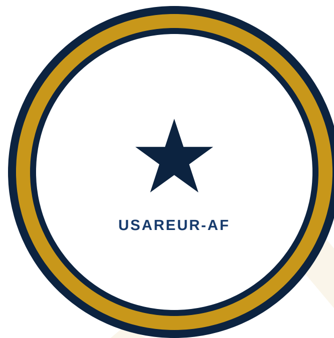
DRAFT — UNOFFICIAL — NOT FOR OPERATIONAL USE

TECHNICAL MANUAL

SL 4M



TM-40M — MAVEN SMART SYSTEM (MSS)

Specialist Course Manual

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

DRAFT — NOT FOR OFFICIAL USE. FOR TRAINING PLANNING PURPOSES ONLY.

26 MARCH 2026

DRAFT — UNOFFICIAL — NOT FOR OPERATIONAL USE

TM-40M — MAVEN SMART SYSTEM (MSS)

Forward: SL 4M qualifies machine learning engineers to build, train, evaluate, deploy, and maintain ML models on the Maven Smart System (MSS) within USAREUR-AF. This manual covers the full ML lifecycle — from Code Workspace setup through production model governance. **Prereqs:** SL 1, Maven User; SL 2, Builder; SL 3, Advanced Builder; Data Literacy Technical Reference (required). Proficiency in Python, scikit-learn, PyTorch or statsmodels, and SQL is assumed. Readers who cannot independently write a training loop, build a feature pipeline, and interpret a confusion matrix should complete prerequisite training before beginning this manual; CONCEPTS_GUIDE_TM40M_ML_ENGINEER (read before this manual). *HQ USAREUR-AF · v1.0 · 2026 · DISTRIB: USG only · AUTH: C2DAO/UDRA v1.1*

WARNING: MODEL OUTPUTS REQUIRE HUMAN VALIDATION BEFORE OPERATIONAL USE. MSS ML models are decision-support tools. Automated model outputs that trigger Actions, update Ontology properties, or feed into operational dashboards without human review in the loop are not authorized without explicit C2DAO architecture review and documented approval. Failure to comply with this requirement may result in incorrect operational decisions based on model error. The machine learning engineer is responsible for ensuring this constraint is enforced in every deployment. WARNING: TRAINING ON OPERATIONAL DATA. Do not export operational data from Foundry to external compute environments (personal laptops, commercial cloud, unauthorized HPC systems) for model training. All training on MSS data must occur within the authorized Code Workspace environment. Violations may constitute a data spillage incident.

WARNING: HUMAN-MACHINE TEAMING (ADP 3-13). ADP 3-13 states AI/ML systems enable speed; humans provide judgment. No ML model output replaces commander decision authority. All automated recommendations must include confidence scores and the data basis for the recommendation. MLEs will design every model output to support — never supplant — the human decision-maker.

CHAPTER 1 — INTRODUCTION: THE MLE ROLE ON MSS

1-1. ML Engineer Specialist Manual

BLUF: SL 4M qualifies machine learning engineers to build, train, evaluate, deploy, and maintain ML models on the Maven Smart System (MSS) within USAREUR-AF. This manual covers the full ML lifecycle — from Code Workspace setup through production model governance.

This manual provides task-based instruction for machine learning engineers (MLEs) operating on MSS. MSS is the USAREUR-AF enterprise AI/data platform built on Palantir Foundry. SL 4M qualified personnel design and implement predictive models, feature pipelines, and ML-backed data products that support operational decision-making across III Corps, V Corps, 21st TSC, 10th AAMDC, 56th MDC-E, SETAF-AF, USAREUR-AF G2, and subordinate commands.

SL 4M covers setting up and using Code Workspaces (JupyterLab/RStudio) on Foundry; reading Foundry datasets into notebooks and managing Python environments; building feature engineering pipelines that feed ML training workflows; training models using scikit-learn, PyTorch, statsmodels, and similar libraries; training models using Model Studio (Palantir's no-code model training workspace, GA February 2026); evaluating model performance with operationally appropriate metrics; conducting bias and fairness checks for models affecting personnel or readiness assessments; publishing trained models to the Ontology and implementing model-backed Object properties; implementing MLOps patterns: versioning, experiment tracking, retraining triggers, drift detection; building and maintaining feature pipelines using Foundry Transforms; applying ML to operational use cases: readiness prediction, logistics demand forecasting, anomaly detection in OPDATA; and completing model governance documentation and navigating the approval process before production deployment.

SL 4M does NOT cover basic Python or data science fundamentals — see data_skills curriculum; no-code pipeline building — see SL 2 and SL 3 (no-code model training via Model Studio IS covered in Chapter 10); Workshop application design — see SL 3; Ontology design methodology — see SL 3 (consumed by MLE, not designed here); TypeScript Functions on Objects or OSDK application development — see SL 4L (Software Engineer); large language model fine-tuning or Agent Studio development — see SL 4H (AI Engineer); or operations research / statistical modeling for optimization — see SL 4G (ORSA).

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

> **NOTE**
>
> The Army established the **49B AI/ML Officer Career Path** in 2025–26, creating the first dedicated uniformed career track for AI/ML expertise. SL 4M directly aligns to 49B qualification requirements. ML Engineers completing this course and SL 5M (Advanced) are positioned for assignment to 49B-coded billets across the force.

Prerequisite: SL 3 (Advanced Builder) is REQUIRED. Python proficiency (scikit-learn, PyTorch or statsmodels, SQL) is required independently of the TM series.

Advanced track: Upon completing SL 4M, qualified MLEs should pursue **SL 5M (Advanced ML Engineer)** for advanced topics including neural architecture selection, large-scale feature store design, advanced MLOps patterns, and federated learning considerations for coalition data environments.

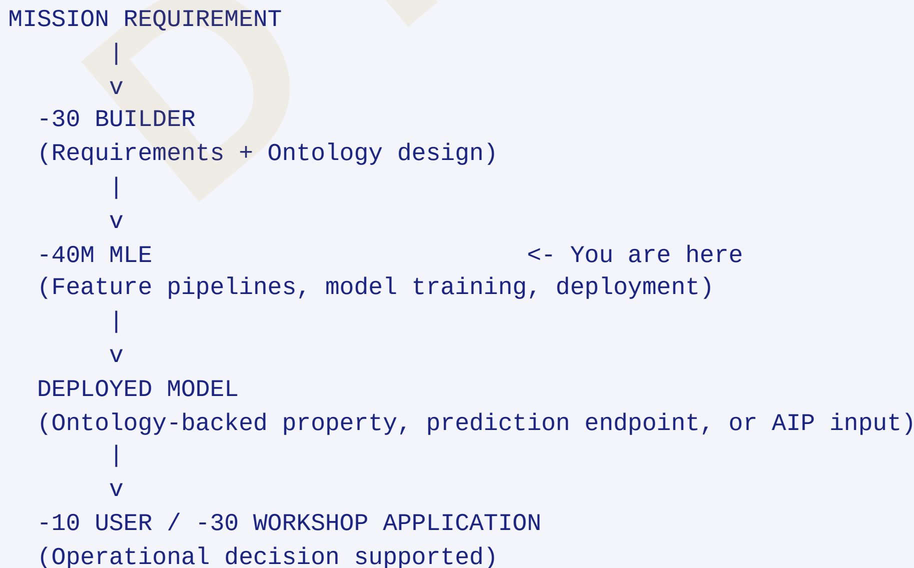
Peer specialist tracks: The ML Engineer coordinates closely with the AI Engineer (SL 4H). The MLE builds and validates the model artifact; the AI Engineer integrates that model into AIP Logic orchestration workflows and human-review-gated operational products. Neither role replaces the other — coordinate at the model/workflow interface. Coordinate with SL 4G (ORSA) when a commander's analytical question requires both a predictive model and a decision-grade quantitative product. Coordinate with SL 4L (Software Engineer) for production pipeline implementation when model complexity exceeds standard Foundry Transforms.

WFF awareness: ML models deployed on MSS generate Ontology-backed properties that appear on objects visible to WFF-qualified users (SL 4A through SL 4F — Intelligence, Fires, Movement and Maneuver, Sustainment, Protection, and Mission Command). A readiness prediction score attached to a vehicle is seen by Sustainment (SL 4D) staff; a threat assessment feature is seen by Intelligence (SL 4A) staff. Design model outputs and their presentation with WFF end-users in mind: operational terminology, asymmetric error cost awareness, and explainability appropriate to the WFF function.

1-3. The MLE in the USAREUR-AF Data Chain

The MLE operates at the deepest technical level of the data chain, translating curated Foundry datasets into deployed predictive models. The MLE is responsible for the full lifecycle: feature pipeline through production monitoring. Unlike a general software engineer, the MLE must hold simultaneous fluency in the operational domain (what the model is predicting and why it matters), the statistical method (whether the model is valid), and the platform (how Foundry executes the workflow).

MLE position in the USAREUR-AF data chain:



```
MISSION REQUIREMENT
       |
       v
  -30 BUILDER
  (Requirements + Ontology design)
       |
       v
  -40M MLE                          <- You are here
  (Feature pipelines, model training, deployment)
       |
       v
  DEPLOYED MODEL
  (Ontology-backed property, prediction endpoint, or AIP input)
       |
       v
  -10 USER / -30 WORKSHOP APPLICATION
  (Operational decision supported)
```

DRAFT

The MLE is not the consumer of operational decisions — the MLE is the engineer of the system that informs those decisions. This distinction carries significant responsibility: a model that silently degrades in production, a feature pipeline that drops records without alerting, or a deployment that bypasses governance gates are MLE failures with operational consequences.

1-4. MLE Role Boundaries

Responsibility	MLE Owns	Coordinate With
Feature pipeline code	MLE authors and maintains	Data steward for source schema changes
Model training and selection	MLE owns	N/A for approach; C2DAO for use case authorization
Evaluation methodology	MLE designs and documents	Mission owner reviews operational thresholds
Deployment to Ontology	MLE implements	-30 builder for Object Type design; C2DAO for production gate
Retraining triggers	MLE codes	Data steward for data freshness SLAs
Drift monitoring	MLE monitors and responds	Data steward + mission owner for degradation thresholds
Model governance documentation	MLE authors	C2DAO reviews and approves
Bias/fairness assessment	MLE conducts	Mission owner accepts residual risk
Incident response	MLE leads technical response	C2DAO notified within 24 hours of production incident

1-5. Prerequisites

Verify the following prerequisites before beginning this manual. These are not recommendations — they are entry conditions.

Prerequisite	Verification Method
SL 3 completion	Certification on file
Python proficiency (intermediate+)	Can write a training loop, build a pipeline, debug a stack trace
scikit-learn or PyTorch working knowledge	Can train, serialize, and load a model

Prerequisite	Verification Method
SQL proficiency	Can write JOIN, GROUP BY, window functions
Foundry dataset and ontology familiarity	Can navigate, filter, and read a Foundry dataset
Git proficiency	Can branch, commit, push, open a pull request

If you cannot independently complete the prerequisite items above, stop. Access `learn-data.armydev.com` for prerequisite training resources before continuing.

1-6. Governing References

Document	Relevance
USAREUR-AF C2DAO Guidance	Theater-level architecture standards; ML model approval authority
DoD AI Ethics Principles (2020)	Responsible AI: responsible, equitable, traceable, reliable, governable
Army AI/ML Framework (current)	Army-level guidance on ML model lifecycle and oversight
NATO Architecture Framework v4 (NAFv4)	Coalition data architecture standards for MPE-accessible model outputs
learn-data.armydev.com	Authoritative reference implementations and approved patterns
Army DIR 2024-03	Digital Engineering Policy — Army-wide digital engineering adoption directive
ADP 3-13	Information Advantage — human-machine teaming principles; AI enables speed, humans provide judgment
FM 3-12	Cyberspace Operations and Electromagnetic Warfare — operational context for ML systems
DA PAM 25-2-5	Software Assurance — software security standards

1-6a. 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 governance hierarchy, stewardship responsibilities, data product standards
UDRA v1.1 (February 2025)	Army Enterprise	Domain ownership, federated governance, ML model classification requirements
DoD Data Strategy (2020)	OSD	VAULTIS-A framework (supersedes VAUTI) applicable to ML model outputs as data products — 8 dimensions per DDOF Playbook v2.2 (Dec 2025); 85% weighted avg = DDOF Phase 3 quality gate

1-7. Model Governance Requirement Overview

All MSS ML models are data products subject to UDRA v1.1 governance. The following gates are mandatory. Chapter 9 covers each in detail. This overview establishes expectations from the outset.

Gate	Timing	Authority
Use Case Authorization	Before any development work begins	C2DAO
Design Review	Before model training on production data	C2DAO + data steward
Evaluation Acceptance	Before any deployment activity	Mission owner
Bias/Fairness Review	Before any deployment activity (personnel models)	Mission owner + G1 coordination
Production Deployment Approval	Before model writes to production Ontology	C2DAO
Post-Deployment Monitoring Plan	Within 30 days of deployment	MLE + data steward

NOTE

Appendix B contains the list of pre-authorized use case categories for USAREUR-AF. If your use case is not on this list, start with the C2DAO Use Case Authorization process before writing any code.

DRAFT

1-8. DDOF Phases 4–5 for ML

BLUF: ML models on MSS follow the Data and Digital Object Foundry (DDOF) lifecycle. Phases 4 (Development) and 5 (Test & Evaluation) define the engineering and validation gates every model must clear before production deployment. The MVP mandate is 30 days from Phase 4 entry to Phase 5 exit.

ML development on Foundry is not unconstrained experimentation. The DDOF lifecycle imposes structure: each phase has a defined entry condition, required activity, and gate output. Phases 4 and 5 are where the MLE spends the majority of hands-on effort. Failure to produce the required gate output blocks progression — there are no waivers for incomplete documentation.

DDOF Phase	ML Activity	Gate Output
Phase 4 (Development)	Feature engineering, model training, hyperparameter tuning	Functional model with documented architecture
Phase 5 (T&E)	Validation on holdout data, bias testing, adversarial testing, UAT	Test report with accuracy/precision/recall, sponsor sign-off

MVP mandate: 30 days. The MLE must produce a minimum viable model — functional, documented, and evaluated — within 30 calendar days of Phase 4 entry. This timeline forces scope discipline: select a tractable problem, use available features, and deliver a baseline model that the mission owner can evaluate. Iteration follows after the initial gate is cleared, not before.

Phase 4 standards. The gate output is not a notebook with promising results. It is a functional model with: (1) documented architecture (algorithm, hyperparameters, feature set), (2) reproducible training pipeline in a Code Workspace, and (3) version-controlled model artifact in Foundry. If the model cannot be retrained from the pipeline without manual intervention, it does not pass Phase 4.

