourly"),
    raw=Input("/usareur/readiness/readiness_raw"),
)
def compute_readiness_summary(raw: DataFrame) -> DataFrame:
    """
    Pre-aggregate readiness data by echelon and hour.
    Consumers query this output — avoids N*M on-demand aggregation.
    """
    return (
        raw
        .groupBy("echelon", "unit_type", F.date_trunc("hour", "reported_at"))
        .agg(
            F.count("*").alias("total_units"),
            F.sum(F.when(F.col("status") == "C1", 1).otherwise(0)).alias("c1_count"),
            F.sum(F.when(F.col("status") == "C2", 1).otherwise(0)).alias("c2_count"),
            F.sum(F.when(F.col("status").isin("C3", "C4"),
1).otherwise(0)).alias("degraded_count"),
            F.avg("readiness_pct").alias("avg_readiness_pct"),
        )
    )
```

3-6. Performance Profiling Tools and Procedures

NOTE — Palantir Developers reference: *Code in Production: Gallatin x Observability | DevCon 3* — A production case study on observability and monitoring for Foundry applications, covering how a real engineering team instruments and monitors their Foundry deployment — directly relevant to the profiling and performance incident procedures in this section. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

CONDITIONS: A production MSS application is underperforming. You need to systematically diagnose the bottleneck and document findings for C2DAO review.

STANDARDS: Profiling is performed in development or staging — never in production without C2DAO authorization. Findings are documented in a performance incident report. Proposed fixes are reviewed by a second SL 5L engineer before production deployment.

EQUIPMENT: Python `cProfile`, `pstats`, `io` modules; development or staging MSS environment with representative data volume; performance incident report template (see below); second SL 5L engineer for review.

PROCEDURE — Performance profiling workflow:

1. Reproduce the performance issue in development or staging with representative data volume.
2. Instrument the application with timing:

```
import cProfile
import pstats
import io

def profile_query_workflow(client, workflow_fn):
    """
    Profile a Foundry query workflow using cProfile.
    Use output to identify hot spots before submitting optimization request to C2DAO.
    """
    profiler = cProfile.Profile()
    profiler.enable()

    result = workflow_fn(client)

    profiler.disable()

    # Write profile to string for review
    stream = io.StringIO()
    stats = pstats.Stats(profiler, stream=stream)
    stats.sort_stats("cumulative")
    stats.print_stats(20)  # Top 20 functions by cumulative time

    print(stream.getvalue())
    return result
```

1. Review OSDK query logs for slow queries (if accessible in development environment).

2. Document findings in performance incident report format:

```
PERFORMANCE INCIDENT REPORT

Date: [DATE]
Reporter: [NAME / UNIT]
Application: [APP NAME / RID]
Environment: [DEV / STAGING / PROD]

SYMPTOM:
[Describe observed latency / degradation]

REPRODUCTION STEPS:
1. [Step 1]
2. [Step 2]

PROFILING RESULTS:
[Attach cProfile output or query timing data]

ROOT CAUSE HYPOTHESIS:
[N+1 queries / missing index / full scan / excessive compute]

PROPOSED FIX:
[Code change / index request / transform schedule change]

RISK ASSESSMENT:
[Impact of proposed fix on other consumers / tenants]

REVIEWER: [Second SL 5L engineer]
C2DAO COORDINATION: [Ticket ID if production change required]
```

1. Submit performance incident report to C2DAO if fix requires production changes.
2. Implement fix in development, re-profile to confirm improvement.
3. Promote through staging to production per DevSecOps pipeline (Chapter 6).

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

CHAPTER 4 — MULTI-TENANT ARCHITECTURE AND DATA ISOLATION

4-1. Scope of This Chapter

BLUF: MSS serves multiple commands, units, and coalition partners across USAREUR-AF and EUCOM. Multi-tenant architecture ensures each tenant sees only the data they are authorized to see. This chapter covers tenant isolation patterns, CBAC at scale, and coalition data sharing.

NOTE — Palantir Developers reference: *Product Launch: Edge Embedded Ontology | DevCon 2 —* Covers edge-embedded Ontology as an advanced deployment pattern where Ontology queries run without central infrastructure connectivity — a specialized architecture relevant to disconnected operations and forward-deployed MSS components in the European AOR. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

Multi-tenancy in MSS is not a convenience feature — it is a security and legal requirement. USAREUR-AF data shared with coalition partners is subject to NATO information sharing agreements. Unit operational data is need-to-know. Classification markings are enforced. A data bleed between tenants is a security incident with operational consequences.

4-2. Tenant Architecture in MSS

MSS multi-tenancy is implemented through a combination of:

Mechanism	What It Controls	Owner
CBAC (Content-Based Access Control)	Which objects/datasets a user or service account can read	C2DAO / G6
Foundry Markings	Classification and handling requirements on data	Data Steward
Dataset-level ACLs	Which Foundry groups can access a dataset	C2DAO
Ontology Visibility Rules	Which Object Types are visible to which users	C2DAO / Builders
Service Account Scoping	Which datasets/objects a service account can access	C2DAO

SL 5L engineers do not own CBAC schema — that is C2DAO. SL 5L engineers design applications that correctly consume and enforce the CBAC model.

Tenant boundary model:

```
MSS TENANT BOUNDARIES
```

```
+----------------------------------+
|  USAREUR-AF TOP LEVEL            |
|  (G3, G2 All-Source, C2DAO)      |
|                                  |
|  +------------+  +------------+  |
|  |  V CORPS   |  |  21ST TSC  |  |
|  |  Tenant    |  |  Tenant    |  |
|  +------------+  +------------+  |
|                                  |
|  +------------+  +------------+  |
|  | COALITION  |  |  7TH ATC   |  |
|  | Tenant     |  |  Tenant    |  |
|  | (MPE-gated)|  +------------+  |
|  +------------+                  |
+----------------------------------+
```

Coalition tenant data is additionally gated by Multinational Participation Environment (MPE) controls. No US-only data flows to the coalition tenant without explicit G6/J6 authorization.

4-3. Implementing Tenant Isolation in Code

CONDITIONS: You are building an MSS application that serves multiple tenant groups and must enforce isolation in application logic.

STANDARDS: Application never bypasses CBAC by fetching data on behalf of one tenant and serving it to another. Application enforces tenant context at every data access. Tenant context is authenticated — not user-supplied without validation.

EQUIPMENT: OSDK / Platform SDK (Python or TypeScript); C2DAO-provisioned tenant-scoped service accounts; DoD PKI or USAREUR-AF SSO for user authentication.

PROCEDURE: 1. Request C2DAO provisioning of a dedicated service account for each tenant, scoped to that tenant's data. 2. Initialize a `TenantRouter` with the per-tenant scoped clients (see 4-3c). 3. Authenticate all tenant context server-side from DoD PKI or SSO — never from user-supplied request parameters. 4. Route every data access request through `TenantRouter.get_client()`. 5. For coalition tenant access, validate MPE requirements via `_validate_coalition_access()` before routing. 6. Audit all tenant context routing events — log tenant ID, user ID, and data access type.

4-3a. Anti-Pattern: Server-Side Data Aggregation Across Tenants

```
# BAD: Service fetches all unit data (across all tenants) and filters by user claim
# This pattern trusts the user claim without server-side authorization
# A spoofed claim provides access to other tenants' data

def get_unit_data_BAD(user_tenant_claim: str) -> list:
    # Service account has broad access — fetches everything
    all_units = admin_client.ontology.objects.Unit.all()
    # Filter by user-supplied claim — UNSAFE
    return [u for u in all_units if u.tenant == user_tenant_claim]
```

4-3b. Correct Pattern: Tenant-Scoped Service Accounts

```
# GOOD: Each tenant has a dedicated service account scoped to that tenant's data
# The service account CBAC enforces isolation — no code-level filtering needed

# Service account provisioned by C2DAO with access limited to V Corps objects only
TENANT_SERVICE_ACCOUNTS = {
    "v_corps": "SA_V_CORPS_APP",
    "21st_tsc": "SA_21TSC_APP",
    # Additional tenants added by C2DAO — not by application developers
}

def get_unit_data_CORRECT(
    tenant_id: str,
    tenant_clients: dict[str, FoundryClient],  # Pre-initialized, scoped clients
) -> list:
    if tenant_id not in tenant_clients:
        raise PermissionError(f"No authorized client for tenant: {tenant_id}")

    # This client's CBAC limits it to the tenant's data — isolation enforced at
platform level
    client = tenant_clients[tenant_id]
    return list(client.ontology.objects.Unit.all())
```

4-3c. Multi-Tenant Request Routing

```
from dataclasses import dataclass
from enum import Enum

class TenantID(str, Enum):
    V_CORPS = "v_corps"
    TSC_21 = "21st_tsc"
    ATC_7 = "7th_atc"
    USAREUR_HQ = "usareur_hq"
    COALITION = "coalition"  # MPE-gated; additional controls apply

@dataclass
class TenantContext:
    tenant_id: TenantID
    user_id: str          # Authenticated user ID (from DoD PKI or SSO)
    clearance_level: str  # From authoritative source — not user-supplied
    coalition_nation: str | None = None  # Only for COALITION tenant

class TenantRouter:
    """
    Routes requests to tenant-scoped Foundry clients.
    Tenant context is validated server-side — never trusted from client request.
    """

    def __init__(self, tenant_clients: dict[TenantID, FoundryClient]):
        self._clients = tenant_clients

    def get_client(self, ctx: TenantContext) -> FoundryClient:
        """
```

```
        Return the tenant-scoped client for an authenticated tenant context.