Phase 5 standards. Validation must use a holdout dataset that was never seen during training or hyperparameter tuning. The test report must include accuracy, precision, recall, and F1 at minimum. For models affecting personnel or readiness, bias testing across relevant demographic or unit categories is mandatory. Adversarial testing — deliberate injection of edge cases and malformed inputs — is required for any model that processes user-provided or external data. User acceptance testing (UAT) with the mission sponsor closes Phase 5; sponsor sign-off is the gate output.

WARNING: ADP 3-13 states AI/ML systems enable speed; humans provide judgment. No ML model output replaces commander decision authority. Phase 5 UAT must verify that model outputs are presented as decision support — not automated decisions. Any deployment design that removes the human from the decision loop requires C2DAO architecture review before proceeding.

Source: DDOF Playbook v2.2 (Dec 2025)

DRAFT

1-9. ML Model as UDRA Data Product

BLUF: Every ML model deployed on MSS is a data product governed by UDRA v1.1. The VAULTIS-A framework defines eight quality dimensions that the MLE must satisfy before a model enters production. A model that does not meet all eight dimensions is not a compliant data product.

UDRA v1.1 treats ML models the same as any other data product in the Army enterprise. The VAULTIS-A framework (superseding the earlier VAUTI dimensions) provides the compliance checklist. The MLE is responsible for ensuring the model meets each dimension; the C2DAO verifies compliance at the production deployment gate.

VAULTIS-A Dimension	ML Application
V — Visible	Model registered in the Artifact Data Catalog (ADC) with discoverable metadata
A — Accessible	Inference API or Ontology-backed property available to authorized consumers
U — Understandable	Model card published: purpose, limitations, intended use, known failure modes
L — Linked	Training data lineage documented from source datasets through feature pipeline to model artifact
T — Trustworthy	Validation metrics (accuracy, precision, recall, F1) documented and meeting mission-owner thresholds
I — Interoperable	Inputs and outputs use standard Foundry data formats; no proprietary serialization without C2DAO approval
S — Secure	Classification marking applied to model artifact, training data, and inference outputs per DA PAM 25-2-5
A — Auditable	Training logs, inference logs, and retraining history retained per data retention policy

Minimum compliance. All eight dimensions must be addressed in the model governance documentation (see Chapter 9). The C2DAO will not approve production deployment for a model missing any VAULTIS-A dimension. The MLE should use this table as a pre-submission checklist.

Source: UDRA v1.1 (February 2025)

CHAPTER 2 — CODE WORKSPACES

2-1. Overview

BLUF: Code Workspaces provide JupyterLab and RStudio environments running inside Foundry with direct dataset access, managed compute, and full version control. All MLE development on MSS occurs inside an authorized Code Workspace — never on local machines or external compute.

Code Workspaces are Foundry-managed development environments that execute code on Foundry infrastructure with direct access to datasets, the Ontology, and Foundry Transforms. They eliminate the data movement problem: you read Foundry datasets directly in-notebook without exporting files. Every notebook is version-controlled through Foundry's Git-backed repository system.

Code Workspace types available on MSS:

Environment	Language	Primary Use
JupyterLab	Python (3.10+)	ML training, EDA, feature engineering, pipeline development
RStudio	R	Statistical analysis, ORSA workflows (see SL 4G)
VS Code (Preview)	Python / TypeScript	Application development (see SL 4L)

2-2. Task: Create and Configure a Code Workspace

CONDITIONS: You have a Foundry project with appropriate permissions. You are assigned to an approved ML use case. You have SL 3 prerequisites satisfied.

STANDARDS: A configured JupyterLab workspace that connects to at least one Foundry dataset, runs a Python environment with required libraries installed, and is committed to version control.

EQUIPMENT: MSS (Foundry) access with Code Workspaces enabled; access to the target dataset.

PROCEDURE:

1. Navigate to your Foundry project. Select **New** > **Code Workspace**.
2. Select **JupyterLab** as the environment type.
3. Name the workspace using the convention: `[PROJECT]-[USECASE]-[YOUR_INITIALS]` (e.g., `READINESS-C2PRED-JRS`).
4. Select compute profile. For initial development, use **Standard** (4 vCPU, 16GB RAM). Reserve **Large** (16 vCPU, 64GB RAM) for training runs on datasets exceeding 10M rows. Large compute incurs higher resource cost — do not leave it running idle.
5. Select the Python version: **Python 3.10** (MSS standard).

6. Click **Create**. Wait for workspace initialization (typically 60-120 seconds).
7. Once the workspace opens in JupyterLab, open a terminal (`File > New > Terminal`).
8. Verify the Foundry Python SDK is available:

```
import foundry
print(foundry.__version__)
```

1. Install project-specific dependencies using the workspace terminal:

```
pip install scikit-learn==1.4.0 pandas==2.1.4 numpy==1.26.3 matplotlib==3.8.2 \
    shap==0.44.0 optuna==3.5.0 mlflow==2.10.0
```

1. Commit the environment to version control. Open a terminal and run:

```
# From workspace root
pip freeze > requirements.txt
git add requirements.txt
git commit -m "Initialize workspace: add ML dependencies"
```

NOTE

Pin all dependency versions in requirements.txt. Unpinned dependencies are a reproducibility failure. Six months from now, `pip install scikit-learn` may install a version with breaking API changes that corrupts your serialized model.

CAUTION

Do not install packages that attempt to make outbound network connections to external package registries not approved by the MSS environment. Use only packages available through the Foundry-managed package repository. If a required package is not available, submit a request through the C2DAO package approval process.

2-3. Task: Connect to Foundry Datasets in a Notebook

CONDITIONS: A configured Code Workspace. A target Foundry dataset with read permissions granted to your user account.

STANDARDS: Successfully read a Foundry dataset into a pandas DataFrame, verify schema, and perform basic exploratory checks.

EQUIPMENT: Code Workspace (JupyterLab); Foundry dataset RID.

PROCEDURE:

1. Obtain the dataset RID from the Foundry dataset page (`...` menu > **Copy RID**). The RID format is: `ri.foundry.main.dataset.xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`.
2. In a notebook cell, import the Foundry dataset client:

```
from foundry.transforms import Dataset

# Read a Foundry dataset by RID
dataset = Dataset.get("ri.foundry.main.dataset.<your-dataset-rid>")
df = dataset.pandas()

# Always verify schema and shape immediately after read
print(f"Shape: {df.shape}")
print(f"\nSchema:\n{df.dtypes}")
print(f"\nNull counts:\n{df.isnull().sum()}")
print(f"\nFirst 5 rows:\n{df.head()}")
```

1. For large datasets (>1M rows), read a stratified sample for initial EDA:

```
# Sample 100k rows for exploratory work — do not train on samples unless
# you have confirmed the sample is representative (see Chapter 5)
df_sample = dataset.pandas(row_limit=100_000)
```

1. For Spark-based access on very large datasets (>50M rows):

```
from pyspark.sql import SparkSession
from foundry.transforms import Dataset

spark = SparkSession.builder.getOrCreate()
dataset = Dataset.get("ri.foundry.main.dataset.<your-dataset-rid>")
df_spark = dataset.spark(spark)

# Perform heavy aggregations in Spark before collecting to pandas
df_agg = df_spark.groupBy("unit_uic").agg({"C_rating": "mean"}).toPandas()
```

1. Verify row counts against the Foundry dataset lineage panel. If the count in your notebook differs from the lineage panel by more than 1%, investigate before proceeding.

CAUTION

Never call `.toPandas()` on a full Spark DataFrame exceeding available driver memory. For large DataFrames, aggregate or filter in Spark first, then collect.

NOTE

Dataset RIDs are stable across schema changes but dataset branch matters. Confirm you are reading from the correct branch (`master` for production data; feature branch for development data).

DRAFT

2-4. Task: Manage Compute and Environment Resources

CONDITIONS: Active Code Workspace. Ongoing training run or EDA session.

STANDARDS: Compute is right-sized for the task, not left running idle, and environment state is committed to version control before shutdown.

EQUIPMENT: Active Code Workspace (JupyterLab); access to Foundry Manage Workspaces panel.

PROCEDURE:

1. **Right-size compute for the task:**

Task	Recommended Profile
EDA on <1M rows	Standard (4 vCPU, 16GB)
EDA on 1M–10M rows	Standard or Medium (8 vCPU, 32GB)
Model training on <1M rows (sklearn)	Standard
Model training on 1M–10M rows	Medium or Large
PyTorch training (GPU-required)	GPU profile (request via C2DAO)
Spark processing >50M rows	Spark cluster (request via C2DAO)

1. Before shutting down a workspace, commit all changes:

```
git add -A
git commit -m "WIP: [brief description of state]"
```

1. Shut down the workspace when not actively working. Navigate to **Manage Workspaces** > select workspace > **Stop**. A stopped workspace retains state but does not consume compute resources.
2. If a training run will take more than 30 minutes, consider using a Foundry Transform instead of an interactive notebook (see Chapter 4). Transforms run as managed jobs, survive workspace shutdown, and produce versioned outputs.

2-5. Task: Version Control in Code Workspaces

CONDITIONS: Active Code Workspace with pending changes.

STANDARDS: All code changes committed to a named feature branch, with descriptive commit messages. No uncommitted work lost across sessions.

EQUIPMENT: Active Code Workspace (JupyterLab); terminal access within the workspace; Git configured.

PROCEDURE:

1. Create a feature branch for each distinct piece of work:

```
git checkout -b feature/readiness-feature-engineering
```

1. Commit frequently with meaningful messages:

```
# Good
git commit -m "Add C-rating rolling average features (30/60/90 day windows)"

# Unacceptable
git commit -m "changes"
git commit -m "wip"
git commit -m "fix"
```

1. Before merging to main, run a final validation check:

```
# validate_pipeline.py — run before any merge to main
import subprocess, sys

checks = [
    "pytest tests/ -v",           # Unit tests
    "python -m py_compile *.py",  # Syntax check all Python files
]

for cmd in checks:
    result = subprocess.run(cmd.split(), capture_output=True, text=True)
    if result.returncode != 0:
        print(f"FAILED: {cmd}\n{result.stderr}")
        sys.exit(1)

print("All checks passed. Ready for merge.")
```

1. Open a pull request for peer review before merging ML code to the main branch. Require at least one reviewer who can evaluate the statistical methodology, not just code style.

NOTE — Palantir Developers reference: *Code Repositories* | *How to Unit Test PySpark in Palantir Foundry* — Demonstrates how to write and run unit tests for PySpark Transforms directly within Foundry Code Repositories. Applying this pattern to feature pipeline and inference code is the primary quality gate for ML code before merging to main. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

DRAFT

CHAPTER 3 — FEATURE ENGINEERING

CODE EXAMPLES: Runnable Foundry transform patterns used throughout this chapter are available in the local development shim at `data_skills/13_foundry_patterns/python_transforms.py` and `data_skills/13_foundry_patterns/incremental_transforms.py`. These files mirror the real Foundry API and can be executed locally for development and testing. Activate the venv: `source skills/data_skills/.venv/bin/activate`.

3-1. Overview

BLUF: Feature engineering is the highest-leverage activity in the ML lifecycle. A well-constructed feature pipeline built on curated Foundry datasets produces reproducible, production-ready features. The MLE builds these pipelines as reusable Foundry Transforms, not one-off notebook cells.

NOTE — Palantir Developers reference: *Spark Basics* | *Partitions* — Covers Spark partition fundamentals that directly affect Transform performance when processing large Foundry datasets. Understanding partition count and data skew is critical for building feature pipelines that scale without timeout or OOM failures. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Spark Basics* | *Shuffling* — Explains shuffle operations and the cost of wide transformations (groupBy, join, window functions) in Spark. Knowing when to avoid or minimize shuffles is essential for authoring efficient feature engineering Transforms. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

Feature engineering is the process of transforming raw data into inputs that a model can learn from. For MSS use cases, this means translating military operational data — unit readiness reports, supply requisitions, personnel records, maintenance records — into numeric or categorical feature vectors that capture operationally meaningful signals.

Feature engineering principles for MSS:

1. **Features must be interpretable.** An operational decision-maker who asks "why did the model flag this unit?" must receive an answer traceable to a real-world quantity. Black-box features derived from opaque transformations are not acceptable in operational contexts.
2. **Features must be reproducible.** The training feature pipeline must be identical to the inference feature pipeline. Divergence between training and inference features (training-serving skew) is a primary source of silent model failure.
3. **Features must respect data freshness.** A feature derived from data with a 72-hour staleness window is not valid for a prediction that must be current as of H-1.
4. **Features must not leak the target.** Time-aware train/test splits are mandatory for any time-series operational data.

3-2. Task: Build a Feature Engineering Transform

CONDITIONS: Curated Foundry dataset(s) with appropriate read permissions. An approved ML use case. A target variable identified in coordination with the mission owner.

STANDARDS: A Foundry Transform that reads one or more input datasets, applies documented feature transformations, and outputs a feature dataset with: no null values in model input columns, correct datatypes, documented feature definitions, and a row-level record ID linking each feature vector to its source record.

EQUIPMENT: Code Workspace; Foundry Transforms; source datasets.

PROCEDURE:

1. Create a new Python Transforms file in your repository (e.g., `transforms/feature_engineering.py`).
2. Define inputs and output using the `@transform_df` decorator:

```
from transforms.api import transform_df, Input, Output
import pandas as pd
import numpy as np
from datetime import datetime


@transform_df(
    Output("/ml/readiness/features/unit_readiness_features"),
    unit_status=Input("/data/curated/unit_status_reports"),
    equipment=Input("/data/curated/equipment_on_hand"),
    personnel=Input("/data/curated/personnel_strength"),
)
def compute_features(unit_status, equipment, personnel):
    """
    Build feature vectors for unit C-rating prediction.

    Features:
    - Personnel fill rate (%) at report time
    - Equipment on-hand rate (%) at report time
    - Rolling 30-day C-rating average (lagged — no leakage)
    - Days since last maintenance event
    - Training event count (trailing 90 days)

    Target variable: current_c_rating (4=C1, 3=C2, 2=C3, 1=C4)
    This column is retained in the feature dataset for training
    but must be dropped at inference time.
    """
    # --- Merge inputs on unit identifier (UIC) and report date ---
    df = unit_status.merge(
        equipment[["uic", "report_date", "eoh_rate"]],
        on=["uic", "report_date"],
        how="left"
    ).merge(
        personnel[["uic", "report_date", "personnel_fill_rate"]],
        on=["uic", "report_date"],
        how="left"
```

```
    )

    # --- Sort for time-aware operations ---
    df = df.sort_values(["uic", "report_date"])

    # --- Lagged rolling C-rating (exclude current row to prevent leakage) ---
    df["c_rating_30d_avg"] = (
        df.groupby("uic")["current_c_rating"]
        .transform(lambda x: x.shift(1).rolling(window=30, min_periods=5).mean())
    )

    # --- Days since last maintenance event ---
    df["days_since_maint"] = (
        pd.to_datetime(df["report_date"]) - pd.to_datetime(df["last_maint_date"])
    ).dt.days.clip(lower=0, upper=365)  # Cap at 1 year — older data unreliable

    # --- Training events in trailing 90 days ---
    df["training_events_90d"] = (
        df.groupby("uic")["training_events"]
        .transform(lambda x: x.rolling(window=90, min_periods=1).sum())
    )

    # --- Impute remaining nulls with domain-appropriate defaults ---
    # NOTE: Imputation strategy documented in model card (see Chapter 9)
    df["eoh_rate"] = df["eoh_rate"].fillna(df["eoh_rate"].median())
    df["personnel_fill_rate"] = df["personnel_fill_rate"].fillna(
        df["personnel_fill_rate"].median()
    )
    df["c_rating_30d_avg"] = df["c_rating_30d_avg"].fillna(2.5)  # Neutral prior

    # --- Drop rows where target is null (cannot train without label) ---
    df = df.dropna(subset=["current_c_rating"])

    # --- Select and rename final feature columns ---
    feature_cols = [
        "uic",
        "report_date",
        "personnel_fill_rate",
        "eoh_rate",
        "c_rating_30d_avg",
        "days_since_maint",
        "training_events_90d",
        "current_c_rating",  # Target — drop at inference
    ]

    df = df[feature_cols]

    # --- Validate no nulls in model input columns ---
    model_input_cols = [c for c in feature_cols if c not in ["uic", "report_date",
"current_c_rating"]]
    null_counts = df[model_input_cols].isnull().sum()
    if null_counts.any():
        raise ValueError(
            f"Null values remain in model input columns after
imputation:\n{null_counts[null_counts > 0]}"
```

DRAFT

```
        )

    return df
```

1. Register the transform output as a dataset in Foundry. Run the transform via Pipeline Builder to produce the initial feature dataset.

NOTE — Palantir Developers reference: *Code Repositories | How to Write Data Transformations in Palantir Foundry* — Walks through the core procedure for authoring Python Transforms in Foundry Code Repositories, including the `@transform_df` decorator pattern, input/output registration, and running transforms in development. Directly supplements steps 1–3 of this task. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

1. Document each feature in a `FEATURE_DEFINITIONS.md` file in the repository:

Feature	Description	Source	Staleness Tolerance	Imputation
personnel_fill_rate	Assigned/authorized strength ratio (%)	personnel_strength	72h	Median fill
eoh_rate	Equipment on-hand vs. required (%)	equipment_on_hand	72h	Median EOH
c_rating_30d_avg	Rolling 30-day lagged C-rating average	unit_status_reports	24h	2.5 (neutral)
days_since_maint	Days since most recent maintenance event	unit_status_reports	24h	0 days
training_events_90d	Count of training events in trailing 90 days	unit_status_reports	24h	Rolling sum

WARNING: TIME LEAKAGE. Never include any feature that incorporates information about the future relative to the prediction point. For readiness prediction, a feature computed from the same report that contains the target label is a form of leakage if that information would not be available at inference time. The `.shift(1)` in the rolling average above is not optional — it is what prevents the model from learning a trivial identity mapping. Review every feature for leakage before training.

NOTE — Palantir Developers reference: *Code Repositories | How to Write Incremental Data Transforms in Palantir Foundry* — Covers the `@incremental` decorator pattern for processing only new or changed records rather than recomputing the full dataset on every run. Incrementally-processed feature pipelines are significantly more efficient at production scale and are the standard pattern for high-frequency MSS data feeds. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

3-3. Task: Build a Feature Store Entry

CONDITIONS: A completed feature engineering transform producing a stable feature dataset. Multiple models or analysts require the same features.

STANDARDS: Feature dataset registered as a shared resource in the project, with documented schema, ownership, and update schedule.

EQUIPMENT: Code Workspace; Foundry Transforms; Pipeline Builder; source feature dataset.

PROCEDURE:

1. Identify features that are likely to be consumed by more than one model or analytical workflow. These are candidates for the feature store.
2. Structure the feature store dataset with a consistent schema:

```
# Feature store schema convention for MSS
# Required columns in every feature store entry:
# - entity_id: the unique identifier of the entity (e.g., UIC, DODID, NSN)
# - entity_type: the type of entity ("unit", "person", "item")
# - as_of_date: the date the feature vector was valid (not the date it was computed)
# - feature_set_version: version string (e.g., "v1.2")
# - [feature columns...]: the actual feature values
# - created_at: timestamp of computation (for lineage, not for model input)

FEATURE_STORE_SCHEMA = {
    "entity_id": "string",
    "entity_type": "string",
    "as_of_date": "date",
    "feature_set_version": "string",
    "personnel_fill_rate": "double",
    "eoh_rate": "double",
    "c_rating_30d_avg": "double",
    "days_since_maint": "integer",
    "training_events_90d": "integer",
    "created_at": "timestamp",
}
```

1. Schedule the feature transform to run on the same cadence as the source data updates. If source data updates daily at 0600Z, schedule the feature transform to run at 0700Z. Document this dependency in the pipeline schedule.
2. Register the feature store dataset path in a project-level `FEATURE_REGISTRY.md`:

```
## UNIT READINESS FEATURE SET v1.2
- Dataset path: /ml/readiness/features/unit_readiness_features
- Owner: [MLE name, unit]
- Update schedule: Daily 0700Z
- Source dependencies: unit_status_reports, equipment_on_hand, personnel_strength
- Consumers: readiness_prediction_model_v2, readiness_dashboard_quiver
- Contact: [email]
```

3-4. Common Feature Engineering Patterns for MSS Use Cases

The following patterns appear across most USAREUR-AF ML use cases. Implement them correctly once; reuse them.

Pattern 1: Time-Series Lag Features

```
def add_lag_features(df: pd.DataFrame, group_col: str, value_col: str,
                     lags: list[int]) -> pd.DataFrame:
    """
    Add lagged versions of a column for each group.
    Used for readiness trend features, supply consumption history.
    """
    df = df.sort_values([group_col, "report_date"])
    for lag in lags:
        df[f"{value_col}_lag_{lag}d"] = (
            df.groupby(group_col)[value_col]
            .shift(lag)
        )
    return df

# Usage
df = add_lag_features(df, group_col="uic", value_col="c_rating", lags=[7, 14, 30, 60])
```

Pattern 2: Rolling Window Aggregations

```
def add_rolling_features(df: pd.DataFrame, group_col: str, value_col: str,
                          windows: list[int], min_periods: int = 5) -> pd.DataFrame:
    """
    Add rolling mean, std, min, max for a column.
    Used for demand forecasting feature construction.
    """
    df = df.sort_values([group_col, "date"])
    for w in windows:
        grp = df.groupby(group_col)[value_col]
        df[f"{value_col}_mean_{w}d"] = grp.transform(
            lambda x: x.rolling(w, min_periods=min_periods).mean()
        )
        df[f"{value_col}_std_{w}d"] = grp.transform(
            lambda x: x.rolling(w, min_periods=min_periods).std()
        )
    return df
```

Pattern 3: Categorical Encoding for Military Codes

```
def encode_military_categoricals(df: pd.DataFrame) -> pd.DataFrame:
    """
    Encode standard military categorical fields.
    Use target encoding only on training set — never on full dataset (leakage risk).
    Use ordinal encoding for ordered categories (C1 > C2 > C3 > C4).
    """
```

```
    # Ordinal encoding for C-ratings
    c_rating_map = {"C1": 4, "C2": 3, "C3": 2, "C4": 1, "C5": 0}
    if "readiness_level" in df.columns:
        df["readiness_level_ordinal"] = df["readiness_level"].map(c_rating_map)

    # One-hot for MOS category (7-10 categories max before considering embeddings)
    if "branch_code" in df.columns:
        df = pd.get_dummies(df, columns=["branch_code"], prefix="branch",
drop_first=True)

    return df
```

CHAPTER 4 — MODEL TRAINING

4-1. Overview

BLUF: Model training on MSS uses Code Workspaces for interactive development and Foundry Transforms for production training jobs. All experiments must be tracked. All models must be serialized to a versioned Foundry dataset before deployment.

Training on MSS follows a standard progression: 1. Exploratory training in a notebook (interactive, fast iteration) 2. Experiment tracking enabled (MLflow or Foundry-native) 3. Best model configuration promoted to a production training Transform 4. Trained model artifact serialized and stored in a versioned Foundry dataset 5. Model evaluation (Chapter 5) before any deployment activity

4-2. Task: Train a scikit-learn Model with Experiment Tracking

CONDITIONS: Feature dataset available in Foundry. Code Workspace active. MLflow or Foundry experiment tracking configured.

STANDARDS: Model trained on a proper train/validation/test split. All hyperparameters, metrics, and artifacts logged to the experiment tracker. Trained model serialized and saved to Foundry.

EQUIPMENT: Code Workspace; feature dataset; MLflow (or Foundry experiment tracker).

PROCEDURE:

1. Load the feature dataset and define the train/validation/test split. For time-series operational data, use a temporal split — never a random split.

```
import pandas as pd
import numpy as np
from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassifier
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
import mlflow
import mlflow.sklearn
import joblib
from foundry.transforms import Dataset

# --- Load features ---
feature_dataset = Dataset.get("ri.foundry.main.dataset.<feature-set-rid>")
df = feature_dataset.pandas()

# --- Temporal train/validation/test split ---
# For operational data, NEVER use random_state-based splits.
# Models trained on future data predicting the past are not valid.
df["report_date"] = pd.to_datetime(df["report_date"])
df = df.sort_values("report_date")

# Split: 70% train, 15% validation, 15% test — by date
n = len(df)
train_end_idx = int(n * 0.70)
val_end_idx = int(n * 0.85)

train_df = df.iloc[:train_end_idx]
val_df = df.iloc[train_end_idx:val_end_idx]
test_df = df.iloc[val_end_idx:]

print(f"Train: {len(train_df)} rows ({train_df['report_date'].min().date()} to
{train_df['report_date'].max().date()})")
print(f"Validation: {len(val_df)} rows ({val_df['report_date'].min().date()} to
{val_df['report_date'].max().date()})")
print(f"Test: {len(test_df)} rows ({test_df['report_date'].min().date()} to
{test_df['report_date'].max().date()})")

# --- Define features and target ---
FEATURE_COLS = [
    "personnel_fill_rate",
    "eoh_rate",
    "c_rating_30d_avg",
    "days_since_maint",
    "training_events_90d",
]
TARGET_COL = "current_c_rating"

X_train = train_df[FEATURE_COLS]
y_train = train_df[TARGET_COL]
X_val = val_df[FEATURE_COLS]
y_val = val_df[TARGET_COL]
X_test = test_df[FEATURE_COLS]
y_test = test_df[TARGET_COL]
```

1. Configure MLflow experiment tracking:

```
# Set experiment name — use project + use case + date convention
mlflow.set_experiment("usareur-af/readiness/c-rating-prediction")

EXPERIMENT_PARAMS = {
    "use_case": "readiness_c_rating_prediction",
    "feature_set_version": "v1.2",
    "train_date_range": f"{train_df['report_date'].min().date()} to
{train_df['report_date'].max().date()}",
    "train_n_rows": len(X_train),
    "val_n_rows": len(X_val),
    "test_n_rows": len(X_test),
    "split_method": "temporal",
}
```

1. Train candidate models with experiment tracking:

```
def train_and_log(model, model_name: str, hyperparams: dict,
                  X_train, y_train, X_val, y_val) -> float:
    """Train a model and log to MLflow. Returns validation accuracy."""
    with mlflow.start_run(run_name=model_name):
        # Log experiment context
        mlflow.log_params(EXPERIMENT_PARAMS)
        mlflow.log_params(hyperparams)

        # Create pipeline with preprocessing
        pipeline = Pipeline([
            ("scaler", StandardScaler()),
            ("model", model),
        ])

        # Train
        pipeline.fit(X_train, y_train)

        # Evaluate on validation set
        val_preds = pipeline.predict(X_val)
        val_proba = pipeline.predict_proba(X_val) if hasattr(pipeline,
"predict_proba") else None

        # Log metrics
        from sklearn.metrics import accuracy_score, f1_score
        val_acc = accuracy_score(y_val, val_preds)
        val_f1_macro = f1_score(y_val, val_preds, average="macro")

        mlflow.log_metric("val_accuracy", val_acc)
        mlflow.log_metric("val_f1_macro", val_f1_macro)

        if val_proba is not None:
            # Multiclass ROC-AUC (one-vs-rest)
            try:
                val_auc = roc_auc_score(y_val, val_proba, multi_class="ovr")
                mlflow.log_metric("val_roc_auc_ovr", val_auc)
            except Exception:
                pass  # AUC requires all classes represented in val set
```

```
        # Log the trained pipeline
        mlflow.sklearn.log_model(pipeline, artifact_path="model")

        print(f"{model_name}: val_accuracy={val_acc:.4f}, val_f1_macro=
{val_f1_macro:.4f}")
        return val_acc

# --- Train candidate models ---
candidates = [
    (
        GradientBoostingClassifier(n_estimators=200, max_depth=4, learning_rate=0.05,
random_state=42),
        "GBM_baseline",
        {"n_estimators": 200, "max_depth": 4, "learning_rate": 0.05}
    ),
    (
        RandomForestClassifier(n_estimators=200, max_depth=8, random_state=42,
n_jobs=-1),
        "RandomForest_baseline",
        {"n_estimators": 200, "max_depth": 8}
    ),
    (
        LogisticRegression(C=1.0, max_iter=500, multi_class="multinomial",
random_state=42),
        "LogisticRegression_baseline",
        {"C": 1.0, "max_iter": 500}
    ),
]

results = {}
for model, name, params in candidates:
    acc = train_and_log(model, name, params, X_train, y_train, X_val, y_val)
    results[name] = acc

best_model_name = max(results, key=results.get)
print(f"\nBest model: {best_model_name} (val_accuracy=
{results[best_model_name]:.4f})")
```

1. Retrain the best model on train+validation combined, then evaluate on the held-out test set:

```
# Retrain best model on train+val combined for final evaluation
X_trainval = pd.concat([X_train, X_val])
y_trainval = pd.concat([y_train, y_val])

# Use best hyperparams from validation phase
final_model = Pipeline([
    ("scaler", StandardScaler()),
    ("model", GradientBoostingClassifier(
        n_estimators=200, max_depth=4, learning_rate=0.05, random_state=42
    )),
])

with mlflow.start_run(run_name="final_model_test_evaluation"):
    final_model.fit(X_trainval, y_trainval)
```

```
    # Test set evaluation
    test_preds = final_model.predict(X_test)
    test_report = classification_report(y_test, test_preds, output_dict=True)

    mlflow.log_metric("test_accuracy", test_report["accuracy"])
    mlflow.log_metric("test_f1_macro", test_report["macro avg"]["f1-score"])
    mlflow.sklearn.log_model(final_model, artifact_path="final_model")

    print(classification_report(y_test, test_preds))
    print(f"Confusion matrix:\n{confusion_matrix(y_test, test_preds)}")
```

1. Serialize the final model and save it to a Foundry dataset:

```
import io
from foundry.transforms import Dataset

# Serialize to bytes
model_buffer = io.BytesIO()
joblib.dump(final_model, model_buffer)
model_bytes = model_buffer.getvalue()

# Write to Foundry dataset (binary artifact store)
model_artifact_dataset = Dataset.get_or_create(
    "/ml/readiness/models/c_rating_predictor_v1"
)
with model_artifact_dataset.transaction() as tx:
    with tx.write_file("model.joblib") as f:
        f.write(model_bytes)
    with tx.write_file("metadata.json") as f:
        import json
        json.dump({
            "model_name": "c_rating_predictor",
            "version": "v1.0",
            "feature_set_version": "v1.2",
            "feature_cols": FEATURE_COLS,
            "target_col": TARGET_COL,
            "test_accuracy": test_report["accuracy"],
            "test_f1_macro": test_report["macro avg"]["f1-score"],
            "train_date_range": EXPERIMENT_PARAMS["train_date_range"],
            "trained_by": "[MLE name]",
            "trained_date": pd.Timestamp.now().isoformat(),
        }, f)

print("Model artifact saved to Foundry.")
```

NOTE

The `metadata.json` alongside every serialized model is mandatory, not optional. When a governance reviewer or an on-call MLE six months from now opens the artifact store, the metadata is the first thing they read.