        SECURITY: ctx must be populated from server-side authentication (DoD PKI,
        USAREUR-AF SSO), not from user-supplied request parameters.
        """
        if ctx.tenant_id == TenantID.COALITION:
            # Coalition access requires additional MPE validation
            self._validate_coalition_access(ctx)

        if ctx.tenant_id not in self._clients:
            raise PermissionError(
                f"No authorized MSS client for tenant {ctx.tenant_id}. "
                "Contact C2DAO for tenant provisioning."
            )

        return self._clients[ctx.tenant_id]

    def _validate_coalition_access(self, ctx: TenantContext) -> None:
        """
        Validate coalition access requirements.
        Called before routing to coalition-scoped client.
        """
        if not ctx.coalition_nation:
            raise PermissionError("Coalition access requires coalition_nation in
context.")
        # Additional MPE validation logic per G6/J6 guidance
        # Implementation specific to current MPE configuration — consult C2DAO
```

4-4. Classification-Aware Design

CONDITIONS: Your application serves data with varying classification markings (UNCLASSIFIED, CUI, SECRET). You must ensure the application correctly handles and communicates markings to users.

STANDARDS: Markings are displayed on every data product. Application never strips or suppresses markings. Aggregated products carry the highest marking of any source data. Application logs include marking metadata for audit.

EQUIPMENT: OSDK / Platform SDK (Python or TypeScript); classification marking library (see 4-4a); AR 380-5 marking reference; application UI framework.

PROCEDURE: 1. Implement `ClassificationLevel` enum and `CLASSIFICATION_BANNERS` lookup (see 4-4a). 2. Display the classification banner on every page, view, or report that presents data. 3. For aggregated products, compute the aggregate marking using `get_aggregate_marking()` before display. 4. Include classification marking in all audit log entries for data access events. 5. Never strip or suppress markings in API responses, exports, or data transformations. 6. Review marking display implementation during every code review using Appendix A checklist.

WARNING

An application that displays classified data without the appropriate banner/footer/marking indicators violates DoD Instruction 5200.01 and may constitute a security violation. Marking display is not optional. Build it in from the start.**

4-4a. Marking Display Pattern

```
from enum import Enum

class ClassificationLevel(str, Enum):
    UNCLASSIFIED = "UNCLASSIFIED"
    CUI = "CUI"
    SECRET = "SECRET"
    SECRET_NOFORN = "SECRET//NOFORN"

# Display strings for application UI — must match AR 380-5 requirements
CLASSIFICATION_BANNERS = {
    ClassificationLevel.UNCLASSIFIED: {
        "banner": "UNCLASSIFIED",
        "color": "#007a00",  # Green
        "css_class": "banner-unclassified",
    },
    ClassificationLevel.CUI: {
        "banner": "CUI",
        "color": "#502b85",  # Purple
        "css_class": "banner-cui",
    },
    ClassificationLevel.SECRET: {
        "banner": "SECRET",
        "color": "#c8102e",  # Red
        "css_class": "banner-secret",
    },
    ClassificationLevel.SECRET_NOFORN: {
        "banner": "SECRET//NOFORN",
        "color": "#c8102e",
        "css_class": "banner-secret-noforn",
    },
}

def get_aggregate_marking(markings: list[ClassificationLevel]) -> ClassificationLevel:
    """
    Return the highest classification level from a list of markings.
    An aggregate product must carry the highest marking of any source data.
    """
    level_order = [
        ClassificationLevel.UNCLASSIFIED,
        ClassificationLevel.CUI,
        ClassificationLevel.SECRET,
        ClassificationLevel.SECRET_NOFORN,
    ]
    highest = ClassificationLevel.UNCLASSIFIED
    for m in markings:
```

```
        if level_order.index(m) > level_order.index(highest):
            highest = m
    return highest
```

4-5. Coalition Interoperability Patterns

CONDITIONS: Your application must share data with coalition partners. Data must be filtered to authorized releasable information before it reaches coalition-facing interfaces.

STANDARDS: No US-only data is exposed via coalition interfaces. Releasability is enforced at the data layer, not the presentation layer. Coalition data flows are documented in data lineage. J6/G6 authorization is documented before any coalition integration goes live.

EQUIPMENT: OSDK / Platform SDK (Python or TypeScript); C2DAO-provisioned coalition-scoped service account; current COALITION_NATION_RELEASABILITY mapping (maintained by C2DAO/J6); written J6/G6 authorization.

PROCEDURE: 1. Obtain written J6/G6 authorization for the coalition integration before any implementation work. 2. Request C2DAO provisioning of a coalition-scoped service account with MPE controls. 3. Obtain the current `COALITION_NATION_RELEASABILITY` mapping from C2DAO/J6 — do not construct independently. 4. Implement `filter_for_coalition()` as a defense-in-depth control at the application data layer. 5. Ensure the coalition service account CBAC is the primary isolation control — application filtering is secondary. 6. Document all coalition data flows in dataset lineage before activation. 7. Verify with J6/G6 that the integration is active and authorized before enabling in production.

Coalition data releasability filter:

```
from typing import Literal

ReleasabilityCode = Literal["USA", "FVEY", "NATO", "PUBLIC"]

COALITION_NATION_RELEASABILITY: dict[str, list[ReleasabilityCode]] = {
    # This mapping is maintained by C2DAO / J6 — do not modify without authorization
    # Keys are NATO nation codes
    "GBR": ["USA", "FVEY", "NATO"],
    "DEU": ["NATO"],
    "FRA": ["NATO"],
    "POL": ["NATO"],
    # Additional nations per current USAREUR-AF coalition configuration
}

def filter_for_coalition(
    objects: list[dict],
    coalition_nation: str,
) -> list[dict]:
    """
    Filter object list to items releasable to a coalition partner nation.

    SECURITY: This function enforces the releasability boundary in application logic.
    It is a defense-in-depth control — CBAC at the Foundry layer is the primary
```

```
control.
    Never rely solely on this filter. Ensure coalition service accounts also have
    appropriately scoped CBAC.
    """
    allowed_codes = COALITION_NATION_RELEASABILITY.get(coalition_nation, [])
    if not allowed_codes:
        return []  # Unknown nation — return nothing, log for review

    return [
        obj for obj in objects
        if obj.get("releasability_code") in allowed_codes
    ]
```

NOTE

Coalition interoperability patterns are governed by NATO STANAG agreements and current USAREUR-AF J6 policy. The code patterns in this section are illustrative. Consult G6/J6 and C2DAO for current releasability matrices before implementing any coalition integration.

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.

CHAPTER 5 — HIGH-SCALE EXTERNAL INTEGRATION PATTERNS

5-1. Scope of This Chapter

BLUF: MSS does not operate in isolation. It integrates with external Army systems: GCSS-Army, DCPDS, AHLTA-T, ABCS, and external feeds from coalition systems. This chapter covers high-scale integration patterns: REST at scale, gRPC, and event streaming via Kafka and Kinesis.

Integration is a primary risk surface. External systems introduce data quality, security, and availability risks. Every integration point is a potential failure mode. Design for failure.

NOTE

All external integrations must be reviewed by C2DAO and authorized before going live. New integration points may require ATO amendment. Do not activate external integrations in production without documented C2DAO authorization and, where required, ATO impact assessment.

5-2. REST Integration at Scale

CONDITIONS: You are building or managing a high-throughput REST integration between an external Army system and MSS. The integration must handle thousands of records per minute with guaranteed delivery.

STANDARDS: Integration is resilient to target system downtime. Failed deliveries are queued for retry. Duplicate deliveries are idempotent. Integration performance is monitored with alerting on failure thresholds.

EQUIPMENT: Python `httpx` library; Platform SDK (Python) for Foundry writes; dead-letter storage (Foundry dataset or approved equivalent); C2DAO authorization documentation for the integration.

PROCEDURE: 1. Obtain C2DAO written authorization for the external integration before implementation. 2. Implement `ResilientRestClient` with exponential backoff, circuit breaker, and dead-letter queue (see 5-2a). 3. Implement idempotent ingestion using `ingest_with_deduplication()` with content hash keys (see 5-2b). 4. Configure monitoring and alerting on dead-letter queue depth and circuit breaker state. 5. Test failure scenarios in development: target system downtime, duplicate delivery, malformed payloads. 6. Document the integration data flow in dataset lineage before production activation.

5-2a. Resilient REST Client Pattern

```
import time
import logging
from typing import Any
import httpx

logger = logging.getLogger(__name__)

class ResilientRestClient:
    """
    HTTP client with exponential backoff, circuit breaker, and dead-letter queue.
    Use for all high-scale REST integrations connecting to external Army systems.
    """

    def __init__(
        self,
        base_url: str,
        token: str,
        max_retries: int = 5,
        initial_backoff_seconds: float = 1.0,
        circuit_breaker_threshold: int = 10,
    ):
        self._base_url = base_url
        self._headers = {"Authorization": f"Bearer {token}"}
        self._max_retries = max_retries
        self._backoff = initial_backoff_seconds
        self._failure_count = 0
        self._circuit_open = False
        self._circuit_threshold = circuit_breaker_threshold
        self._dead_letters: list[dict] = []
```

```
    def post(self, path: str, payload: dict) -> httpx.Response | None:
        """
        POST with retry, backoff, and circuit breaker.
        Returns response on success; None if all retries exhausted (payload dead-
lettered).
        """
        if self._circuit_open:
            logger.warning(
                "Circuit breaker OPEN — not attempting POST to %s. "
                "External system may be down. Dead-lettering payload.",
                path
            )
            self._dead_letters.append({"path": path, "payload": payload})
            return None

        url = f"{self._base_url}{path}"
        backoff = self._backoff

        for attempt in range(self._max_retries):
            try:
                with httpx.Client(timeout=30) as client:
                    response = client.post(url, json=payload, headers=self._headers)
                    response.raise_for_status()
                    self._failure_count = 0  # Reset on success
                    return response

            except (httpx.HTTPError, httpx.TimeoutException) as exc:
                self._failure_count += 1
                logger.warning(
                    "POST attempt %d/%d failed: %s. Backing off %.1fs.",
                    attempt + 1, self._max_retries, exc, backoff
                )

                if self._failure_count >= self._circuit_threshold:
                    self._circuit_open = True
                    logger.error(
                        "Circuit breaker OPENED after %d failures. "
                        "Manual reset required after external system recovery.",
                        self._failure_count
                    )

                if attempt < self._max_retries - 1:
                    time.sleep(backoff)
                    backoff = min(backoff * 2, 60)  # Cap at 60s

        # All retries exhausted
        self._dead_letters.append({"path": path, "payload": payload})
        logger.error(
            "All retries exhausted for POST to %s. Payload dead-lettered. "
            "Dead letter queue size: %d.",
            path, len(self._dead_letters)
        )
        return None
```

```
    def get_dead_letters(self) -> list[dict]:
        """Return dead-lettered payloads for manual review or retry."""
        return list(self._dead_letters)

    def reset_circuit(self) -> None:
        """Manually reset circuit breaker after external system recovery."""
        self._circuit_open = False
        self._failure_count = 0
        logger.info("Circuit breaker reset. Resuming normal operation.")
```

5-2b. Idempotent Ingestion Pattern

External systems may deliver duplicate records. All ingestion code must be idempotent — processing the same record twice must produce the same result as processing it once.

```
import hashlib
import json

def compute_record_hash(record: dict, key_fields: list[str]) -> str:
    """
    Compute a deterministic hash for a record based on key fields.
    Used to detect and deduplicate duplicate deliveries.
    """
    key_data = {k: record.get(k) for k in sorted(key_fields)}
    serialized = json.dumps(key_data, sort_keys=True, default=str)
    return hashlib.sha256(serialized.encode()).hexdigest()


def ingest_with_deduplication(
    records: list[dict],
    key_fields: list[str],
    seen_hashes: set[str],  # Persistent set backed by dataset or Redis
) -> tuple[list[dict], list[dict]]:
    """
    Split records into new (ingest) and duplicate (skip) sets.

    seen_hashes: Persistent set of already-processed record hashes.
    Backed by a Foundry dataset or Redis for durability across restarts.

    Returns: (new_records, duplicate_records)
    """
    new_records = []
    duplicate_records = []

    for record in records:
        record_hash = compute_record_hash(record, key_fields)
        if record_hash in seen_hashes:
            duplicate_records.append(record)
        else:
            new_records.append(record)
            seen_hashes.add(record_hash)

    return new_records, duplicate_records
```

5-3. gRPC Integration Patterns

CONDITIONS: High-throughput, low-latency integrations with internal Army systems that support gRPC (e.g., real-time sensor feeds, streaming operational data).

STANDARDS: gRPC connections use mutual TLS (mTLS). Service definitions are versioned. Breaking changes to service contracts require coordination with consuming systems. Streaming gRPC connections implement reconnection logic.

EQUIPMENT: Python `grpc` library; DoD PKI-issued client certificates (or equivalent per AR 25-2); Protocol Buffer definitions for the target service; C2DAO authorization for the integration.

PROCEDURE: 1. Obtain DoD PKI-issued or equivalent client certificates for mTLS authentication. 2. Store certificate files on the filesystem referenced by environment configuration — never embed in code. 3. Create the mTLS channel using `create_mtls_channel()` with certificate paths from config (see 5-3a). 4. For streaming connections, wrap with `stream_with_reconnection()` to handle network disruptions (see 5-3b). 5. Version all service definitions — coordinate breaking changes with all consuming systems before deployment. 6. Test reconnection behavior in development by simulating connection drops.

5-3a. gRPC Client Setup with mTLS

```
import grpc
from pathlib import Path

def create_mtls_channel(
    server_address: str,
    client_cert_path: Path,
    client_key_path: Path,
    ca_cert_path: Path,
) -> grpc.Channel:
    """
    Create a gRPC channel with mutual TLS authentication.

    SECURITY: All gRPC connections to Army systems must use mTLS.
    Certificate files must be stored outside the codebase — reference by path
    from environment configuration, never embedded in code.

    Certificates must be DoD PKI-issued or equivalent per AR 25-2.
    """
    # Load certificates from filesystem — never embed in code
    with open(client_cert_path, "rb") as f:
        client_cert = f.read()
    with open(client_key_path, "rb") as f:
        client_key = f.read()
    with open(ca_cert_path, "rb") as f:
        ca_cert = f.read()

    credentials = grpc.ssl_channel_credentials(
        root_certificates=ca_cert,
        private_key=client_key,
        certificate_chain=client_cert,
    )
```

```
    return grpc.secure_channel(server_address, credentials)
```

5-3b. Streaming gRPC with Reconnection

```
import grpc
import time
import logging
from typing import Iterator

logger = logging.getLogger(__name__)

def stream_with_reconnection(
    stub_factory,
    channel_factory,
    request,
    max_reconnect_attempts: int = 10,
    reconnect_backoff_seconds: float = 5.0,
) -> Iterator:
    """
    Stream gRPC messages with automatic reconnection on connection loss.

    MSS integrations that receive continuous operational feeds (e.g., position
reports,
    sensor data) must handle connection interruptions gracefully — network disruptions
    in the European theater are operational realities.
    """
    attempt = 0
    while attempt < max_reconnect_attempts:
        try:
            channel = channel_factory()
            stub = stub_factory(channel)
            for message in stub.StreamData(request):
                attempt = 0  # Reset counter on successful message
                yield message

        except grpc.RpcError as exc:
            attempt += 1
            logger.warning(
                "gRPC stream disconnected (attempt %d/%d): %s. "
                "Reconnecting in %.1fs.",
                attempt, max_reconnect_attempts, exc.details(),
reconnect_backoff_seconds
            )
            time.sleep(reconnect_backoff_seconds)

    raise RuntimeError(
        f"gRPC stream failed after {max_reconnect_attempts} reconnection attempts. "
        "Check network connectivity and remote service health."
    )
```

5-4. Event Streaming: Kafka Integration

CONDITIONS: Your integration requires high-throughput event streaming — external systems push operational events (position updates, status changes, sensor readings) at rates that exceed REST polling capacity.

STANDARDS: Kafka consumers are idempotent. Consumer group offsets are committed only after successful processing. Dead-letter topic captures failed messages. Partition assignment is monitored. Consumer lag is alerted.

EQUIPMENT: Python `kafka-python` library; Platform SDK (Python) for Foundry writes; C2DAO-provisioned Kafka broker connection details and topic configuration; dead-letter Foundry dataset (C2DAO-provisioned).

> **NOTE**
>
> Kafka deployments in the USAREUR-AF environment must be approved by C2DAO. Do not deploy Kafka independently. This section covers patterns for connecting to C2DAO-provisioned Kafka infrastructure. Contact C2DAO for broker connection details and topic provisioning.

PROCEDURE: 1. Obtain Kafka broker connection details and topic assignment from C2DAO. 2. Initialize `FoundryKafkaIngester` with `enable_auto_commit=False` to ensure manual offset control. 3. Configure dead-letter dataset RID — all failed messages must be captured for review. 4. Implement `_validate_message()` with required field checks and schema sanitization. 5. Run the ingestion loop: process batches, write to Foundry, commit offsets only after successful write. 6. Monitor consumer lag and dead-letter queue depth. Alert if either exceeds defined thresholds.

5-4a. Foundry-Kafka Ingestion Pipeline

```
from kafka import KafkaConsumer
from kafka.errors import KafkaError
import json
import logging
from foundry_sdk import FoundryClient

logger = logging.getLogger(__name__)

class FoundryKafkaIngester:
    """
    Consume events from a Kafka topic and write to Foundry dataset.

    Pattern: at-least-once delivery with idempotent writes.
    Offsets committed after successful Foundry write — not before.
    Failed messages written to dead-letter dataset for review.
    """

    def __init__(
        self,
        kafka_config: dict,        # From environment config — broker, auth, SSL
```

```
        topic: str,
        foundry_client: FoundryClient,
        target_dataset_rid: str,
        dead_letter_dataset_rid: str,
        target_branch: str = "master",
    ):
        self._consumer = KafkaConsumer(
            topic,
            **kafka_config,
            enable_auto_commit=False,  # Manual offset commit for at-least-once
            auto_offset_reset="earliest",
            value_deserializer=lambda v: json.loads(v.decode("utf-8")),
        )
        self._client = foundry_client
        self._target_rid = target_dataset_rid
        self._dead_letter_rid = dead_letter_dataset_rid
        self._branch = target_branch

    def run(self, batch_size: int = 100) -> None:
        """
        Main ingestion loop. Processes messages in batches for efficiency.
        Commits offsets only after successful Foundry write.
        """
        batch = []

        for message in self._consumer:
            batch.append(message)

            if len(batch) >= batch_size:
                self._process_batch(batch)
                self._consumer.commit()
                batch = []

    def _process_batch(self, messages: list) -> None:
        """Process a batch of Kafka messages with dead-letter handling."""
        valid_records = []
        failed_records = []

        for msg in messages:
            try:
                validated = self._validate_message(msg.value)
                valid_records.append(validated)
            except (ValueError, KeyError) as exc:
                logger.warning("Message validation failed: %s. Dead-lettering.", exc)
                failed_records.append({
                    "topic": msg.topic,
                    "partition": msg.partition,
                    "offset": msg.offset,
                    "raw_value": str(msg.value),
                    "error": str(exc),
                })

        if valid_records:
            self._write_to_foundry(valid_records, self._target_rid)
        if failed_records:
```

```
            self._write_to_foundry(failed_records, self._dead_letter_rid)

    def _validate_message(self, data: dict) -> dict:
        """
        Validate and sanitize message before Foundry ingestion.
        All external data must be validated at this boundary.
        Required fields: source_system, event_type, timestamp, payload
        """
        required = ["source_system", "event_type", "timestamp", "payload"]
        missing = [f for f in required if f not in data]
        if missing:
            raise ValueError(f"Missing required fields: {missing}")
        # Sanitize: remove any keys not in expected schema
        return {k: data[k] for k in required}

    def _write_to_foundry(self, records: list[dict], dataset_rid: str) -> None:
        """Write validated records to Foundry dataset."""
        import pandas as pd
        df = pd.DataFrame(records)
        with self._client.datasets.transactions.start(
            dataset_rid=dataset_rid,
            branch=self._branch,
            transaction_type="APPEND",
        ) as txn:
            txn.write_pandas(df)
```

NOTE

APPEND transactions are not inherently idempotent. Re-runs will write duplicate rows unless you implement deduplication (content hash + INSERT OR IGNORE, or surrogate key). Use SNAPSHOT for atomic full-dataset replacement. For Kafka at-least-once delivery, implement a deduplication key (e.g., message offset + partition) in the target dataset to prevent duplicate rows on consumer replay.

5-5. Event Streaming: Kinesis Integration

CONDITIONS: Your operational environment uses AWS Kinesis Data Streams for event streaming (common in cloud-connected Army deployments). You need to connect Kinesis streams to MSS.

STANDARDS: Same as Kafka — idempotent processing, dead-letter handling, at-least-once delivery.

EQUIPMENT: Python `boto3` library; AWS IAM role or least-privilege service account credentials (from approved secrets manager); Platform SDK (Python) for Foundry writes; C2DAO cloud guidance for Kinesis use in MSS-connected environments.

PROCEDURE: 1. Configure AWS credentials via IAM role (preferred) or least-privilege service account from approved secrets manager — never hardcode AWS keys. 2. Initialize `FoundryKinesisIngester` with stream name, region, and Foundry client. 3. Enumerate all shards using `get_all_shards()` — deploy one consumer per shard for parallel throughput. 4. For each shard, call `consume_shard()` with a

checkpoint function to persist sequence numbers across restarts. 5. In `_write_to_foundry()`, validate and sanitize records before Foundry write. 6. Monitor shard iterator expiration and implement reconnection if iterator expires.

```
import boto3
import json
import time
from foundry_sdk import FoundryClient

class FoundryKinesisIngester:
    """
    Consume records from AWS Kinesis Data Stream and write to Foundry.