DRAFT

4-3. Task: Hyperparameter Tuning with Optuna

CONDITIONS: Baseline model trained. Validation metric identified. Compute budget available.

STANDARDS: Tuning study logged to MLflow. Best hyperparameters documented. Tuning runtime under 2 hours on Standard compute.

EQUIPMENT: Code Workspace; feature dataset (train/validation split prepared); MLflow experiment tracker; Optuna installed in workspace environment.

PROCEDURE:

1. Define the Optuna objective function, create a study, and optimize. Log best parameters to MLflow:

```
import optuna
import mlflow

def objective(trial: optuna.Trial) -> float:
    """Optuna objective: maximize validation F1-macro for C-rating prediction."""

    params = {
        "n_estimators": trial.suggest_int("n_estimators", 100, 500),
        "max_depth": trial.suggest_int("max_depth", 2, 8),
        "learning_rate": trial.suggest_float("learning_rate", 0.01, 0.3, log=True),
        "subsample": trial.suggest_float("subsample", 0.6, 1.0),
        "min_samples_leaf": trial.suggest_int("min_samples_leaf", 5, 50),
    }

    model = Pipeline([
        ("scaler", StandardScaler()),
        ("model", GradientBoostingClassifier(**params, random_state=42)),
    ])
    model.fit(X_train, y_train)

    val_preds = model.predict(X_val)
    return f1_score(y_val, val_preds, average="macro")


# Create study and optimize
study = optuna.create_study(direction="maximize", study_name="c_rating_gbm_tuning")
study.optimize(objective, n_trials=50, timeout=3600)  # 50 trials or 1 hour

best_params = study.best_params
print(f"Best params: {best_params}")
print(f"Best val F1 macro: {study.best_value:.4f}")

# Log best params to MLflow
with mlflow.start_run(run_name="optuna_best"):
    mlflow.log_params(best_params)
    mlflow.log_metric("val_f1_macro", study.best_value)
```

1. After the study completes, use `best_params` to retrain the final model on train+validation combined and run against the held-out test set (see Task 4-2, Step 4 for the final evaluation pattern).

2. Document the best hyperparameters in the model card and MLflow run. Record the tuning study name, number of trials, and final validation metric.

4-4. Task: Training a PyTorch Model on Foundry Data

CONDITIONS: Feature dataset in Foundry. PyTorch available in workspace environment. Use case requires a neural network (tabular deep learning or time-series forecasting).

STANDARDS: Training loop with validation loss tracking, early stopping, model serialization, and full MLflow logging.

EQUIPMENT: Code Workspace; feature dataset; PyTorch and MLflow installed in workspace environment; Foundry artifact store dataset for model output.

> **NOTE**
>
> For most tabular MSS use cases, gradient boosting methods (GBM, XGBoost) outperform neural networks and are significantly easier to interpret. Use PyTorch for time-series sequence modeling (LSTM for demand forecasting), image-based use cases, or when an existing research baseline specifically requires a neural architecture.

PROCEDURE:

1. Prepare tensors, define the model architecture, configure the optimizer, loss function, and learning rate scheduler. Implement the training loop with early stopping and MLflow logging. Save the best model state and serialize to Foundry on completion:

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset
import mlflow

# --- Prepare tensors ---
X_train_t = torch.tensor(X_train.values, dtype=torch.float32)
y_train_t = torch.tensor(y_train.values - 1, dtype=torch.long)  # 0-indexed classes
X_val_t = torch.tensor(X_val.values, dtype=torch.float32)
y_val_t = torch.tensor(y_val.values - 1, dtype=torch.long)

train_loader = DataLoader(
    TensorDataset(X_train_t, y_train_t),
    batch_size=256,
    shuffle=True,
)

# --- Define model ---
class ReadinessClassifier(nn.Module):
    def __init__(self, input_dim: int, n_classes: int, hidden_dim: int = 128):
        super().__init__()
```

```
        self.net = nn.Sequential(
            nn.BatchNorm1d(input_dim),
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(hidden_dim // 2, n_classes),
        )

    def forward(self, x):
        return self.net(x)


model = ReadinessClassifier(
    input_dim=len(FEATURE_COLS),
    n_classes=len(y_train.unique()),
)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4)
criterion = nn.CrossEntropyLoss()
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode="min", patience=5, factor=0.5
)

# --- Training loop with early stopping ---
best_val_loss = float("inf")
patience_counter = 0
PATIENCE = 10
MAX_EPOCHS = 100

with mlflow.start_run(run_name="pytorch_readiness_classifier"):
    mlflow.log_params({
        "hidden_dim": 128,
        "batch_size": 256,
        "lr": 1e-3,
        "dropout": 0.3,
        "patience": PATIENCE,
    })

    for epoch in range(MAX_EPOCHS):
        model.train()
        train_loss = 0.0
        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()
            logits = model(X_batch)
            loss = criterion(logits, y_batch)
            loss.backward()
            optimizer.step()
            train_loss += loss.item() * len(X_batch)

        train_loss /= len(X_train_t)

        # Validation
```

DRAFT

```
        model.eval()
        with torch.no_grad():
            val_logits = model(X_val_t)
            val_loss = criterion(val_logits, y_val_t).item()
            val_preds = val_logits.argmax(dim=1).numpy()

        val_acc = (val_preds == y_val_t.numpy()).mean()
        scheduler.step(val_loss)

        mlflow.log_metrics({
            "train_loss": train_loss,
            "val_loss": val_loss,
            "val_accuracy": val_acc,
        }, step=epoch)

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            torch.save(model.state_dict(), "best_model.pt")
            patience_counter = 0
        else:
            patience_counter += 1

        if patience_counter >= PATIENCE:
            print(f"Early stopping at epoch {epoch}")
            break

    # Save best model
    model.load_state_dict(torch.load("best_model.pt"))
    mlflow.pytorch.log_model(model, "pytorch_model")
    print(f"Training complete. Best val_loss={best_val_loss:.4f}")
```

1. After training completes, serialize the best model state to a Foundry artifact store dataset using the `save_model_to_foundry` pattern from Appendix C-3. Include a `metadata.json` file specifying model architecture, training date range, and feature columns.
2. Proceed to Chapter 5 for evaluation. PyTorch model evaluation uses the same evaluation framework as scikit-learn models — load the model, run inference on the test set, compute classification metrics.

CHAPTER 5 — MODEL EVALUATION

5-1. Overview

BLUF: Model evaluation for operational use requires more than accuracy metrics. The MLE must assess performance across operationally relevant subgroups, verify the model meets the mission owner's threshold, check for bias in personnel-affecting models, and document everything in the model evaluation report before any deployment activity begins.

Model evaluation is a gate, not a formality. A model that does not pass evaluation does not deploy. The MLE does not decide whether evaluation results are "good enough" for operational use — the mission owner and C2DAO make that determination based on documented evidence the MLE provides.

5-2. Task: Compute and Document the Model Evaluation Report

CONDITIONS: Trained model serialized to Foundry. Test dataset available (temporally held out, never used during training or hyperparameter tuning).

STANDARDS: Evaluation report covering: overall metrics, per-class metrics, confusion matrix, operationally meaningful error analysis, and cross-validation summary. Report submitted to mission owner and data steward.

EQUIPMENT: Code Workspace; serialized model artifact in Foundry; test feature dataset; scikit-learn, SHAP, matplotlib, seaborn installed.

PROCEDURE:

1. Compute the full evaluation suite on the held-out test set:

```
import pandas as pd
import numpy as np
from sklearn.metrics import (
    classification_report, confusion_matrix, roc_auc_score,
    precision_recall_curve, average_precision_score,
    brier_score_loss
)
import matplotlib
matplotlib.use("Agg")  # Headless — no display needed in Foundry
import matplotlib.pyplot as plt
import seaborn as sns

# Load model and test data
# (model loaded from Foundry artifact store — see Chapter 6 for load pattern)

test_preds = final_model.predict(X_test)
test_proba = final_model.predict_proba(X_test)

# --- Overall metrics ---
print("=== TEST SET EVALUATION REPORT ===")
print(f"Model: c_rating_predictor_v1")
print(f"Test period: {test_df['report_date'].min().date()} to
{test_df['report_date'].max().date()}")
print(f"Test rows: {len(X_test)}")
print()
print(classification_report(y_test, test_preds, target_names=["C4","C3","C2","C1"]))

# --- Confusion matrix ---
cm = confusion_matrix(y_test, test_preds)
fig, ax = plt.subplots(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
```

```
            xticklabels=["C4","C3","C2","C1"],
            yticklabels=["C4","C3","C2","C1"])
ax.set_xlabel("Predicted")
ax.set_ylabel("Actual")
ax.set_title("Confusion Matrix — C-Rating Prediction")
plt.tight_layout()
plt.savefig("confusion_matrix.png", dpi=150)
plt.close()  # Always close figure

# --- Operational error analysis: directional error ---
# For readiness prediction, overpredicting readiness (C2 predicted, actual C4)
# is a more dangerous error than underpredicting.
# Compute the "optimistic error rate" — predicted >= actual + 1 level
test_df_eval = test_df.copy()
test_df_eval["predicted"] = test_preds
test_df_eval["error"] = test_df_eval["predicted"] - test_df_eval[TARGET_COL]
test_df_eval["optimistic_error"] = (test_df_eval["error"] > 0).astype(int)

optimistic_rate = test_df_eval["optimistic_error"].mean()
print(f"\nOptimistic error rate (predicted more ready than actual):
{optimistic_rate:.3f}")
print("Mission owner threshold: [document in evaluation report]")
```

1. Compute time-series cross-validation to validate temporal stability:

```
from sklearn.model_selection import TimeSeriesSplit

# Walk-forward validation: 5 folds
tscv = TimeSeriesSplit(n_splits=5, gap=30)  # 30-day gap between train and val

fold_metrics = []
for fold_idx, (train_idx, val_idx) in enumerate(tscv.split(df)):
    fold_train = df.iloc[train_idx]
    fold_val = df.iloc[val_idx]

    X_fold_train = fold_train[FEATURE_COLS]
    y_fold_train = fold_train[TARGET_COL]
    X_fold_val = fold_val[FEATURE_COLS]
    y_fold_val = fold_val[TARGET_COL]

    fold_model = Pipeline([
        ("scaler", StandardScaler()),
        ("model", GradientBoostingClassifier(
            n_estimators=200, max_depth=4, learning_rate=0.05, random_state=42
        )),
    ])
    fold_model.fit(X_fold_train, y_fold_train)
    fold_preds = fold_model.predict(X_fold_val)

    fold_acc = (fold_preds == y_fold_val.values).mean()
    fold_f1 = f1_score(y_fold_val, fold_preds, average="macro")
    fold_metrics.append({"fold": fold_idx, "accuracy": fold_acc, "f1_macro": fold_f1})
    print(f"Fold {fold_idx}: accuracy={fold_acc:.4f}, f1_macro={fold_f1:.4f}")
```

```
fold_df = pd.DataFrame(fold_metrics)
print(f"\nCV mean accuracy: {fold_df['accuracy'].mean():.4f} ±
{fold_df['accuracy'].std():.4f}")
print(f"CV mean f1_macro: {fold_df['f1_macro'].mean():.4f} ±
{fold_df['f1_macro'].std():.4f}")
```

1. Compute feature importance and SHAP values for interpretability:

```
import shap

# Compute SHAP values (use TreeExplainer for GBM)
explainer = shap.TreeExplainer(final_model.named_steps["model"])
X_test_scaled = final_model.named_steps["scaler"].transform(X_test)
shap_values = explainer.shap_values(X_test_scaled)

# Summary plot (saved headlessly)
fig = plt.figure(figsize=(8, 5))
shap.summary_plot(shap_values, X_test_scaled, feature_names=FEATURE_COLS,
                  class_names=["C4","C3","C2","C1"], show=False)
plt.tight_layout()
plt.savefig("shap_summary.png", dpi=150, bbox_inches="tight")
plt.close()

print("SHAP summary saved.")
```

NOTE

SHAP values are not optional for models that affect operational decisions. The mission owner must be able to understand why the model predicts what it predicts for any specific prediction. "The model said so" is not an acceptable answer to a commander asking why a unit was flagged as C3.

5-3. Task: Conduct Bias and Fairness Assessment

CONDITIONS: Model evaluation complete. Model affects assessments of personnel, units, or individual readiness status.

STANDARDS: Documented demographic parity analysis, equalized odds analysis, and documented residual risk acceptance by the mission owner and G1 (for personnel-affecting models).

EQUIPMENT: Code Workspace; test evaluation dataset with subgroup attribute columns; evaluation results from Task 5-2.

WARNING: BIAS IN PERSONNEL-AFFECTING MODELS. Any model whose output is used to make or inform decisions about individual service members — duty assignments, readiness flags, performance assessments, logistics priority scoring for units of different types — is subject to bias review. This is a DoD AI Ethics requirement and an Army policy obligation. Failing to conduct this review before deployment is not a technical oversight — it is a governance violation.

PROCEDURE:

1. Define the protected attributes relevant to the use case and check distributional parity:

```
def compute_group_metrics(df_eval: pd.DataFrame, protected_col: str,
                           pred_col: str, target_col: str) -> pd.DataFrame:
    """
    Compute prediction accuracy and positive rate by protected group.
    'Positive' defined as predicted C1 or C2 (ready) vs C3/C4 (not ready).
    """
    results = []
    for group_val in df_eval[protected_col].unique():
        group = df_eval[df_eval[protected_col] == group_val]
        acc = (group[pred_col] == group[target_col]).mean()
        # Positive rate: fraction predicted as C1 or C2
        pos_rate = (group[pred_col] >= 3).mean()
        results.append({
            "group": group_val,
            "n": len(group),
            "accuracy": acc,
            "positive_rate": pos_rate,
        })
    return pd.DataFrame(results)


# For unit-level readiness model, protected attributes might include:
# - unit_type (armor, infantry, aviation, sustainment)
# - echelon (company, battalion, brigade)
# - theater_location (Germany, Poland, Romania)

for protected_attr in ["unit_type", "echelon", "theater_location"]:
    if protected_attr in test_df_eval.columns:
        group_metrics = compute_group_metrics(
            test_df_eval, protected_attr, "predicted", TARGET_COL
        )
        print(f"\n=== Group metrics by {protected_attr} ===")
        print(group_metrics.to_string(index=False))

        # Flag if accuracy spread exceeds 15% — document and escalate
        acc_spread = group_metrics["accuracy"].max() - group_metrics["accuracy"].min()
        if acc_spread > 0.15:
            print(f"\n[FLAG] Accuracy spread across {protected_attr} groups:
{acc_spread:.3f} > 0.15")
            print("Document in model card and escalate to mission owner for risk
acceptance.")
```

1. Document findings in the bias section of the model evaluation report. If bias flags are raised, the mission owner must sign a documented risk acceptance statement before deployment proceeds. There is no bypass for this step.

5-4. Evaluation Metrics Reference by Use Case

Use Case	Primary Metric	Secondary Metrics	Operational Threshold
C-rating prediction	Weighted F1-score	Per-class recall, confusion matrix, optimistic error rate	Weighted F1 >= 0.72; optimistic error rate < 0.10
Demand forecasting	MAE (units)	RMSE, MAPE, bias (over/under)	MAE < [domain-specific]; no systematic under-forecast bias
Anomaly detection	Precision at top-k	Recall, false positive rate, alert volume	Precision at top-20 >= 0.60; false positive rate < 0.05/day
Binary readiness flag	F1-score, PR-AUC	Precision, recall, threshold sensitivity	Mission owner defines operational precision/recall tradeoff

NOTE

Thresholds in the table above are starting points derived from USAREUR-AF initial deployments. Mission owners must confirm or adjust these thresholds based on operational tolerance for each specific deployment. Document the agreed threshold in the model card before deployment.

CHAPTER 6 — MODEL DEPLOYMENT

6-1. Overview

BLUF: Deploying a model on MSS means integrating it into the Foundry data product ecosystem — publishing predictions as an Ontology Object property, running batch inference as a scheduled Transform, or exposing predictions through an API endpoint. All three patterns require governance approval before production deployment.

Model deployment is not the end of the MLE's responsibility — it is the start of the monitoring responsibility. A deployed model that is not monitored is a liability.

DRAFT

6-2. Deployment Patterns on MSS

Pattern	Use Case	Latency	Governance Complexity
Batch inference Transform	Daily/weekly predictions appended to dataset	Hours	Standard
Ontology-backed Object property	Prediction visible as Object property in Workshop	Near-real-time (on schedule)	High — C2DAO review required
Model-backed API endpoint	On-demand predictions from Workshop Actions or external callers	Sub-second	Highest — architecture review required
Notebook-based inference	Ad hoc analysis, one-time scoring	N/A	Standard (no production deployment)

Most USAREUR-AF operational use cases use the batch inference Transform pattern. The Ontology-backed Object property pattern provides the best user experience for Workshop application consumers.

NOTE — Palantir Developers reference: *Build with AIP: Compute Modules* — Covers Compute Modules, which allow custom code to be invoked within AIP Logic workflows. When a model deployed here needs to be called on-demand inside an AIP Logic action or agent workflow, Compute Modules are the integration pattern — coordinate with the SL 4H AI Engineer for this interface. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *AIP with Jeg: Building a Streaming Ingestor with Compute Modules* — Demonstrates how to build a streaming data ingestor using Compute Modules for real-time ingestion use cases. Relevant when the operational requirement calls for sub-minute data freshness feeding ML inference rather than scheduled batch transforms. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

6-3. Task: Deploy Batch Inference as a Foundry Transform

CONDITIONS: Model artifact stored in Foundry dataset. Feature engineering pipeline operational. Production deployment approval obtained (Chapter 9).

STANDARDS: A scheduled Transform that loads the latest model artifact, computes predictions on the current feature dataset, and appends predictions to an output dataset. Transform is idempotent and includes error handling.

EQUIPMENT: Code Workspace; model artifact dataset; feature dataset; Foundry Transforms.

PROCEDURE:

1. Create `transforms/batch_inference.py`:

```
from transforms.api import transform_df, Input, Output
import pandas as pd
import numpy as np
import joblib
import json
import io
from foundry.transforms import Dataset as FoundryDataset


@transform_df(
    Output("/ml/readiness/predictions/c_rating_predictions"),
    features=Input("/ml/readiness/features/unit_readiness_features"),
    model_artifact=Input("/ml/readiness/models/c_rating_predictor_v1"),
)
def run_batch_inference(features, model_artifact):
    """
    Batch inference: load the latest C-rating prediction model and score
    all units in the current feature dataset.

    Output schema:
    - uic: unit identifier
    - report_date: date of the feature vector
    - predicted_c_rating: integer 1-4
    - prediction_confidence: float (max class probability)
    - model_version: string (from model metadata)
    - inference_timestamp: timestamp of this inference run
    """
    # --- Load model artifact from Foundry binary store ---
    # model_artifact is a DataFrame pointing to binary files — use FoundryDataset to
read
    model_dataset = FoundryDataset.get(model_artifact.foundry_schema.rid)

    with model_dataset.open_file("model.joblib", "rb") as f:
        model = joblib.load(f)

    with model_dataset.open_file("metadata.json", "r") as f:
        metadata = json.load(f)

    model_version = metadata.get("model_version", "unknown")
    feature_cols = metadata.get("feature_cols", [])

    # --- Validate required features are present ---
    missing_cols = set(feature_cols) - set(features.columns)
    if missing_cols:
        raise ValueError(
            f"Feature dataset is missing required columns: {missing_cols}. "
            f"Check for feature pipeline schema changes."
        )

    # --- Run inference ---
    X = features[feature_cols]
    predictions = model.predict(X)
    probabilities = model.predict_proba(X)
    confidence = probabilities.max(axis=1)
```

```
    # --- Build output DataFrame ---
    output = features[["uic", "report_date"]].copy()
    output["predicted_c_rating"] = predictions
    output["prediction_confidence"] = confidence
    output["model_version"] = model_version
    output["inference_timestamp"] = pd.Timestamp.utcnow()

    # --- Validate output ---
    assert output["predicted_c_rating"].between(1, 4).all(), \
        "Predicted C-ratings contain out-of-range values — check model artifact."
    assert output["prediction_confidence"].between(0, 1).all(), \
        "Confidence scores out of range [0,1] — check model artifact."
    assert not output[["uic", "report_date"]].duplicated().any(), \
        "Duplicate (uic, report_date) pairs in inference output."

    return output
```

1. Register the transform in Pipeline Builder. Set the schedule to run after the feature pipeline completes (add an explicit dependency edge in Pipeline Builder).
2. Verify the Transform runs successfully in a development branch before promoting to production.

6-4. Task: Publish Predictions as an Ontology Object Property

CONDITIONS: Batch inference Transform producing predictions to a Foundry dataset. An existing Ontology Object Type for the entity being predicted (e.g., `Unit` Object Type with `uic` as the primary key). Production deployment approval obtained.

STANDARDS: Ontology Object Type updated with a model-backed property that reflects the latest batch inference output. The property displays correctly in Workshop and shows prediction confidence alongside predicted value.

EQUIPMENT: Foundry Ontology Manager access (coordinated with -30 builder); prediction output dataset; Code Workspace for validation queries.

> **NOTE**
>
> Adding a model-backed property to a production Ontology Object Type requires coordination with the -30 builder who owns the Object Type design and the data steward who governs that domain. You do not modify the production Ontology unilaterally.

PROCEDURE:

1. Coordinate with the -30 builder to add the following properties to the `Unit` Object Type:
2. `predicted_c_rating` (Integer) — model-backed, source: prediction dataset
3. `prediction_confidence` (Double) — model-backed, source: prediction dataset
4. `model_version` (String) — model-backed, source: prediction dataset

5. `inference_timestamp` (Timestamp) — model-backed, source: prediction dataset
6. In the Ontology Manager (coordinate with -30 builder), configure the sync rule:
7. Source dataset: `/ml/readiness/predictions/c_rating_predictions`
8. Join key: `uic` matches `Unit.uic`
9. Sync schedule: on dataset transaction completion (event-driven)
10. Validate the sync in the development environment:

```
# Validation query via OSDK (run in notebook after Ontology sync)
from foundry_ml_gateway import OntologyClient  # Foundry-managed client

client = OntologyClient()

# Spot-check 5 units
sample_units = ["W12345", "W23456", "W34567", "W45678", "W56789"]
for uic in sample_units:
    unit = client.objects.Unit.get(uic)
    print(f"{uic}: predicted_c_rating={unit.predicted_c_rating}, "
          f"confidence={unit.prediction_confidence:.3f}, "
          f"model_version={unit.model_version}, "
          f"as_of={unit.inference_timestamp}")
```

1. Verify prediction values match the inference Transform output for the same UICs. If they do not match, halt the deployment and investigate the sync configuration before proceeding.

6-5. Task: Implement Inference Caching and Freshness Indicators

CONDITIONS: Batch inference Transform deployed. Workshop application consuming model-backed properties.

STANDARDS: Prediction freshness visible to Workshop users. Stale predictions (>24 hours old) flagged as degraded. Workshop application does not display model predictions without a visible staleness indicator.

EQUIPMENT: Foundry Workshop editor access; prediction dataset with inference_timestamp column; Code Workspace for transform-based freshness computation.

PROCEDURE:

1. Add a computed property to the Workshop application that displays prediction freshness using conditional display logic:

```
# In Workshop formula editor or Python transform:
# Compute hours since last inference
inference_age_hours = (NOW() - inference_timestamp) / 3600

# Display logic (Workshop conditional formatting):
```

DRAFT

```
# IF inference_age_hours > 24: show WARNING banner "Predictions are stale - last
updated [timestamp]"
# IF inference_age_hours > 72: show ERROR banner "Prediction pipeline may be degraded
- contact data steward"
# IF inference_age_hours <= 24: show normal prediction display

# As a Python feature for the prediction dataset:
def add_freshness_indicator(df: pd.DataFrame) -> pd.DataFrame:
    df["inference_age_hours"] = (
        pd.Timestamp.utcnow() - pd.to_datetime(df["inference_timestamp"])
    ).dt.total_seconds() / 3600
    df["prediction_is_fresh"] = df["inference_age_hours"] <= 24
    df["prediction_staleness_flag"] = df["inference_age_hours"].apply(
        lambda h: "CURRENT" if h <= 24 else ("STALE" if h <= 72 else "DEGRADED")
    )
    return df
```

1. Apply the `add_freshness_indicator` function as a step in the batch inference Transform, appending freshness columns to the prediction output dataset before it is synced to the Ontology.
2. In the Workshop application, configure a conditional banner widget: display a warning banner when `prediction_staleness_flag` is STALE or DEGRADED. The banner must include the last inference timestamp so users can assess operational relevance of the displayed predictions.

WARNING: STALE PREDICTIONS IN OPERATIONAL CONTEXT. A readiness prediction that is 96 hours old may reflect pre-exercise manning levels for a unit that is now depleted. A logistics forecast that is 72 hours old during a high-tempo sustainment period may be dangerously misleading. Freshness indicators are not cosmetic — they are operational safety information. Never suppress them to simplify the UI.

CHAPTER 7 — MLOPS ON FOUNDRY

7-1. Overview

BLUF: MLOps on MSS encompasses model versioning, automated retraining triggers, production drift detection, and pipeline orchestration. A model without an MLOps plan is a model that will silently degrade.

The gap between ML prototype and production ML is MLOps. Every model that deploys to the MSS production environment must have, at minimum: a version control strategy, a documented retraining trigger, and an active drift monitoring implementation. These are not best practices — they are governance requirements under UDRA v1.1.

NOTE — Palantir Developers reference: *Pipeline Monitoring* | *How to Start Monitoring Data Health in Palantir Foundry* — Covers foundational pipeline health monitoring in Foundry, including setting up dataset checks and health indicators. This is the starting point for implementing the monitoring

infrastructure described throughout this chapter. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Pipeline Monitoring | How to Monitor Health Across a Pipeline in Palantir Foundry* — Extends single-dataset monitoring to full pipeline health visibility, covering how to track data quality and freshness across a chain of dependent transforms. Directly applicable to the drift detection and retraining trigger patterns in sections 7-3 and 7-4. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

NOTE — Palantir Developers reference: *Deep Dive: Optimizing Data Pipelines with Iceberg Tables and Lightweight Compute | DevCon 4* — An advanced treatment of Iceberg table format for time-travel queries, efficient incremental reads, and production pipeline optimization. Iceberg's snapshot isolation and schema evolution capabilities are directly relevant to maintaining stable feature pipelines and model artifact storage on MSS. Available on the Palantir Developers YouTube channel (@PalantirDevelopers).

7-2. Task: Implement Model Versioning

CONDITIONS: Model artifact stored in Foundry. Multiple versions may exist over time.

STANDARDS: All model versions uniquely identified, stored, and retrievable. A "current production" pointer that can be updated without deleting prior versions. Rollback to any prior version achievable in under 30 minutes.

EQUIPMENT: Code Workspace; Foundry artifact store dataset for model binary files; a dedicated Foundry dataset for the version tracking registry.

PROCEDURE:

1. Use semantic versioning for all model artifacts: `v{MAJOR}.{MINOR}.{PATCH}`.
2. MAJOR: breaking change to feature set or model architecture
3. MINOR: retrain with new data, same feature set
4. PATCH: hyperparameter adjustment, same feature set and data window
5. Implement a model registry pattern in Foundry:

```
# Model registry — a Foundry dataset containing version metadata
# Path: /ml/readiness/model_registry/c_rating_predictor_registry

MODEL_REGISTRY_SCHEMA = {
    "model_name": "string",
    "version": "string",
    "status": "string",       # CANDIDATE | VALIDATED | PRODUCTION | RETIRED
    "artifact_path": "string",
    "feature_set_version": "string",
    "test_accuracy": "double",
    "test_f1_macro": "double",
    "train_date_start": "date",
```

```
    "train_date_end": "date",
    "deployed_date": "date",
    "retired_date": "date",
    "approved_by": "string",
    "notes": "string",
}


def register_model(model_name: str, version: str, status: str,
                   artifact_path: str, metadata: dict) -> None:
    """
    Register a new model version in the Foundry model registry dataset.
    Call after training completes and before submitting for evaluation.
    """
    from foundry.transforms import Dataset

    registry_dataset =
Dataset.get("/ml/readiness/model_registry/c_rating_predictor_registry")

    new_entry = pd.DataFrame([{
        "model_name": model_name,
        "version": version,
        "status": status,
        "artifact_path": artifact_path,
        "feature_set_version": metadata.get("feature_set_version"),
        "test_accuracy": metadata.get("test_accuracy"),
        "test_f1_macro": metadata.get("test_f1_macro"),
        "train_date_start": metadata.get("train_date_start"),
        "train_date_end": metadata.get("train_date_end"),
        "deployed_date": None,
        "retired_date": None,
        "approved_by": None,
        "notes": metadata.get("notes", ""),
    }])

    # Append to existing registry
    existing = registry_dataset.pandas()
    updated = pd.concat([existing, new_entry], ignore_index=True)

    with registry_dataset.transaction() as tx:
        tx.write_pandas(updated)

    print(f"Registered {model_name} version {version} as {status}.")
```

1. To promote a model to PRODUCTION, update its registry entry status and update the inference Transform to reference the new artifact path. Update the previous PRODUCTION version to RETIRED.
2. Maintain all retired model versions. Do not delete them. Deletion of model artifacts requires C2DAO authorization and is treated as a data destruction event.

7-3. Task: Implement Drift Detection

CONDITIONS: Model deployed to production. Inference Transform running on schedule. Feature dataset and prediction dataset accumulating historical records.

STANDARDS: Automated drift detection running on the same schedule as inference. Alerts generated to the data steward and MLE when drift thresholds are exceeded. All drift metrics logged to a monitoring dataset.

EQUIPMENT: Code Workspace; Foundry Transforms; production feature dataset; training baseline dataset; Pipeline Builder for alert configuration.

PROCEDURE:

1. Implement input feature drift detection using population stability index (PSI):

```
def compute_psi(expected: np.ndarray, actual: np.ndarray,
                n_bins: int = 10) -> float:
    """
    Population Stability Index (PSI).
    PSI < 0.10: no significant drift
    PSI 0.10–0.25: moderate drift — investigate
    PSI > 0.25: significant drift — retrain or halt
    """
    # Create bins from expected distribution
    breakpoints = np.nanpercentile(expected, np.linspace(0, 100, n_bins + 1))
    breakpoints = np.unique(breakpoints)  # Remove duplicates at distribution edges

    # Bin both distributions
    expected_counts = np.histogram(expected, bins=breakpoints)[0]
    actual_counts = np.histogram(actual, bins=breakpoints)[0]

    # Avoid division by zero
    expected_pct = (expected_counts + 0.0001) / len(expected)
    actual_pct = (actual_counts + 0.0001) / len(actual)

    psi = np.sum((actual_pct - expected_pct) * np.log(actual_pct / expected_pct))
    return psi


@transform_df(
    Output("/ml/readiness/monitoring/drift_metrics"),
    current_features=Input("/ml/readiness/features/unit_readiness_features"),
    baseline_features=Input("/ml/readiness/baselines/c_rating_predictor_v1_baseline"),
)
def compute_drift_metrics(current_features, baseline_features):
    """
    Compare current feature distribution to training baseline.
    Emit PSI for each feature. Alert if any PSI > 0.25.
    """
    feature_cols = [
        "personnel_fill_rate", "eoh_rate", "c_rating_30d_avg",
        "days_since_maint", "training_events_90d"
    ]
```

```
    # Use last 30 days of current data for drift comparison
    cutoff = pd.Timestamp.now() - pd.Timedelta(days=30)
    current_window = current_features[
        pd.to_datetime(current_features["report_date"]) >= cutoff
    ]

    results = []
    for col in feature_cols:
        psi = compute_psi(
            baseline_features[col].dropna().values,
            current_window[col].dropna().values,
        )
        status = "OK" if psi < 0.10 else ("WARN" if psi < 0.25 else "ALERT")
        results.append({
            "feature": col,
            "psi": psi,
            "status": status,
            "computed_at": pd.Timestamp.utcnow(),
            "window_days": 30,
            "window_n_rows": len(current_window),
        })

        if status == "ALERT":
            # This will appear in Transform logs and trigger pipeline monitoring
alerts
            print(f"[DRIFT ALERT] Feature {col}: PSI={psi:.4f} > 0.25. "
                  f"Notify data steward and MLE immediately.")

    return pd.DataFrame(results)
```

1. Implement prediction distribution drift (output drift):

```
@transform_df(
    Output("/ml/readiness/monitoring/prediction_drift"),
    current_preds=Input("/ml/readiness/predictions/c_rating_predictions"),

baseline_preds=Input("/ml/readiness/baselines/c_rating_predictor_v1_pred_baseline"),
)
def compute_prediction_drift(current_preds, baseline_preds):
    """
    Compare current prediction distribution to baseline prediction distribution.
    A shift in prediction distribution may indicate data drift, concept drift,
    or a change in the underlying operational reality being modeled.
    """
    cutoff = pd.Timestamp.now() - pd.Timedelta(days=30)
    current_window = current_preds[
        pd.to_datetime(current_preds["inference_timestamp"]) >= cutoff
    ]

    # Compare class distribution
    current_dist = current_window["predicted_c_rating"].value_counts(normalize=True)
    baseline_dist = baseline_preds["predicted_c_rating"].value_counts(normalize=True)
```

```
    comparison = pd.DataFrame({
        "baseline_rate": baseline_dist,
        "current_rate": current_dist,
    }).fillna(0)
    comparison["delta"] = comparison["current_rate"] - comparison["baseline_rate"]
    comparison["abs_delta"] = comparison["delta"].abs()
    comparison["computed_at"] = pd.Timestamp.utcnow()

    # Flag if any class shifts by more than 15 percentage points
    for c_rating, row in comparison.iterrows():
        if row["abs_delta"] > 0.15:
            print(f"[PREDICTION DRIFT] C{c_rating} rate shifted {row['delta']:+.3f} "
                  f"({row['baseline_rate']:.3f} -> {row['current_rate']:.3f}). "
                  f"Notify data steward.")

    return comparison.reset_index().rename(columns={"index": "c_rating"})
```

1. Configure Pipeline Builder monitoring alerts on the drift Transform outputs. Set alert thresholds: any ALERT status row in the drift_metrics dataset triggers an email to the MLE and data steward.

7-4. Task: Configure Retraining Triggers

CONDITIONS: Drift monitoring operational. Model version registry in place. Retraining conditions agreed with the mission owner.

STANDARDS: Documented retraining trigger conditions. Automated detection of trigger condition. Retraining does not automatically promote to production — it produces a CANDIDATE model that must pass evaluation and governance gates before promotion.

EQUIPMENT: Code Workspace; Foundry Transforms; drift monitoring dataset; model version tracking dataset; Pipeline Builder for alert routing.

PROCEDURE:

1. Define retraining triggers in the model card (see Chapter 9). Standard USAREUR-AF trigger categories:

Trigger Type	Condition	Response
Scheduled retraining	Quarterly, or after 90 days of new data	Retrain → CANDIDATE → evaluate → promote
Drift trigger	Any feature PSI > 0.25	Retrain → CANDIDATE → fast-track evaluation
Performance trigger	Test set accuracy drops below threshold when re-evaluated on last 30 days	Retrain → CANDIDATE → standard evaluation

Trigger Type	Condition	Response
Data schema trigger	Source dataset schema change	Halt inference → review feature pipeline → retrain
Operational event trigger	Major operational event (theater rotation, large exercise, mobilization)	Evaluate performance on post-event data → retrain if degraded

1. Implement an automated retraining check as a Foundry Transform:

```
@transform_df(
    Output("/ml/readiness/monitoring/retrain_recommendations"),
    drift_metrics=Input("/ml/readiness/monitoring/drift_metrics"),
    model_registry=Input("/ml/readiness/model_registry/c_rating_predictor_registry"),
)
def check_retrain_conditions(drift_metrics, model_registry):
    """
    Evaluate current monitoring signals against retrain trigger conditions.
    Outputs a recommendation record — does NOT trigger retraining automatically.
    The MLE reviews this output and initiates retraining manually.
    """
    recommendations = []

    # Check drift triggers
    alert_features = drift_metrics[drift_metrics["status"] == "ALERT"]
    if len(alert_features) > 0:
        recommendations.append({
            "trigger_type": "DRIFT",
            "trigger_detail": f"PSI ALERT on features:
{list(alert_features['feature'])}",
            "recommendation": "RETRAIN",
            "urgency": "HIGH",
            "created_at": pd.Timestamp.utcnow(),
        })

    # Check scheduled trigger (quarterly)
    prod_models = model_registry[model_registry["status"] == "PRODUCTION"]
    if len(prod_models) > 0:
        latest_prod = prod_models.sort_values("deployed_date").iloc[-1]
        days_deployed = (
            pd.Timestamp.now() - pd.to_datetime(latest_prod["deployed_date"])
        ).days
        if days_deployed > 90:
            recommendations.append({
                "trigger_type": "SCHEDULED",
                "trigger_detail": f"Model in production for {days_deployed} days (>90
day threshold)",
                "recommendation": "RETRAIN",
                "urgency": "NORMAL",
                "created_at": pd.Timestamp.utcnow(),
            })

    if not recommendations:
        recommendations.append({
```

```
            "trigger_type": "NONE",
            "trigger_detail": "No retrain triggers active",
            "recommendation": "MONITOR",
            "urgency": "NONE",
            "created_at": pd.Timestamp.utcnow(),
        })

    return pd.DataFrame(recommendations)
```

NOTE

Automated retraining without human review is not authorized on MSS. The retrain check Transform outputs a recommendation. A human MLE reviews it, initiates the retrain, and the resulting CANDIDATE model must pass the full evaluation and governance process before promotion. There is no auto-promote path.

CHAPTER 8 — OPERATIONAL USE CASES

8-1. Overview

BLUF: Three worked examples cover the primary USAREUR-AF ML use cases: unit readiness C-rating prediction, logistics demand forecasting, and OPDATA anomaly detection. Each example is operationally grounded, not academic.

These use cases are drawn from authorized USAREUR-AF ML applications. They illustrate the full pipeline: feature design, training approach, evaluation, and deployment. Adapt them — do not copy them verbatim. Every deployment requires its own feature set, evaluation, and governance approval.

8-2. Use Case: Unit Readiness C-Rating Prediction

Operational Problem: USAREUR-AF G3/G4 and brigade S3s must monitor readiness across hundreds of units. The volume of unit status reports makes manual trend analysis difficult. A predictive model that identifies units at risk of C-rating degradation 7-14 days before the next report allows targeted intervention before the degradation is reported.

Framing: Binary + ordinal classification problem. Predict the C-rating a unit will report in the next readiness report (7-14 days ahead), given current features. The operationally critical prediction is: "Will this unit degrade from C2 to C3 or worse?"

Feature set summary:

Feature	Operational Interpretation
personnel_fill_rate	Personnel strength relative to authorization — a leading indicator of degradation
eoh_rate	Equipment on-hand vs. required — drives P2/P3 readiness level
c_rating_30d_avg	Trend — is this unit's readiness stable, improving, or declining?
days_since_maint	Maintenance event recency — equipment age since last service
training_events_90d	Training tempo — high tempo predicts short-term degradation (wear), recovery thereafter

Model selection rationale: - Gradient Boosting Classifier (scikit-learn `GradientBoostingClassifier` or `XGBClassifier`) outperforms linear models on this feature set in USAREUR-AF testing. - Neural networks do not improve over GBM on this tabular feature set and are harder to explain to mission owners. - Logistic Regression serves as an interpretable baseline and is acceptable when F1 is within 3 points of GBM.

Evaluation criteria: - Weighted F1 >= 0.72 on temporally held-out test set. - Optimistic error rate (predicted more ready than actual) < 0.10. - Per-echelon accuracy spread < 0.15.

Deployment pattern: Batch inference daily at 0800Z. Predictions appended to unit_readiness_predictions dataset. Synced to Unit Ontology Object as model-backed property. Workshop readiness dashboard displays predicted vs. reported C-rating with confidence indicator.

Known limitations: - Model does not capture ad hoc operational events (unexpected deployments, injuries, urgent maintenance) that have not yet appeared in source data. - Predictions for newly stood-up units (less than 90 days of history) have reduced confidence — flag these in the dashboard. - Model scope: USAREUR-AF garrison-based units. Accuracy for units in rotational deployment status has not been validated.

8-3. Use Case: Logistics Demand Forecasting

Operational Problem: 21st TSC requires forward visibility into parts demand at Theater Distribution Center (TDC) and Brigade Support Battalion (BSB) level to position Class IX (repair parts) ahead of demand peaks. Current pull-based requisition processes result in stock-outs and emergency requisitions during exercise preparation periods. A 30-day demand forecast enables pre-positioning and reduces emergency requisition volume.

Framing: Time-series regression problem. For each National Stock Number (NSN) x location combination, forecast total requisition quantity over the next 30 days. Hierarchical structure: NSN-level forecast rolled up to Class IX category and location.

Feature set summary:

Feature	Operational Interpretation
demand_lag_7d	Last 7-day requisition count (same NSN, same location)
demand_lag_30d	Last 30-day requisition count
demand_lag_90d	Last 90-day requisition count — captures seasonal/exercise patterns
days_to_next_exercise	Days until the next scheduled exercise at this location
on_hand_qty	Current on-hand quantity — affects demand (lower on-hand → higher requisition rate)
nsn_category	Class IX sub-category (engines, tracks, electronics...)
location_type	TDC, BSB, FSB — echelon determines demand pattern

Training approach:

```
# Demand forecasting uses LightGBM with direct multi-step forecasting.
# For each NSN-location pair, features are constructed at the 7-day level
# and the target is total demand in the next 30 days.