    NOTE: AWS credentials for Kinesis access must be provisioned per AR 25-2
    and stored in approved secrets management infrastructure. Never hardcode
    AWS access keys. Use IAM roles where possible (preferred); service accounts
    with least-privilege policies otherwise.
    """

    def __init__(
        self,
        stream_name: str,
        aws_region: str,
        foundry_client: FoundryClient,
        target_dataset_rid: str,
        target_branch: str = "master",
        poll_interval_seconds: float = 1.0,
    ):
        # Credentials from environment / IAM role — not hardcoded
        self._kinesis = boto3.client("kinesis", region_name=aws_region)
        self._stream_name = stream_name
        self._foundry_client = foundry_client
        self._target_rid = target_dataset_rid
        self._branch = target_branch
        self._poll_interval = poll_interval_seconds

    def get_all_shards(self) -> list[str]:
        """Discover all shard IDs for the stream."""
        response = self._kinesis.describe_stream_summary(StreamName=self._stream_name)
        shards = response["StreamDescriptionSummary"]["ShardLevelMetrics"]
        # In practice, use list_shards for accurate shard enumeration
        response = self._kinesis.list_shards(StreamName=self._stream_name)
        return [s["ShardId"] for s in response["Shards"]]

    def consume_shard(
        self,
        shard_id: str,
        checkpoint_fn,    # Callable to persist/retrieve shard iterator
    ) -> None:
        """
        Consume records from a single Kinesis shard with checkpointing.
        Deploy one consumer per shard for parallel throughput.
        """
        shard_iterator = checkpoint_fn(shard_id) or self._kinesis.get_shard_iterator(
```

```
            StreamName=self._stream_name,
            ShardId=shard_id,
            ShardIteratorType="TRIM_HORIZON",
        )["ShardIterator"]

        while shard_iterator:
            response = self._kinesis.get_records(
                ShardIterator=shard_iterator,
                Limit=500,
            )
            records = response["Records"]

            if records:
                parsed = [json.loads(r["Data"]) for r in records]
                self._write_to_foundry(parsed)
                # Checkpoint the last sequence number
                checkpoint_fn(shard_id, records[-1]["SequenceNumber"])

            shard_iterator = response.get("NextShardIterator")
            time.sleep(self._poll_interval)

    def _write_to_foundry(self, records: list[dict]) -> None:
        import pandas as pd
        df = pd.DataFrame(records)
        with self._foundry_client.datasets.transactions.start(
            dataset_rid=self._target_rid,
            branch=self._branch,
            transaction_type="APPEND",
        ) as txn:
            txn.write_pandas(df)
```

NOTE

APPEND transactions are not inherently idempotent. Re-runs will write duplicate rows unless you implement deduplication (content hash + INSERT OR IGNORE, or surrogate key). Use SNAPSHOT for atomic full-dataset replacement. For Kinesis at-least-once delivery, implement a deduplication key (e.g., Kinesis sequence number) in the target dataset to prevent duplicate rows on shard replay.

5-6. Advanced UDRA Implementation: Computational Governance as Code

BLUF: UDRA v1.1 mandates that data governance move from manual policy enforcement to computational governance — standards as code, policies as code, and incentives as code. The SL 5L SWE implements this by embedding Policy Decision Points (PDPs) and Policy Enforcement Points (PEPs) directly into pipeline architecture.

A **PDP** evaluates a governance rule against the current context (user identity, data classification, action requested) and returns an allow/deny decision. A **PEP** intercepts data flow at a pipeline boundary and enforces the PDP's decision. Every external integration point (Chapter 5), every tenant boundary

(Chapter 4), and every deployment promotion (Chapter 6) is a candidate PEP location.

Implementation pattern: Define governance policies as machine-readable rules (JSON/YAML policy documents version-controlled in the MSS repository). PDPs consume these rules at runtime. PEPs call PDPs before permitting data to cross a boundary. This eliminates manual policy review for routine operations and produces a complete, auditable enforcement log.

5-7. DDIL-Resilient Pipeline Design

Deployed MSS nodes must operate in Denied, Degraded, Intermittent, and Limited (DDIL) bandwidth environments. The SL 5L SWE designs pipelines that degrade gracefully when connectivity to the primary Foundry environment is lost. Source: Army Data Plan SO 6; UDRA Appendix E.

Store-and-forward: Edge nodes queue pipeline outputs locally when upstream connectivity is unavailable. On reconnection, queued outputs transmit in order with deduplication at the receiving end (content hash or idempotency key). Design pipelines so that every output record is idempotent — re-processing the same input produces the same output without side effects.

Delta synchronization: Transmit only changed records, not full dataset snapshots. Use change data capture (CDC) or timestamp-based incremental reads to minimize bandwidth. DDIL environments cannot tolerate full-dataset transfers on every sync cycle.

Graceful degradation: Define explicit degradation tiers for each pipeline: Tier 1 (full connectivity — normal operation), Tier 2 (intermittent — batch sync at reduced frequency), Tier 3 (denied — local-only operation with queued outputs). Each tier has defined behavior; pipelines must not fail silently when connectivity degrades.

NOTE — Zero trust architecture alignment: UDRA mandates Attribute-Based Access Control (ABAC) for all data access, with encryption at rest, in motion, and in use, per DoD Zero Trust Strategy and NIST 800-39/800-37. The SL 5L SWE ensures that PDP/PEP enforcement (Section 5-6) aligns with zero trust principles: no implicit trust based on network location, every access request authenticated and authorized at the PDP, all data encrypted across every pipeline boundary. DDIL-resilient pipelines (Section 5-7) must maintain zero trust enforcement even when operating in degraded mode — store-and-forward queues encrypt data at rest, and reconnection re-authenticates before transmitting queued outputs.

CHAPTER 6 — DEVSECOPS FOR FOUNDRY ENVIRONMENTS

6-1. Scope of This Chapter

BLUF: DevSecOps integrates security into every phase of the software development lifecycle — not as a gate at the end, but as a continuous practice. For MSS, DevSecOps supports ATO maintenance, reduces time-to-detection for security issues, and ensures code quality across the USAREUR-AF SWE community.

NOTE — Palantir Developers reference: *Product Launch: Developer Experience* | *DevCon 5* — Covers developer experience improvements announced at DevCon 5, including toolchain enhancements relevant to CI/CD pipeline setup and the DevSecOps workflow described in this chapter. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

SL 4L covered CI/CD fundamentals. This chapter covers the full DevSecOps pipeline: security scanning, Ontology CI, automated testing at scale, ATO-supporting artifact generation, and the governance processes that connect technical practices to the RMF/ATO lifecycle.

6-2. CI/CD Pipeline Architecture for MSS

The MSS DevSecOps pipeline:

```
DEVELOPER LOCAL ENVIRONMENT
  - Pre-commit hooks (linting, secrets detection, unit tests)
  |
  v
FEATURE BRANCH (Foundry Code Repository)
  - Pull request triggers CI pipeline
  |
  v
CI PIPELINE (automated):
  1. Static analysis (ruff / flake8 / mypy for Python; eslint / tsc for TypeScript)
  2. Secrets scanning (detect-secrets, TruffleHog)
  3. Dependency vulnerability scan (pip-audit, npm audit)
  4. Unit tests (pytest / jest) — minimum 80% coverage gate
  5. Integration tests (against dev/staging MSS environment)
  6. SAST (Semgrep rules — see Chapter 7)
  7. OWASP dependency check
  8. Artifact generation (see 6-5 for ATO artifacts)
  |
  v (all gates pass)
CODE REVIEW (second SL 5L engineer)
  |
  v
STAGING BRANCH PROMOTION
  - Automated deployment to staging MSS environment
  - Smoke tests against staging
```

```
  |
  v
C2DAO REVIEW + APPROVAL (for production changes)
  |
  v
PRODUCTION PROMOTION
  - Automated deployment to production MSS
  - Automated regression tests
  - Monitoring baseline update
```

6-3. Pre-Commit Hooks

CONDITIONS: You are establishing development standards for an MSS SWE team. All developers must enforce baseline code quality and security checks before code reaches the repository.

STANDARDS: Pre-commit hooks run on every commit. Hooks cannot be bypassed without explicit team lead approval. Secrets detection is mandatory on all repositories containing Foundry-related code.

EQUIPMENT: `pre-commit` Python package; `ruff`, `mypy`, `detect-secrets` tools; `node`/`npm` for TypeScript hooks; internet access to approved package sources (or mirror approved by C2DAO).

PROCEDURE: 1. Install `pre-commit`: `pip install pre-commit`. 2. Copy the MSS standard `.pre-commit-config.yaml` (below) into the repository root. 3. Initialize the secrets detection baseline: `detect-secrets scan > .secrets.baseline`. 4. Install hooks: `pre-commit install`. 5. Run hooks against all existing files to establish baseline: `pre-commit run --all-files`. 6. Resolve any findings before committing the hook configuration. 7. Instruct all team members to run `pre-commit install` in their local clones.

`.pre-commit-config.yaml` for MSS projects:

```
# .pre-commit-config.yaml
# MSS standard pre-commit configuration
# Install: pip install pre-commit && pre-commit install

repos:
  # Python formatting
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.4.0
    hooks:
      - id: ruff
        args: [--fix]
      - id: ruff-format

  # Python type checking
  - repo: https://github.com/pre-commit/mirrors-mypy
    rev: v1.10.0
    hooks:
      - id: mypy
        additional_dependencies: [types-all]

  # Secrets detection — mandatory on all MSS repos
```

```
  - repo: https://github.com/Yelp/detect-secrets
    rev: v1.4.0
    hooks:
      - id: detect-secrets
        args: ['--baseline', '.secrets.baseline']

  # General file hygiene
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.6.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
      - id: check-json
      - id: check-merge-conflict
      - id: check-added-large-files
        args: ['--maxkb=1000']  # Flag files over 1MB
      - id: no-commit-to-branch
        args: ['--branch', 'master', '--branch', 'main', '--branch', 'staging']

  # TypeScript
  - repo: local
    hooks:
      - id: eslint
        name: ESLint
        entry: npx eslint --fix
        language: node
        files: \.(ts|tsx)$
      - id: typescript-check
        name: TypeScript type check
        entry: npx tsc --noEmit
        language: node
        files: \.(ts|tsx)$
        pass_filenames: false
```

NOTE

The `no-commit-to-branch` hook prevents direct commits to master/staging. This enforces the PR-based workflow. Do not remove this hook. Direct commits to production branches bypass code review.

6-4. Automated Testing Standards

CONDITIONS: You are establishing or reviewing the automated testing framework for an MSS application or pipeline.

STANDARDS: Unit test coverage minimum 80% for all production code. Integration tests cover all OSDK query paths and external integration points. Tests are deterministic — no test that passes sometimes and fails others. Tests run in under 10 minutes total (parallelize if needed).

EQUIPMENT: Python `pytest`, `pytest-cov`, `pytest-xdist` packages; `unittest.mock` for mocking Foundry clients; access to dev/staging MSS environment for integration tests.

PROCEDURE: 1. Implement `conftest.py` with `mock_foundry_client` fixture for unit tests (see 6-4a). 2. Create `integration_foundry_client` fixture that skips unless `RUN_INTEGRATION_TESTS=1` is set. 3. Organize tests: `tests/unit/` for unit tests; `tests/integration/` for integration tests. 4. Run unit tests with coverage gate: `pytest tests/unit/ --cov=src --cov-fail-under=80`. 5. Run integration tests against dev MSS as part of CI pipeline (not in pre-commit). 6. Parallelize test execution in CI using `pytest -n auto` to meet the 10-minute runtime target.

6-4a. pytest configuration for MSS projects:

```
# conftest.py — MSS standard test fixtures

import pytest
from unittest.mock import MagicMock, patch
from foundry_sdk import FoundryClient

@pytest.fixture
def mock_foundry_client():
    """
    Mock FoundryClient for unit tests.
    Never connect to production MSS in unit tests.
    Use integration tests (with dev environment) for end-to-end validation.
    """
    client = MagicMock(spec=FoundryClient)

    # Pre-configure common mock behaviors
    client.ontology.objects.Unit.all.return_value = iter([])
    client.datasets.transactions.start.return_value.__enter__ = MagicMock()
    client.datasets.transactions.start.return_value.__exit__ =
MagicMock(return_value=False)

    return client

@pytest.fixture
def sample_unit_data():
    """Sample unit data for testing — uses realistic but non-operational data."""
    return [
        {"unit_id": "1-1-CAV", "unit_name": "1st Squadron, 1st Cavalry", "echelon":
"BN", "status": "C1"},
        {"unit_id": "2-1-AD", "unit_name": "2nd Battalion, 1st Armor", "echelon":
"BN", "status": "C2"},
        {"unit_id": "3-1-FA", "unit_name": "3rd Battalion, 1st Field Artillery",
"echelon": "BN", "status": "C1"},
    ]

@pytest.fixture
def integration_foundry_client():
    """
    Real FoundryClient for integration tests.
    Connects to dev/staging MSS environment only.
    Requires FOUNDRY_URL and FOUNDRY_SA_TOKEN environment variables.
    Skip if not in integration test environment.
    """
    import os
```

```
    if not os.getenv("RUN_INTEGRATION_TESTS"):
        pytest.skip("Integration tests disabled. Set RUN_INTEGRATION_TESTS=1 to
enable.")

    from foundry_sdk import FoundryClient
    return FoundryClient(
        hostname=os.environ["FOUNDRY_URL"],
        auth=os.environ["FOUNDRY_SA_TOKEN"],
    )
```

6-4b. Test categories and execution:

```
# Run unit tests only (fast, no external connections)
pytest tests/unit/ -v --tb=short

# Run with coverage report
pytest tests/unit/ --cov=src --cov-report=term-missing --cov-fail-under=80

# Run integration tests against dev MSS (requires credentials)
RUN_INTEGRATION_TESTS=1 pytest tests/integration/ -v --tb=long

# Run full suite in CI (parallel)
pytest tests/ -n auto --dist=loadfile
```

6-5. Ontology CI

CONDITIONS: Your team modifies the Foundry Ontology (Object Types, Link Types, Actions, FOO functions). Ontology changes must be validated before deployment to production.

STANDARDS: Ontology changes are tested in an isolated development branch before promotion. Automated checks validate schema compatibility. Breaking changes are flagged and require C2DAO approval.

EQUIPMENT: Foundry Ontology branch (development); CI pipeline; `ontology_diff.py` schema comparison script (below); Python for schema diff tooling.

PROCEDURE: 1. Make all Ontology changes on a development branch — never directly on master. 2. Export schema from both the development branch and the current production branch. 3. Run `detect_breaking_changes()` against the two schema exports (see script below). 4. If breaking changes detected, halt promotion — obtain C2DAO approval before proceeding. 5. Run all FOO TypeScript unit tests and Action validator tests against the development branch. 6. After all CI checks pass and (if applicable) C2DAO approval is obtained, promote through staging to production per DevSecOps pipeline.

Ontology CI checks:

Check	What It Validates	Fail Condition
Schema compatibility	New Object Type properties don't break existing OSDK consumers	Required property added without default; property type changed

Check	What It Validates	Fail Condition
CBAC rule completeness	All new Object Types have CBAC rules defined	Object Type with no visibility rules
Link Type validity	Link Types reference existing Object Types on both ends	Broken link type reference
FOO function tests	TypeScript FOO unit tests pass	Any FOO test failure
Action validator tests	Action TypeScript validators pass test suite	Any validator test failure
Breaking change detection	Comparison against previous schema version	Any backward-incompatible change

Ontology schema diff script (run in CI):

```
# ontology_diff.py — compare Ontology schema versions for breaking changes

from dataclasses import dataclass
import json

@dataclass
class SchemaDiff:
    added_types: list[str]
    removed_types: list[str]
    modified_properties: list[dict]
    breaking_changes: list[str]

def detect_breaking_changes(
    old_schema: dict,
    new_schema: dict,
) -> SchemaDiff:
    """
    Compare two Ontology schema exports and identify breaking changes.

    Breaking changes require C2DAO approval before promotion.
    Non-breaking changes (additive) can proceed through standard review.

    Breaking changes:
    - Removing an Object Type
    - Removing a property from an Object Type
    - Changing a property type
    - Making an optional property required

    Non-breaking changes:
    - Adding a new Object Type
    - Adding an optional property to an existing Object Type
    - Adding a new Link Type
    """
    old_types = {t["apiName"]: t for t in old_schema.get("objectTypes", [])}
```

```
    new_types = {t["apiName"]: t for t in new_schema.get("objectTypes", [])}

    breaking = []

    # Check for removed types
    removed = set(old_types) - set(new_types)
    for t in removed:
        breaking.append(f"BREAKING: Object Type removed: {t}")

    # Check for modified properties on existing types
    modified = []
    for type_name in set(old_types) & set(new_types):
        old_props = {p["apiName"]: p for p in old_types[type_name].get("properties",
[])}
        new_props = {p["apiName"]: p for p in new_types[type_name].get("properties",
[])}

        for prop_name in set(old_props) - set(new_props):
            breaking.append(f"BREAKING: Property removed: {type_name}.{prop_name}")

        for prop_name in set(old_props) & set(new_props):
            old_type = old_props[prop_name].get("dataType")
            new_type = new_props[prop_name].get("dataType")
            if old_type != new_type:
                breaking.append(
                    f"BREAKING: Property type changed: {type_name}.{prop_name} "
                    f"({old_type} -> {new_type})"
                )
                modified.append({"type": type_name, "property": prop_name, "old":
old_type, "new": new_type})

    return SchemaDiff(
        added_types=list(set(new_types) - set(old_types)),
        removed_types=list(removed),
        modified_properties=modified,
        breaking_changes=breaking,
    )
```

6-6. Container Deployment Pattern

CONDITIONS: MSS integration services and external applications are deployed as containers. You need to establish secure container build and deployment patterns.

STANDARDS: Container images are built from approved base images. No secrets in image layers. Images are scanned for vulnerabilities before deployment. Container runtime uses least-privilege configuration.

EQUIPMENT: Docker; approved base image registry (verify with C2DAO for current approved registry); container vulnerability scanner (Trivy or equivalent C2DAO-approved tool); container orchestration platform for runtime secret injection.

PROCEDURE: 1. Use a multi-stage Dockerfile: builder stage for dependency installation; production stage for runtime (see pattern below). 2. Create and configure a non-root `appuser` — do not run container as root. 3. Never include secrets in `ENV` instructions — inject at runtime via orchestration environment injection. 4. Implement a `HEALTHCHECK` instruction so the orchestrator can detect unhealthy containers. 5. Scan the built image for vulnerabilities before promotion to staging: `trivy image <image_name>`. 6. Resolve Critical and High CVEs before production deployment. Document Medium/Low with risk acceptance.

Dockerfile pattern for MSS services:

```
# Use approved Army base image — verify with C2DAO for current approved registry
FROM python:3.11-slim AS builder

# Build stage — installs dependencies, does not run application
WORKDIR /app

# Copy only dependency files first (Docker cache optimization)
COPY requirements.txt .
RUN pip install --no-cache-dir --prefix=/install -r requirements.txt

# --- Production stage ---
FROM python:3.11-slim AS production

# Security: run as non-root user
RUN groupadd -r appuser && useradd -r -g appuser appuser

WORKDIR /app

# Copy installed dependencies from builder stage
COPY --from=builder /install /usr/local

# Copy application code
COPY --chown=appuser:appuser src/ ./src/

# Switch to non-root before running
USER appuser

# No credentials in ENV — mount secrets at runtime via environment injection
# FOUNDRY_URL and FOUNDRY_SA_TOKEN provided by container orchestration, not here

# Health check — allows orchestrator to detect unhealthy containers
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD python -c "import src.health; src.health.check()" || exit 1

ENTRYPOINT ["python", "-m", "src.main"]
```

CAUTION: Never include FOUNDRY_SA_TOKEN, AWS_SECRET_ACCESS_KEY, or any other secret in a Dockerfile ENV instruction. These values are embedded in the image layer and visible to anyone with image access. Inject secrets at runtime via container orchestration environment injection.

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 7 — SECURITY ASSESSMENT AND SECURE CODE REVIEW

7-1. Scope of This Chapter

BLUF: SL 5L engineers are responsible for the security posture of MSS applications. This chapter covers OWASP-based secure code review in the Foundry context, secrets management, injection prevention, input validation, and authorized penetration testing patterns.

WARNING: Security assessment activities (penetration testing, vulnerability scanning, active probing) require written authorization from C2DAO and the USAREUR-AF ISSM before execution. Unauthorized security testing against MSS constitutes a computer security incident regardless of intent. Obtain authorization in writing before any assessment activity.

7-2. OWASP Top 10 in the Foundry Context

The OWASP Top 10 applies to all web-connected applications, including those built on Foundry. The following maps OWASP risks to MSS-specific patterns.

OWASP Risk	MSS Context	Mitigation
A01: Broken Access Control	CBAC bypass in application code; tenant data bleed	Use tenant-scoped service accounts (Chapter 4); never filter server-side without CBAC backing
A02: Cryptographic Failures	Credentials in environment files committed to repos; unencrypted transit	Secrets scanning in CI; TLS on all connections; mTLS for gRPC
A03: Injection	Ontology API name injection; Spark SQL injection in transforms	Whitelist API names; use parameterized queries; validate all inputs at boundary

OWASP Risk	MSS Context	Mitigation
A04: Insecure Design	Multi-tenant isolation via client-side filtering only	Enforce isolation at service account layer; defense-in-depth
A05: Security Misconfiguration	Open CORS on Workshop/OSDK-external apps (formerly Slate); permissive CBAC rules	Restrict CORS to MSS domain; minimal CBAC grants
A06: Vulnerable Components	Outdated OSDK versions with known CVEs	Automated dependency scanning in CI (pip-audit, npm audit)
A07: Authentication Failures	PATs in committed code; shared service accounts	detect-secrets in pre-commit; individual service accounts per service
A08: Software Integrity Failures	Unverified external data ingested into Ontology	Hash verification on file ingest; schema validation on all external records
A09: Logging Failures	Missing audit trails; insufficient error logging	Structured logging on all operations; audit log on all writes
A10: SSRF	External integrations that follow attacker-controlled URLs	Whitelist allowed external endpoints; never accept URLs from user input

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. Security controls (CORS, CBAC) must be reviewed and re-verified when migrating from Slate to Workshop or OSDK-external.

7-3. Injection Prevention

CONDITIONS: Your application accepts external input (API requests, Kafka messages, file uploads) that is used to query or modify the Foundry Ontology or datasets.

STANDARDS: All external input is validated against a strict whitelist before use. No user-supplied strings are used directly as Ontology API names, dataset RIDs, or query parameters without validation. SQL/SPARQL/Spark queries use parameterization.

EQUIPMENT: Python type annotations and `frozenset` whitelists; `re` module for format validation; Spark DataFrame API (not spark.sql for parameterized queries).

PROCEDURE: 1. Define `ALLOWED_OBJECT_TYPES` as a `frozenset` maintained by C2DAO — not user-configurable (see 7-3a). 2. Validate all user-supplied object type names, property names, and dataset RIDs against the whitelist before passing to OSDK. 3. For Spark queries, use the DataFrame API with

column references instead of `spark.sql()` with f-strings (see 7-3b). 4. Apply regex format validation on all string inputs used in queries (e.g., `unit_id` format check). 5. Validate and sanitize all Kafka, gRPC, and file upload message payloads at the boundary before Foundry write. 6. Include injection prevention checks in the SAST ruleset (see 7-5) — flag violations as blocking in CI.

7-3a. Ontology API Name Injection

A common injection vector is allowing user input to specify which Ontology Object Type or property to query.

```
# ALLOWED_OBJECT_TYPES: Maintained by C2DAO, not user-configurable
ALLOWED_OBJECT_TYPES = frozenset({
    "Unit",
    "Equipment",
    "Personnel",
    "Sitrep",
    "ReadinessReport",
})

def query_object_type(
    client: FoundryClient,
    object_type_name: str,  # From user input or external system
) -> list:
    """
    Query an Ontology Object Type by name.
    SECURITY: Validate object_type_name against whitelist before use.
    Do not pass user-supplied strings directly to the OSDK.
    """
    # Whitelist validation — reject anything not in the approved set
    if object_type_name not in ALLOWED_OBJECT_TYPES:
        raise ValueError(
            f"Object type '{object_type_name}' is not in the approved list. "
            f"Allowed types: {sorted(ALLOWED_OBJECT_TYPES)}"
        )

    # Safe to use after whitelist validation
    object_type = getattr(client.ontology.objects, object_type_name)
    return list(object_type.all())
```

7-3b. Spark SQL Injection Prevention

```
from pyspark.sql import DataFrame, SparkSession
import re

# BAD: String interpolation in Spark SQL — injection risk
def query_data_BAD(spark: SparkSession, unit_id: str) -> DataFrame:
    # If unit_id = "1-1-CAV' OR '1'='1", this returns all rows
    return spark.sql(f"SELECT * FROM units WHERE unit_id = '{unit_id}'")

# GOOD: Parameterized query using Spark DataFrame API — no injection
def query_data_GOOD(spark: SparkSession, unit_id: str) -> DataFrame:
    # Validate format before use
    if not re.match(r"^[A-Z0-9\-]{1,20}$", unit_id):
```

```
        raise ValueError(f"Invalid unit_id format: {unit_id!r}")

    return (
        spark.table("units")
        .filter(F.col("unit_id") == unit_id)  # Parameterized — safe
    )
```

7-4. Secrets Management

CONDITIONS: You are establishing secrets management standards for an MSS SWE team. Multiple services require Foundry credentials, AWS credentials, and integration API keys.

STANDARDS: No secrets in source code, committed files, Docker images, or logs. All secrets injected at runtime from approved secrets manager. Secret rotation is automated or procedurally enforced per AR 25-2. Secret access is audited.

EQUIPMENT: Approved secrets manager (HashiCorp Vault preferred; see hierarchy below); `hvac` Python library for Vault integration; `detect-secrets` for scanning; AR 25-2 reference.

PROCEDURE: 1. Select the appropriate secrets management tier for the deployment context (see hierarchy below). 2. For production services, configure Vault AppRole authentication using `VaultSecretsProvider` (see 7-4a). 3. Retrieve all secrets at service startup from the approved secrets manager — never hardcode. 4. Run `detect-secrets scan` as a pre-commit hook and in CI to prevent accidental secret commits. 5. Establish a secret rotation schedule per AR 25-2. Document rotation procedures in runbook. 6. Audit secret access logs quarterly. Report any anomalous access to C2DAO and unit S6/G6.

Secrets management hierarchy for MSS:

```
APPROVED SECRETS MANAGEMENT OPTIONS (in order of preference):

1. HashiCorp Vault (C2DAO-managed instance)
   - Full audit trail
   - Dynamic secret generation
   - Automated rotation
   - Preferred for production deployments

2. AWS Secrets Manager (for cloud-connected services)
   - IAM role-based access
   - Automated rotation support
   - Approved per C2DAO cloud guidance

3. Kubernetes Secrets (for containerized deployments)
   - Encrypted at rest (requires etcd encryption configured)
   - Injected as environment variables or mounted volumes
   - Acceptable for C2DAO-approved Kubernetes environments

4. Environment variables (for development / simple deployments)
   - Acceptable ONLY in development
   - Never use as sole secrets mechanism in production
```

```
NOT APPROVED:
  - .env files committed to repositories
  - Hardcoded values in any file
  - Secret values in Docker ENV instructions
  - Secret values in CI/CD pipeline environment variable UI (visible to all project
members)
```

7-4a. HashiCorp Vault Integration Pattern

```
import os
import hvac  # HashiCorp Vault client

class VaultSecretsProvider:
    """
    Retrieve MSS secrets from HashiCorp Vault.
    Authenticates using AppRole (recommended for services) or Token (development
only).
    """

    def __init__(self):
        vault_addr = os.environ["VAULT_ADDR"]  # e.g., https://vault.usareur.army.mil
        vault_token = os.environ.get("VAULT_TOKEN")
        vault_role_id = os.environ.get("VAULT_ROLE_ID")
        vault_secret_id = os.environ.get("VAULT_SECRET_ID")

        self._client = hvac.Client(url=vault_addr)

        if vault_role_id and vault_secret_id:
            # AppRole auth — preferred for production services
            self._client.auth.approle.login(
                role_id=vault_role_id,
                secret_id=vault_secret_id,
            )
        elif vault_token:
            # Token auth — development only
            self._client.token = vault_token
        else:
            raise EnvironmentError(
                "Vault authentication not configured. "
                "Set VAULT_ROLE_ID + VAULT_SECRET_ID (production) "
                "or VAULT_TOKEN (development only)."
            )

    def get_secret(self, path: str, key: str) -> str:
        """Retrieve a secret value from Vault KV store."""
        secret = self._client.secrets.kv.v2.read_secret_version(path=path)
        return secret["data"]["data"][key]

    def get_foundry_token(self) -> str:
        """Retrieve the MSS service account token."""
        return self.get_secret("mss/service-accounts", "foundry_sa_token")
```

7-5. Static Application Security Testing (SAST)

CONDITIONS: You are configuring automated SAST for MSS application repositories as part of the CI pipeline.

STANDARDS: SAST runs on every pull request. Critical and high findings block merge. Medium findings generate warnings that must be reviewed before merge. All findings are tracked in the security issue log.

EQUIPMENT: Semgrep (open source or Team tier); CI pipeline integration (Foundry Code Repositories or equivalent); security issue tracking system; access to MSS application repositories.

Semgrep configuration for MSS Python projects:

```
# .semgrep/mss-rules.yaml
# MSS-specific SAST rules for Foundry application security

rules:
  - id: foundry-token-in-code
    pattern: |
      $X = "ri...."
    message: |
      Potential Foundry RID or token hardcoded in source. Move to environment
      variable or Vault. Foundry RIDs are not secrets but tokens are.
    severity: WARNING
    languages: [python, typescript]

  - id: no-direct-string-interpolation-in-sparksql
    pattern: |
      spark.sql(f"... $X ...")
    message: |
      f-string interpolation in spark.sql() is an injection risk.
      Use DataFrame API (filter, where) with column references instead.
    severity: ERROR
    languages: [python]

  - id: no-admin-client-in-tenant-functions
    pattern: |
      def $FUNC(...):
        ...
        admin_client.$X(...)
    message: |
      Using admin_client inside a tenant-scoped function may expose cross-tenant data.
      Use tenant-scoped client (TenantRouter) for all tenant data access.
    severity: WARNING
    languages: [python]

  - id: classification-marking-required
    pattern: |
      def $FUNC(...) -> dict:
        ...
        return {...}
    message: |
      Functions returning data dicts should include classification_marking field.
      Verify all data products carry correct marking before return.
```

```
    severity: INFO
    languages: [python]
```

7-6. Authorized Penetration Testing Patterns

CONDITIONS: You have written authorization from C2DAO and USAREUR-AF ISSM to conduct authorized security testing against MSS application components.

STANDARDS: Testing is bounded by the authorization scope. No testing against production without explicit written authorization scope. All findings documented and reported to C2DAO within 24 hours. No exploited vulnerabilities remain unmitigated after testing.

EQUIPMENT: Written C2DAO and ISSM authorization document; Python `pytest` with security markers; development MSS environment (never production without explicit scope); findings report template.

WARNING: The following section describes authorized security testing patterns. Executing these patterns without written authorization is a computer fraud offense under 18 U.S.C. § 1030 and a violation of AR 25-2. Obtain written authorization before executing any security test.

Authorized test categories for MSS applications:

Test Category	What It Tests	Required Authorization Level
CBAC boundary testing	Verify service accounts cannot access data outside their CBAC grant	C2DAO written authorization
OSDK input validation	Verify OSDK rejects malformed queries without leaking error details	C2DAO written authorization
Token exposure scan	Verify no tokens in repos, images, or environment dumps	C2DAO written authorization
Tenant isolation validation	Verify tenant A cannot read tenant B data through any code path	C2DAO + ISSM written authorization
External integration fuzzing	Fuzz external API endpoints for injection and parsing vulnerabilities	C2DAO + ISSM + system owner authorization
Authentication bypass testing	Verify authentication cannot be bypassed on all endpoints	C2DAO + ISSM + system owner authorization

CBAC boundary test pattern:

```
# Authorized CBAC boundary test
# Run ONLY against development MSS with written C2DAO authorization

import pytest

@pytest.mark.security
@pytest.mark.requires_authorization  # Marker — CI skips unless AUTH_TEST=1
```

```
class TestCBACBoundaries:
    """
    Verify CBAC isolation between tenant service accounts.
    Run only with explicit C2DAO authorization (document auth ref in test session
log).
    """

    def test_vcorps_sa_cannot_read_tsc_objects(
        self,
        vcorps_client,   # Service account scoped to V Corps
        tsc_object_rid,  # A known 21st TSC object RID
    ):
        """
        V Corps service account must not be able to fetch a 21st TSC object.
        Expected: PermissionDenied or empty result.
        Failure: If TSC object is returned, CBAC is misconfigured — security incident.
        """
        try:
            result = vcorps_client.ontology.objects.get(tsc_object_rid)
            # If we reach here without exception, CBAC is failing
            assert result is None, (
                f"SECURITY FINDING: V Corps SA retrieved TSC object {tsc_object_rid}.
"
                "CBAC boundary is not enforced. Report to C2DAO immediately."
            )
        except Exception as exc:
            # Expected — permission denied is the correct behavior
            assert "permission" in str(exc).lower() or "not found" in
str(exc).lower(), (
                f"Unexpected error type: {exc}. Investigate before concluding CBAC is
working."
            )
```

CHAPTER 8 — PLATFORM LEADERSHIP AND ENGINEERING STANDARDS

8-1. Scope of This Chapter

BLUF: SL 5L engineers are not just senior individual contributors. They lead the MSS SWE community — defining standards, conducting architecture reviews, onboarding developers, managing technical debt, and maintaining the platform's engineering health. This chapter covers the leadership responsibilities of the SL 5L role.

8-2. Architecture Review

CONDITIONS: A SL 4L developer or team proposes a new MSS application, integration, or significant change to an existing system.

STANDARDS: All new MSS applications and major changes undergo architecture review before development begins. Architecture review is documented. Findings are resolved before development proceeds. Review covers security, performance, multi-tenancy, and maintainability.

EQUIPMENT: Architecture Review Record template (see below); access to system design documentation from the proposing developer; C2DAO coordination for ATO impact assessment.

Architecture review checklist:

Domain	Questions to Answer
Security	What data does this system access? What is the CBAC model? How are credentials managed? What is the attack surface? Has threat modeling been done?
Multi-tenancy	Does this system serve multiple tenants? How is isolation enforced — CBAC, service accounts, code? What happens if isolation fails?
Performance	What is the expected query volume? Are there N+1 risks? What is the caching strategy? What is the performance SLA? How was it validated?
External integrations	What external systems are connected? How are failures handled? Is the integration idempotent? Has C2DAO authorized the integration?
Maintainability	Can a new SL 4L developer understand and maintain this without the original author? Are abstractions well-chosen? Is the test coverage plan realistic?
Operability	How will failures be detected? What monitoring and alerting is in place? What is the runbook for common failure modes?
ATO impact	Does this change affect the ATO boundary? Does it introduce new data flows, new integrations, or new access patterns that require ATO amendment?

Architecture review output:

```
ARCHITECTURE REVIEW RECORD

System Name: [NAME]
Review Date: [DATE]
Reviewer(s): [NAMES + TM level]
Developer(s): [NAMES]
Ticket: [CHANGE MANAGEMENT REFERENCE]

SYSTEM DESCRIPTION:
[2-3 paragraph description of the system, its purpose, and its place in the MSS
ecosystem]

FINDINGS:
```

```
Critical (must resolve before development):
1. [FINDING — RESOLUTION REQUIRED]

Major (must resolve before production):
1. [FINDING — RESOLUTION REQUIRED]

Minor (resolve before code review completion):
1. [FINDING — RECOMMENDATION]

APPROVED FOR DEVELOPMENT: [YES / NO / CONDITIONAL]
CONDITIONS: [If conditional — what must be resolved]

Reviewer Signature: _____ Date: _____
C2DAO Coordination: [TICKET ID or N/A]
```

8-3. Code Review Standards

CONDITIONS: You are conducting code review for a SL 4L developer's pull request.

STANDARDS: All code merged to staging or production branches has been reviewed by a SL 5L engineer. Review is substantive — not a rubber stamp. Security findings are blocking. Performance findings are blocking if exceeding defined thresholds. Findings are documented, not just verbally communicated.

EQUIPMENT: Repository pull request interface (Foundry Code Repositories or equivalent); Appendix A code review checklist; access to the application's architecture documentation and test results.

Code review approach:

1. Read the PR description and linked ticket. Understand what the change is trying to accomplish before reading code.
2. Review architecture-level concerns first: does the overall approach fit MSS patterns?
3. Review security: run through Appendix A checklist for every PR.
4. Review performance: check for N+1 patterns, missing pagination, unguarded full scans.
5. Review maintainability: will the next engineer understand this without the author present?
6. Leave specific, actionable comments. "This looks fine" is not a code review.

Blocking vs. non-blocking findings:

Finding Type	Action
Security vulnerability (any severity)	Blocking — must fix before merge
Missing input validation	Blocking
Hardcoded credential or secret	Blocking — do not merge under any circumstances
N+1 query in production code path	Blocking

Finding Type	Action
Missing test coverage for new logic	Blocking if coverage drops below 80%
Performance concern (not confirmed issue)	Non-blocking — document, developer acknowledges
Code style / readability	Non-blocking — suggest, do not mandate unless violates team standard
Missing documentation for public function	Non-blocking — flag, but does not block

8-4. Developer Onboarding

CONDITIONS: A new SL 4L engineer joins the MSS SWE team. You are responsible for onboarding them to team standards and the MSS environment.

STANDARDS: New engineers can independently build and deploy an MSS application within 30 days of onboarding. Onboarding covers security, tooling, platform access, and team standards. Onboarding is documented — not ad hoc.

EQUIPMENT: Onboarding checklist (see below); C2DAO credentialing request form; development MSS environment access; team repository access; pre-commit hook configuration.

MSS SWE onboarding sequence:

Week	Activities	Validation
Week 1	SL 4L review (if not current); C2DAO credentialing; development environment setup; repository access; pre-commit hook installation	Can connect to dev MSS, run test suite
Week 2	Codebase walkthrough with SL 5L mentor; first PR (small change) reviewed by SL 5L	First merged PR
Week 3	Assigned first independent task with SL 5L review; security training (Appendix A checklist walkthrough)	Task completed, security checklist reviewed
Week 4	Architecture review participation (as observer); deployment to staging via CI pipeline	Attended architecture review; deployed to staging independently

8-5. Technical Debt Management

MSS applications accrue technical debt. Left unmanaged, debt degrades platform reliability and makes future development harder. SL 5L engineers are responsible for tracking and managing debt on systems they own.

Technical debt categories:

Category	Examples	Priority
Security debt	Outdated dependency with known CVE; missing input validation; weak authentication	P0 — fix immediately
Performance debt	Known N+1 queries; unindexed frequently-queried properties; unbounded result sets	P1 — fix in current quarter
Maintainability debt	Undocumented complex logic; functions over 200 lines; missing tests for critical paths	P2 — fix in next two quarters
Architectural debt	Patterns that bypass platform standards; workarounds for missing platform features	P3 — address when platform capability available

Debt tracking format:

TECHNICAL DEBT REGISTER

System: [NAME] | **Last updated:** [DATE] | **Owner:** [SL 5L engineer]

ID	Category	Description	Impact	Est. Effort	Target Qtr	Status
TD-001	Security	pip-audit reports CVE-2024-34069 in werkzeug<3.0.6 (debugger RCE)	Medium	0.5 days	26Q2	Open
TD-002	Performance	Unit search endpoint does full scan on each call	High	2 days	26Q2	In Progress

8-6. Platform Governance

SL 5L engineers participate in MSS platform governance — the process by which the SWE community, C2DAO, and stakeholders make decisions about platform direction, standards, and investments.

Platform governance responsibilities:

Activity	Frequency	SL 5L Role
SWE community sync	Monthly	Present, vote on standards changes, raise architectural concerns
C2DAO coordination	As needed	Represent SWE requirements; provide technical input on CBAC and infrastructure decisions
Platform roadmap review	Quarterly	Review planned Foundry platform updates for impact on MSS applications

Activity	Frequency	SL 5L Role
ATO annual review	Annual	Provide technical artifacts, respond to ISSM findings, document control implementations
Incident post-mortems	After each P1/P2 incident	Lead technical investigation; draft post-mortem; track remediation

8-7. Managing Platform Upgrades

CONDITIONS: Palantir releases a new version of Foundry with breaking changes to the OSDK or Platform SDK. You must manage the upgrade across all MSS applications.

STANDARDS: Upgrade impact is assessed before any application code changes. All affected applications are inventoried. Upgrade is tested in development, then staging, before production. Applications are not left on deprecated API versions.

EQUIPMENT: Palantir upgrade release notes (from C2DAO / Palantir support channel); application inventory (all MSS applications under SWE team ownership); development and staging MSS environments; C2DAO coordination for production promotion.

Upgrade management procedure:

1. Obtain upgrade release notes from C2DAO / Palantir support channel.
2. Assess breaking changes against the application inventory.
3. For each affected application, create a migration ticket with estimated effort.
4. Prioritize applications by operational criticality — mission-critical apps are upgraded last to minimize risk window.
5. Execute upgrades in development environment, run full test suite.
6. Promote to staging, run integration tests.
7. Coordinate production promotion with C2DAO and operational stakeholders.
8. Monitor for 48 hours post-upgrade before closing migration ticket.

NOTE

Never run MSS production applications on deprecated OSDK API versions beyond the Palantir-communicated end-of-support date. Deprecated APIs may stop working without notice during platform upgrades. Track deprecation dates and plan migrations proactively.

APPENDIX A — CODE REVIEW CHECKLIST (USAREUR-AF STANDARDS)

Use this checklist for every pull request review on MSS application code.

A-1. Security Checklist

Credentials and Secrets - ☐ No hardcoded credentials, tokens, API keys, or secrets in any file - ☐ No credentials in environment variable defaults (e.g., `token = os.getenv("TOKEN", "actual_token_here")`) - ☐ `.env` file present in `.gitignore` - ☐ `detect-secrets` baseline is current and no new secrets detected - ☐ All credential references use environment variables or approved secrets manager

Authentication and Authorization - ☐ OSDK and Platform SDK clients use service accounts (not PATs) in production code - ☐ Service account is least-privilege for the function being performed - ☐ Tenant isolation uses scoped service accounts, not code-level filtering - ☐ No code path allows a tenant context to be overridden by user input

Input Validation - ☐ All external inputs validated at system boundary before use - ☐ No user-supplied strings used directly as Ontology API names without whitelist check - ☐ No string interpolation in Spark SQL queries — DataFrame API used - ☐ File uploads: hash verification implemented for files from external systems - ☐ gRPC/Kafka/Kinesis messages: schema validation before Foundry write

Classification and Markings - ☐ Data products carry correct classification markings - ☐ No code strips or suppresses markings - ☐ Coalition-facing data paths implement releasability filter

A-2. Performance Checklist

Query Patterns - ☐ No N+1 query patterns in production code paths - ☐ Filters pushed to OSDK query layer (not Python-side filtering of full result sets) - ☐ Property projection used where only subset of properties needed - ☐ Pagination implemented for all queries that may return large result sets

Caching - ☐ Cache TTLs appropriate to data freshness requirements - ☐ Cache is not shared across authorization boundaries (per-tenant or per-service-account scoping) - ☐ Cache invalidation implemented for write paths

Compute - ☐ No unbounded loops over full object populations without pagination - ☐ Large analytical queries use pre-aggregated datasets, not on-demand aggregation - ☐ Transform schedule appropriate to source data update frequency

A-3. Maintainability Checklist

Code Quality - ☐ Functions have single responsibility and are under 50 lines (flag for discussion if larger) - ☐ Public functions and classes have docstrings - ☐ Non-obvious logic has inline comments explaining "why", not "what" - ☐ No magic numbers or strings — constants defined with descriptive names

Testing - ☐ Unit tests cover new logic - ☐ Coverage does not drop below 80% team threshold - ☐ Tests are deterministic — no time-dependent or network-dependent assertions in unit tests - ☐ Integration tests cover external integration points

Configuration - ☐ No hardcoded URLs, RIDs, or environment-specific values in code - ☐ All configuration uses environment variables or config files - ☐ Configuration schema documented (Pydantic Settings or equivalent)

A-4. Operational Readiness Checklist

Observability - ☐ Structured logging (JSON format preferred) on all significant operations - ☐ Error logging includes enough context to diagnose without source code access - ☐ Audit logging on all dataset writes and Ontology mutations

Resilience - ☐ External integrations implement retry with backoff - ☐ Circuit breaker pattern for integrations that can fail - ☐ Dead-letter handling for failed message deliveries - ☐ Graceful degradation defined for downstream dependency failures

Deployment - ☐ Dockerfile uses non-root user - ☐ No secrets in Docker image layers - ☐ Health check endpoint implemented - ☐ Container resource limits defined

APPENDIX B — ATO SUPPORT DOCUMENTATION GUIDE

B-1. Overview

The Authority to Operate (ATO) process under RMF (NIST SP 800-37) requires technical documentation of security controls for MSS applications. SL 5L engineers are primary contributors to ATO packages for systems they build and own. This appendix describes the artifacts SL 5L engineers are responsible for producing.

NOTE

The ISSM and C2DAO lead the ATO process. SL 5L engineers do not own the ATO — they provide technical evidence and documentation. Coordinate with the ISSM before beginning ATO documentation work to ensure artifacts meet current RMF requirements.

B-2. SL 5L-Owned ATO Artifacts

Artifact	Description	RMF Control Families
System Architecture Diagram	Dataflow diagram showing all MSS application components, external connections, data stores, and authentication points	AC, IA, SC
Data Flow Diagram	How data enters, moves through, and exits the system. Classification levels at each stage.	AC, AU, SC
Authentication and Authorization Description	How users and services authenticate; how access decisions are made; CBAC model	AC, IA
Secrets Management Description	Where secrets are stored; how they are accessed; rotation procedures	IA, SC
Boundary Protection Description	Network boundaries; what traffic is allowed in/out; how external integrations are controlled	SC
Audit Logging Description	What is logged; where logs are stored; how long retained; who can access	AU
Incident Response Procedures	How security incidents are detected and reported; roles and responsibilities	IR
Vulnerability Management Procedures	How dependencies are scanned; how findings are tracked and remediated	SI, RA
CI/CD Security Controls Description	How code is tested and reviewed before deployment; how secrets are managed in pipeline	SA, CM
SAST Configuration and Findings	Evidence that SAST is configured and findings are addressed	SA, SI

B-3. System Architecture Diagram Requirements

The system architecture diagram must include:

1. All application components (services, containers, databases, message queues)
2. All MSS/Foundry integration points (OSDK, Platform SDK, direct dataset access)

3. All external system connections (labeled with protocol, port, and direction)
4. Authentication boundaries (where auth decisions are made)
5. Data classification levels at each storage and transit point
6. Network boundaries / security zones (MSS production, USAREUR network, coalition MPE, internet)

Diagram format: Use a standard notation (e.g., C4 Model, UML deployment diagram, or DoD Architecture Framework view). Diagrams must be maintained — update with each significant architecture change.

B-4. Data Flow Diagram Requirements

For each significant data flow:

1. Data source (system name, classification)
2. Transit path (protocol, encryption in transit)
3. Processing steps (transformations, filtering, aggregation)
4. Data store (system name, classification, encryption at rest)
5. Data consumers (who/what reads this data, authorization requirements)
6. Retention and deletion policy

B-5. Continuous Monitoring Contributions

ATO is not a one-time event — it requires continuous monitoring. SL 5L engineers support continuous monitoring by:

- Running automated dependency vulnerability scans (pip-audit, npm audit) monthly. Document findings. Remediate within SLA (Critical: 30 days, High: 60 days, Medium: 90 days).
- Running SAST on every code change (CI pipeline). Document unresolved findings with risk acceptance.
- Reviewing audit logs quarterly for anomalous access patterns. Coordinate with C2DAO ISSM.
- Updating ATO artifacts when system architecture changes. Notify ISSM of significant changes.
- Participating in annual ATO renewal review. Provide updated artifacts and respond to ISSM questions.

B-6. ATO Documentation Version Control

All ATO artifacts must be version-controlled. Store in the application code repository under `docs/ato/`. Each artifact includes:

```
Document: [ARTIFACT NAME]
System: [SYSTEM NAME]
Version: [MAJOR.MINOR]
Last Updated: [DATE]
Author: [NAME, ROLE]
```

```
Approved By: [ISSM NAME, DATE]
Next Review: [DATE]

CHANGE LOG:
v1.1 - [DATE] - [DESCRIPTION OF CHANGE]
v1.0 - [DATE] - Initial version
```

GLOSSARY

AOR — Area of Responsibility. The geographic region assigned to a commander.

ATO — Authority to Operate. Formal authorization from an Authorizing Official (AO) granting a system permission to operate in a DoD environment under RMF.

ASVS — Application Security Verification Standard. OWASP framework defining security requirements for web applications.

C2DAO — Command and Control Data Architecture Office. USAREUR-AF organization responsible for MSS architecture, governance, CBAC schema, and integration standards.

CBAC — Content-Based Access Control. Foundry's access control mechanism that restricts data visibility based on markings, properties, and user attributes — not just dataset-level permissions.

CI/CD — Continuous Integration / Continuous Deployment. Automated pipeline for building, testing, and deploying software.

Circuit Breaker — Design pattern that stops making calls to a failing dependency when a failure threshold is exceeded. Prevents cascading failures in distributed systems.

CUI — Controlled Unclassified Information. U.S. Government information that requires safeguarding per law, regulation, or policy, but is not classified.

Dead Letter — A message or payload that could not be successfully processed after all retry attempts. Stored separately for manual review and recovery.

DevSecOps — Development, Security, and Operations. Practice that integrates security into every phase of the software development lifecycle.

DoD PKI — Department of Defense Public Key Infrastructure. The system that issues digital certificates (CAC) used for authentication.

FOO — Functions on Objects. TypeScript functions in Foundry that compute derived properties or execute logic against Ontology objects.

gRPC — Google Remote Procedure Call. High-performance, protocol-buffer-based RPC framework. Supports bidirectional streaming.

IAM — Identity and Access Management. AWS service (and general concept) for managing authentication and authorization.

Idempotent — An operation that produces the same result regardless of how many times it is executed. Required property for all retry-capable operations.

ISSM — Information System Security Manager. Responsible for maintaining the security posture and ATO of Army information systems.

Kafka — Apache Kafka. Distributed event streaming platform. Used for high-throughput, fault-tolerant event ingestion.

Kinesis — AWS Kinesis Data Streams. Managed event streaming service in AWS. Similar to Kafka in operational role.

mTLS — Mutual TLS. Both client and server present certificates for authentication — not just the server. Required for gRPC integrations in USAREUR-AF.

MPE — Multinational Participation Environment. The infrastructure and data architecture that enables data sharing with coalition partners under information sharing agreements.

MSS — Maven Smart System. The USAREUR-AF enterprise AI/data platform, built on Palantir Foundry.

N+1 Query — Performance anti-pattern where code fetches N objects and then executes one additional query per object to retrieve linked data, resulting in N+1 total queries.

Ontology — In Foundry, the semantic layer that defines Object Types, Link Types, and Actions — the business model of operational data.

OSDK — Ontology SDK. The primary API for external applications to query and interact with the Foundry Ontology.

OWASP — Open Worldwide Application Security Project. Produces the OWASP Top 10 list of critical web application security risks.

Platform SDK — Foundry Python SDK for infrastructure-level operations: reading and writing datasets, managing transactions, accessing file resources.

RMF — Risk Management Framework. NIST SP 800-37 process for authorizing DoD information systems, including ATO.

SAST — Static Application Security Testing. Automated analysis of source code for security vulnerabilities without executing the code.

SLA — Service Level Agreement. Defined performance and availability targets for a service.

Tenant — A logical grouping of users and data within MSS with defined isolation boundaries. Example tenants: III Corps, V Corps, 21st TSC, 10th AAMDC, 56th MDC-E, SETAF-AF, coalition.

TTL — Time to Live. The duration for which a cached value is considered valid before it is refreshed.

UDRA — Unified Data Reference Architecture. Version 1.1 (February 2025). DoD/Army framework defining canonical data flows, domain ownership, and integration standards.

USAREUR-AF — United States Army Europe and Africa. The Army Service Component Command to USEUCOM and USAFRICOM, headquartered in Wiesbaden, Germany.

USEUCOM — United States European Command. The Combatant Command covering the European AOR. USAREUR-AF serves as ASCC to both USEUCOM and USAFRICOM.

Vault — HashiCorp Vault. Secrets management platform for storing and accessing credentials, tokens, and certificates.

SL 5L — ADVANCED SOFTWARE ENGINEERING — MAVEN SMART SYSTEM HEADQUARTERS, UNITED STATES ARMY EUROPE AND AFRICA Wiesbaden, Germany — 2026 Distribution: Distribution authorized to U.S. Government agencies and their contractors only. Other requests must be referred to Headquarters, USAREUR-AF, C2DAO, Wiesbaden, Germany.

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