# The critical design choice: use a single global model across all NSNs
# rather than per-NSN models. The global model generalizes across items
# with sparse individual histories (most NSNs have sparse demand).

import lightgbm as lgb

# Feature matrix: each row is one (NSN, location, week) observation
# with lag features and calendar context
# Target: sum of requisitions in next 30 days

model = lgb.LGBMRegressor(
    n_estimators=300,
    learning_rate=0.05,
    max_depth=6,
    num_leaves=63,
    subsample=0.8,
    colsample_bytree=0.8,
    objective="poisson",    # Poisson loss for count data (requisitions are counts)
    random_state=42,
)

model.fit(
    X_train, y_train,
    eval_set=[(X_val, y_val)],
    callbacks=[lgb.early_stopping(20), lgb.log_evaluation(50)],
)
```

Evaluation criteria: - MAE on 30-day forward window: <15% of mean demand for that NSN category. - No systematic under-forecast bias: mean signed error within ±5% of mean demand. - Precision at top-20 NSNs by forecasted demand: >= 0.70 (the 20 NSNs forecast to have highest demand should actually have high demand).

Deployment pattern: Weekly batch inference every Monday 0600Z. Forecast dataset published to TDC and BSB-level Workshop applications. Forecast vs. actual tracked in monitoring dataset. 21st TSC G4 is the mission owner; weekly review of forecast quality.

8-4. Use Case: OPDATA Anomaly Detection

Operational Problem: USAREUR-AF G2 analyzes pattern-of-life data across the European and African AOR to detect deviations that may indicate operational significance. Manual review of high-volume telemetry and activity data is not scalable. An anomaly detection system that surfaces statistically unusual observations for analyst review reduces the load on limited all-source analyst capacity.

Framing: Unsupervised anomaly detection. No labeled ground truth for "operational anomaly." The model surfaces observations whose feature vectors are statistically unusual relative to the recent baseline. Human analysts determine operational significance — the model performs triage, not assessment.

WARNING: ANALYST-IN-LOOP REQUIREMENT. Anomaly detection outputs are triage lists for analyst review, not operational alerts. An anomaly score is a statistical measure — it does not mean an event is operationally significant. Every anomaly flagged by the model must be reviewed by a qualified analyst before any operational action is taken. Automated propagation of anomaly flags to operational reporting systems without analyst review is not authorized.

Approach: Isolation Forest with operational context features

```
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler
import numpy as np
import pandas as pd

# --- Features for anomaly scoring ---
# Each row represents one entity-observation (e.g., one activity record)
# Features capture deviation from entity's own baseline (entity-relative features)
# to avoid flagging entities that are always unusual as anomalies

ANOMALY_FEATURES = [
    "activity_count_delta_7d_pct",   # % change in activity count vs 30-day baseline
    "new_location_flag",             # 1 if new location not seen in past 30 days
    "hour_of_day_deviation",         # Deviation from entity's typical activity hour
    "entity_gap_days",               # Days since last observed activity
    "co_occurrence_score",           # Unusual co-occurrence with other entities
]

# --- Train on baseline period (last 90 days of "normal" activity) ---
baseline_df = df[df["date"] < BASELINE_END_DATE][ANOMALY_FEATURES].dropna()

scaler = StandardScaler()
X_baseline_scaled = scaler.fit_transform(baseline_df)

model = IsolationForest(
```

```
    n_estimators=200,
    contamination=0.02,  # Expect ~2% anomaly rate — tune with analyst feedback
    random_state=42,
    n_jobs=-1,
)
model.fit(X_baseline_scaled)

# --- Score current period ---
current_df = df[df["date"] >= SCORING_START_DATE][ANOMALY_FEATURES].fillna(0)
X_current_scaled = scaler.transform(current_df)

# anomaly_score: lower (more negative) = more anomalous
# decision: -1 = anomaly, 1 = normal
anomaly_scores = model.score_samples(X_current_scaled)
anomaly_decisions = model.predict(X_current_scaled)

results = current_df.copy()
results["anomaly_score"] = anomaly_scores
results["anomaly_flag"] = (anomaly_decisions == -1).astype(int)

# Rank anomalies for analyst triage list
anomalies = results[results["anomaly_flag"] == 1].sort_values("anomaly_score")
print(f"Anomalies flagged: {len(anomalies)} out of {len(results)} observations
({len(anomalies)/len(results):.2%})")
print(f"Top 10 anomalies:\n{anomalies.head(10)}")
```

Evaluation criteria: - Precision at top-20 anomalies: >= 0.60 (based on analyst feedback labeling). - False positive rate: maintain analyst alert volume < 25 anomalies/day at normal operational tempo to prevent alert fatigue. - Analyst feedback loop: implement a rating mechanism in the Workshop triage interface; use ratings to tune contamination parameter quarterly.

Deployment pattern: Daily batch scoring 0400Z. Top-K anomalies surface in G2 analyst Workshop triage panel. Analyst marks each as "Significant" / "Noise" / "Pending." Feedback labels stored and used to evaluate model quarterly. Model retrained quarterly with analyst-labeled training data when sufficient labels accumulate.

CHAPTER 9 — MODEL GOVERNANCE

9-1. Overview

BLUF: Model governance on MSS is not bureaucratic overhead — it is the operational safety chain for AI-informed decisions. Every model that reaches production has passed documented gates. Every model in production is actively monitored. Every production incident is reported within 24 hours.

The governance framework is grounded in Army CIO Data Stewardship Policy (April 2, 2024), UDRA v1.1 (February 2025), and DoD AI Ethics Principles (2020). The USAREUR-AF C2DAO is the governance authority for all MSS ML models.

9-2. The Six-Gate Governance Model

```
GATE 1: USE CASE AUTHORIZATION
  ↓ C2DAO approves use case category (see Appendix B)
GATE 2: DESIGN REVIEW
  ↓ C2DAO + data steward review feature design, data sources, privacy assessment
GATE 3: EVALUATION ACCEPTANCE
  ↓ Mission owner reviews and accepts evaluation report; bias review completed
GATE 4: PRE-PRODUCTION REVIEW
  ↓ C2DAO reviews deployment architecture, Ontology impact, freshness design
GATE 5: PRODUCTION DEPLOYMENT APPROVAL
  ↓ C2DAO issues deployment authorization; MLE executes deployment
GATE 6: POST-DEPLOYMENT MONITORING CONFIRMATION
  ↓ Drift monitoring active; monitoring plan submitted to C2DAO within 30 days
```

No model progresses from one gate to the next without documented evidence that the gate criteria are met. The MLE is responsible for preparing the gate package. C2DAO and the mission owner review and approve.

9-2a. DDOF Phases 4–5 for ML Development

BLUF: DDOF Playbook v2.2 Phases 4 (Development) and 5 (Test & Evaluation) map directly to the ML lifecycle. MLEs execute feature engineering, model training, and validation within these phases. The MVP mandate is 30 days from Phase 1 — model complexity must not exceed what can be delivered, tested, and deployed within this timeline.

The table below maps DDOF phases to concrete ML activities and the gate output required to exit each phase.

DDOF Phase	ML Activity	Gate Output
Phase 4 (Development)	Feature engineering, model training, hyperparameter tuning	Functional model with documented architecture
Phase 5 (T&E)	Validation on holdout data, bias testing, adversarial testing, UAT with operational sponsor	Test report with accuracy/precision/recall metrics, sponsor sign-off

NOTE

The 30-day MVP mandate from Phase 1 is a hard constraint. Select the simplest model architecture that meets the operational threshold. A gradient boosting model delivered on day 28 is worth more than a neural architecture still in tuning on day 45. Complexity that jeopardizes the timeline is not engineering rigor — it is scope failure.

WARNING

Do not bypass Phase 5 T&E to meet the MVP timeline. A model that ships without holdout validation and sponsor sign-off is not an MVP — it is an unauthorized deployment. If the timeline is not achievable, escalate to C2DAO for scope reduction before cutting T&E.

Source: DDOF Playbook v2.2 (December 2025).

9-2b. ML Model as UDRA Data Product

BLUF: Per UDRA v1.1, a trained ML model and its inference outputs are data products subject to full VAULTIS-A governance. The MLE is responsible for ensuring every deployed model meets all eight dimensions before production deployment.

VAULTIS-A (as defined in the DDOF Playbook v2.2, extending the DoD Data Strategy) requires an 85% weighted average across all dimensions to pass the DDOF Phase 3 quality gate. The table below maps each dimension to its ML-specific application.

VAULTIS-A Dimension	ML Application
Visible	Model registered in ADC with description, version, and use case
Accessible	Inference API or batch output available to authorized consumers
Understandable	Model card documenting architecture, training data, and limitations
Linked	Lineage from training data through feature pipeline through model to outputs
Trusted	Validated accuracy metrics and sponsor sign-off on file
Interoperable	Standard output formats (Foundry dataset schema); API compatibility with downstream consumers
Secure	Classification marking applied to model artifact and all inference outputs
Auditable	Full training/inference logs retained; version history in model registry

NOTE

VAULTIS-A compliance is not a one-time checkpoint. Each dimension must remain true throughout the model's production lifecycle. A model that was Visible at deployment but is later removed from ADC without replacement is non-compliant. The quarterly model card currency check (Section 9-5) is the enforcement mechanism.

9-3. Task: Complete the Model Card

CONDITIONS: Model trained and evaluated. Evaluation report complete. Pre-production review scheduled.

STANDARDS: Model card complete, reviewed by the MLE's peer, and submitted to the C2DAO gate package. All required sections populated with non-placeholder content.

EQUIPMENT: Model evaluation report; training data documentation; model artifact metadata; C2DAO gate package template; peer reviewer identified.

PROCEDURE:

1. Prepare the model card using the following template. Adapt it to the specific model — no section should contain "TBD" or "N/A" without explanation:

```
# MODEL CARD — [MODEL NAME] v[VERSION]

## 1. Basic Information
- Model name:
- Version:
- Model type: [classification / regression / anomaly detection / forecasting]
- Date trained:
- Trained by: [name, unit]
- Approved use case: [from Appendix B]

## 2. Intended Use
- Primary operational purpose:
- Mission owner: [name, unit, contact]
- Authorized consumers: [list Workshop apps, reports, users]
- Out-of-scope uses: [explicit list of uses this model is NOT authorized for]

## 3. Data
- Training data source(s): [Foundry dataset paths]
- Training date range:
- Training data volume:
- Feature set version:
- Data steward contact: [name, unit]
- Known data limitations or gaps:

## 4. Model Details
- Algorithm:
```

```
- Hyperparameters: [key parameters used in production model]
- Feature columns: [list]
- Target variable:
- Output format: [class label / probability / score]

## 5. Evaluation Results
- Test accuracy / F1:
- Test date range:
- Cross-validation summary:
- Operational error analysis: [optimistic error rate, directional error patterns]
- Performance on subgroups: [by unit type, echelon, location as applicable]

## 6. Bias and Fairness
- Protected attributes assessed: [list]
- Assessment method: [demographic parity, equalized odds, etc.]
- Findings: [numerical results]
- Residual risk: [documented risk accepted by mission owner]
- Mission owner risk acceptance: [name, date]

## 7. Deployment
- Deployment pattern: [batch / ontology property / API]
- Inference schedule:
- Artifact location: [Foundry dataset path]
- Model version in production:

## 8. Limitations
- Known failure modes:
- Scope of valid use: [population, time period, operational context]
- Conditions under which model performance is unknown or degraded:

## 9. Monitoring
- Drift monitoring: [feature PSI thresholds, monitoring dataset path]
- Prediction distribution monitoring: [method and dataset path]
- Retraining triggers: [list from Chapter 7]
- Monitoring review cadence:
- Alert recipients: [names and contact info]

## 10. Incident Response
- Incident threshold: [what constitutes a model incident]
- Escalation path: MLE → data steward → C2DAO
- 24-hour reporting requirement: confirmed
- Rollback procedure: [steps to revert to prior model version]
```

1. Have the completed model card peer-reviewed by a second MLE or senior data engineer before submission. The reviewer must confirm that all evaluation metrics are accurately reported and that no sections contain placeholder content.
2. Submit the model card as part of the C2DAO gate package for the relevant governance gate (Gate 2 for design review; Gate 3 for evaluation acceptance; Gate 4 for pre-production review). Retain a copy in the project repository under `/docs/model_cards/`.

9-4. Task: Manage a Production Model Incident

CONDITIONS: Production model incident detected — either through drift monitoring alerts, mission owner notification, or direct observation of incorrect predictions.

STANDARDS: Incident classified within 2 hours. C2DAO notified within 24 hours. Root cause identified within 72 hours. Corrective action implemented or model rolled back before resuming inference.

EQUIPMENT: Code Workspace; model version tracking dataset; access to inference Transform configuration; C2DAO incident report template; direct communication path to data steward and C2DAO.

PROCEDURE:

1. Classify the incident immediately:

Class	Definition	Response Timeline
Class I — Critical	Model producing systematically incorrect predictions; downstream operational decision affected	Halt inference within 2 hours; C2DAO within 4 hours; rollback within 8 hours
Class II — Degraded	Model accuracy below accepted threshold; not yet impacting operational decisions	Investigate within 24 hours; C2DAO notification within 24 hours
Class III — Monitor	Drift alert but accuracy within threshold; operational impact unconfirmed	Investigate within 72 hours; C2DAO advisory within 72 hours

1. Halt inference for Class I incidents immediately. Do not wait for root cause analysis before halting.
2. Roll back to the last known-good model version:

```
# Rollback procedure — load prior PRODUCTION version from registry
from foundry.transforms import Dataset
import pandas as pd

registry = Dataset.get("/ml/readiness/model_registry/c_rating_predictor_registry")
df_registry = registry.pandas()

# Find last known-good model
prior_prod = (
    df_registry[df_registry["status"].isin(["PRODUCTION", "RETIRED"])]
    .sort_values("deployed_date")
    .iloc[-2]  # -2 = prior to current incident model
)

print(f"Rolling back to: {prior_prod['model_name']} v{prior_prod['version']}")
print(f"Artifact path: {prior_prod['artifact_path']}")
print(f"Originally deployed: {prior_prod['deployed_date']}")

# Update inference Transform to reference prior artifact path
# (This requires a code change — see batch_inference.py artifact path reference)
# After rollback, re-run inference Transform manually to validate output
```

1. File the incident report within 24 hours. Incident report template:

```
MODEL INCIDENT REPORT — [DATE] [TIME]Z

1. Model name and version:
2. Incident class: [I / II / III]
3. Time of detection:
4. Time of C2DAO notification:
5. Incident description: [what was observed]
6. Operational impact assessment: [were operational decisions affected?]
7. Immediate actions taken:
8. Root cause (preliminary): [fill within 72 hours]
9. Corrective action plan:
10. Model status at time of report: [inference halted / rolled back / monitoring]
11. MLE signature and contact:
```

1. Submit incident report to the C2DAO governance inbox within 24 hours of detection regardless of incident class.

9-5. Monitoring Obligations Summary

Obligation	Cadence	Owner	Escalation
Feature drift review	Daily (automated); weekly (manual review)	MLE	Data steward on ALERT
Prediction distribution review	Weekly	MLE	Mission owner if >15% shift
Model performance review (on recent labels)	Monthly	MLE	C2DAO if below threshold
Retrain recommendation review	Weekly	MLE	Initiate retrain if triggers active
Governance audit preparation	Quarterly	MLE + data steward	C2DAO review
Model card currency check	Quarterly	MLE	Update card if any field outdated

CHAPTER 10 — MODEL STUDIO

10-1. Overview

BLUF: Model Studio is Palantir's no-code model training workspace, generally available as of February 2026. It enables MLEs and qualified builders to train, evaluate, and deploy ML models without writing code. Model Studio expands the population of personnel who can produce governed ML models on MSS — but it does not replace code-based approaches for complex architectures.

Model Studio provides a guided interface for the core ML workflow: dataset selection, feature configuration, algorithm selection, hyperparameter tuning, evaluation, and deployment. Models trained in Studio follow the same governance gates (Chapter 9) and monitoring obligations as code-trained models. The tool does not reduce governance requirements — it reduces the engineering barrier to reaching those gates.

When to use Model Studio vs. code-based approaches:

Criterion	Model Studio	Code Workbook / Custom Transforms
Tabular classification and regression	Preferred	Not required unless custom preprocessing needed
Standard algorithms (gradient boosting, random forest, logistic regression, linear regression)	Supported	Also supported
Complex neural architectures (CNNs, transformers, custom layers)	Not supported	Required
Custom loss functions	Not supported	Required
Ensemble methods (stacking, blending, custom voting)	Not supported	Required
Rapid prototyping / proof of concept	Preferred — fastest path to a baseline model	Use for iteration after Studio baseline
Explainability requirements (SHAP, LIME)	Built-in feature importance; limited advanced explainability	Full control over explainability tooling
Non-tabular data (images, text, geospatial tensors)	Not supported	Required
Timestamp-aware feature engineering	Supported (improved timestamp handling, Feb	Full control

Criterion	Model Studio	Code Workbook / Custom Transforms
	2026 GA)	

NOTE

Model Studio is a starting point, not a ceiling. The recommended workflow for new use cases: build a baseline in Studio first, evaluate whether Studio's accuracy meets the operational threshold, and escalate to Code Workbook only if the threshold is not met or the architecture requires code. This approach satisfies the DDOF 30-day MVP mandate (Section 9-2a) by producing a deployable baseline in days rather than weeks.

10-2. Task: Train a Model in Model Studio

CONDITIONS: Feature dataset available in Foundry. MLE or qualified builder has Foundry Editor role. Use case authorized by C2DAO (Gate 1 complete).

STANDARDS: Model trained on a proper train/test split. Algorithm and hyperparameters documented. Evaluation metrics recorded. Model artifact saved to a governed Foundry dataset ready for deployment review.

EQUIPMENT: Foundry platform access; curated feature dataset; C2DAO use case authorization on file.

PROCEDURE:

1. Navigate to Model Studio from the Foundry left navigation panel. Select **New Model**.
2. Select the input dataset. Choose the curated feature dataset — not raw source data. The dataset must already have feature engineering applied (Chapter 3 procedures apply regardless of whether training is code-based or no-code).

WARNING

Model Studio does not replace feature engineering. A model trained on raw, uncurated data will produce unreliable results regardless of the algorithm selected. Complete Chapter 3 feature engineering procedures before entering Studio.

1. Configure the target variable. Select the column the model will predict. Studio will auto-detect whether the task is classification or regression based on the target column's data type.
2. Configure features. Select the feature columns to include. Studio displays summary statistics (mean, distribution, missing values) for each column. Exclude columns that are:
3. Identifiers (UICs, SSNs, names) — these are not features
4. Leaky features (columns derived from the target or available only after the prediction point)

5. High-cardinality categoricals without encoding strategy
6. Configure timestamp handling. For time-series operational data, specify the timestamp column. Studio's improved timestamp handling (GA February 2026) supports:
7. Temporal train/test splitting (prevents data leakage from future observations)
8. Time-aware feature windowing (rolling aggregates aligned to prediction time)
9. Inference-time timestamp alignment (model applies the correct temporal context at scoring time)

WARNING

For operational data with a time dimension, you MUST configure the timestamp column. Failure to do so results in random train/test splits that leak future data into training — producing artificially inflated accuracy that will not hold in production. This is the same temporal split requirement as Section 4-2 but enforced through the Studio interface.

1. Select the algorithm. Studio supports:
2. Gradient Boosted Trees (XGBoost) — default recommendation for tabular data
3. Random Forest
4. Logistic Regression (classification only)
5. Linear Regression (regression only)
6. Light GBM

Select the algorithm appropriate to the use case. When uncertain, start with Gradient Boosted Trees — it provides the strongest baseline for most MSS tabular use cases.

1. Configure hyperparameters. Studio provides sensible defaults. Adjust only with justification:
2. **Learning rate:** Default 0.1; reduce for noisy data
3. **Max depth:** Default 6; reduce if overfitting on small datasets
4. **Number of estimators:** Default 100; increase if validation loss is still decreasing
5. **Early stopping:** Enable early stopping with patience of 10 rounds
6. Start training. Studio displays real-time training progress including train/validation loss curves. Training runs execute on Foundry compute — no local resources consumed.
7. Review evaluation results. Studio provides:
8. Accuracy, precision, recall, F1 (classification) or RMSE, MAE, R^2 (regression)
9. Confusion matrix (classification)
10. Feature importance ranking
11. Train vs. validation performance comparison (overfitting detection)

Apply the same evaluation standards from Chapter 5. Studio-trained models are not exempt from operational threshold review.

DRAFT

1. Save the model. Studio saves the trained model artifact to a Foundry dataset. Record the artifact dataset path — this is required for the model card (Section 9-3) and deployment (Section 10-3).

10-3. Task: Deploy a Studio-Trained Model to Production

CONDITIONS: Model trained in Studio. Evaluation metrics meet operational thresholds. Gate 3 (Evaluation Acceptance) complete. Gate 4 (Pre-Production Review) scheduled.

STANDARDS: Model deployed via Studio's deployment interface or promoted to a production inference Transform. Inference outputs written to a governed Foundry dataset. Model registered in the model registry with version, artifact path, and deployment date.

EQUIPMENT: Trained model artifact in Foundry; model card (Section 9-3); C2DAO gate package; target Ontology Object Type (if deploying as Object property).

PROCEDURE:

1. From the Model Studio interface, select the trained model and choose **Deploy**.
2. Configure the deployment target:
3. **Batch inference:** Model scores a dataset on a schedule (daily, weekly). Output written to a Foundry dataset. This is the standard deployment pattern for most MSS use cases.
4. **Ontology property:** Model output mapped to an Object Type property visible in Workshop applications. Coordinate with the -30 builder for Object Type schema.
5. **Live inference (if available):** Model exposed as an API endpoint for real-time scoring. Requires additional C2DAO review for latency-sensitive use cases.
6. Configure the inference schedule. For batch deployments, set the schedule to match the data freshness SLA agreed with the data steward. Do not set inference frequency higher than the source data refresh rate — scoring stale data more frequently does not improve predictions.
7. Configure timestamp alignment for inference. Studio's improved timestamp handling ensures the model applies the correct temporal context at inference time. Verify that:
8. The inference input dataset includes the same timestamp column used during training
9. The model scores records using only data available at or before the prediction timestamp
10. Rolling window features are computed relative to the inference timestamp, not the current system time
11. Register the model in the model registry:

```
# Register Studio-trained model in the project model registry
# Run this in a Code Workspace after Studio deployment
from foundry.transforms import Dataset
import pandas as pd
from datetime import datetime
```

```
registry = Dataset.get("/ml/<project>/model_registry/<model_name>_registry")
df_registry = registry.pandas()

new_entry = pd.DataFrame([{
    "model_name": "<model_name>",
    "version": "<version>",
    "training_method": "model_studio",
    "algorithm": "<algorithm_selected>",
    "artifact_path": "<studio_artifact_dataset_path>",
    "deployed_date": datetime.utcnow().isoformat(),
    "status": "PRODUCTION",
    "evaluation_f1": <f1_score>,
    "evaluation_accuracy": <accuracy>,
    "notes": "Trained via Model Studio; timestamp-aware deployment"
}])

df_updated = pd.concat([df_registry, new_entry], ignore_index=True)
# Write updated registry back to Foundry
```

1. Submit the Gate 5 (Production Deployment Approval) package to C2DAO. The gate package is identical for Studio-trained and code-trained models. Studio does not bypass any governance gate.
2. Confirm drift monitoring is active (Chapter 7, Section 7-6). Studio-deployed models require the same monitoring obligations as code-deployed models — see Section 9-5.

10-4. Limitations and Escalation to Code

Model Studio accelerates the path from curated data to deployed model. It does not cover the full MLE capability set. The following scenarios require escalation to Code Workbook or custom Transforms:

Scenario	Why Studio Cannot Handle It	Code-Based Alternative
Custom neural architectures (CNN, LSTM, transformer)	Studio supports tree-based and linear models only	PyTorch or TensorFlow in Code Workspace (Section 4-4)
Custom loss functions	Studio uses standard loss functions per algorithm	Define custom loss in training script
Ensemble methods (stacking, blending)	Studio trains single models; no multi-model composition	scikit-learn StackingClassifier or custom ensemble code
Non-tabular inputs (images, free text, geospatial tensors)	Studio operates on tabular datasets only	Specialized libraries in Code Workspace
Advanced explainability (SHAP force plots, LIME, counterfactuals)	Studio provides feature importance only	SHAP/LIME libraries in Code Workspace (Section 5-5)
Feature engineering requiring custom Python logic	Studio consumes pre-engineered features; no inline transforms	Feature pipeline in Transforms (Chapter 3)

Scenario	Why Studio Cannot Handle It	Code-Based Alternative
Hyperparameter search beyond Studio's grid	Studio supports limited hyperparameter ranges	Optuna, Ray Tune, or manual grid search in Code Workspace

Escalation procedure: When a Studio-trained baseline does not meet the operational threshold and the limitation is architectural (not data quality), document the Studio baseline results in the model card, then develop a code-based alternative in a Code Workspace. The Studio baseline serves as the minimum performance benchmark — the code-based model must exceed it to justify the additional development time.

> **NOTE**
>
> A Studio-trained model that meets the operational threshold is the correct production choice even if a code-based model might achieve marginally higher accuracy. Engineering time spent exceeding an already-met threshold is time not spent on the next mission requirement. Ship the model that works; optimize only when the mission demands it.

DRAFT

APPENDIX A — MODEL GOVERNANCE CHECKLIST

Complete this checklist for every model before submitting a gate package. All items must be checked "YES" or explained. Any "NO" without documented exception blocks the gate.

Gate 1: Use Case Authorization

Item	Status	Notes
Use case falls within Appendix B authorized categories	YES / NO	
Use case does not involve prohibited applications (see UDRA v1.1 §7)	YES / NO	
C2DAO use case authorization memo received	YES / NO	
Mission owner identified and engaged	YES / NO	
Data steward for all source datasets identified	YES / NO	

Gate 2: Design Review

Item	Status	Notes
Feature definitions documented (FEATURE_DEFINITIONS.md)	YES / NO	

Item	Status	Notes
All source datasets have appropriate access authorization	YES / NO	
PII/sensitive data assessment completed	YES / NO	
Training-serving skew analysis documented	YES / NO	
Time leakage review completed	YES / NO	
Feature engineering code peer-reviewed	YES / NO	
C2DAO design review complete	YES / NO	

Gate 3: Evaluation Acceptance

Item	Status	Notes
Temporal train/val/test split used	YES / NO	
Test set never used during training or hyperparameter tuning	YES / NO	
All evaluation metrics computed on held-out test set	YES / NO	
Time-series cross-validation completed	YES / NO	
Confusion matrix and per-class metrics documented	YES / NO	
Operational error analysis completed (directional errors)	YES / NO	
SHAP or equivalent interpretability analysis completed	YES / NO	
Bias/fairness assessment completed for applicable models	YES / NO	
Mission owner has reviewed evaluation report	YES / NO	
Mission owner has signed off on performance thresholds	YES / NO	
Residual risk documented and accepted	YES / NO	

Gate 4: Pre-Production Review

Item	Status	Notes
Deployment pattern selected and documented	YES / NO	
Batch inference Transform implemented and tested in dev branch	YES / NO	
Model registry entry created (status: VALIDATED)	YES / NO	
Ontology impact assessment completed (if applicable)	YES / NO	

DRAFT

Item	Status	Notes
-30 builder coordinated for any Ontology changes	YES / NO	
Prediction freshness indicator implemented	YES / NO	
Human-in-the-loop constraint implemented in Workshop (if applicable)	YES / NO	
C2DAO pre-production review complete	YES / NO	

Gate 5: Production Deployment Approval

Item	Status	Notes
Model card complete (all sections populated)	YES / NO	
C2DAO production deployment authorization memo received	YES / NO	
Deployment executed on authorized schedule	YES / NO	
Post-deployment spot-check completed	YES / NO	
Model registry status updated to PRODUCTION	YES / NO	

Gate 6: Post-Deployment Monitoring Confirmation

Item	Status	Notes
Drift monitoring Transform deployed and running	YES / NO	
Monitoring dataset receiving daily records	YES / NO	
Alert routing configured (email/notification to MLE + data steward)	YES / NO	
Retraining trigger conditions documented	YES / NO	
Monitoring plan submitted to C2DAO within 30 days	YES / NO	
First monthly performance review scheduled	YES / NO	

APPENDIX B — APPROVED MODEL USE CASES (USAREUR-AF)

The following use case categories are pre-authorized for ML model development under the USAREUR-AF C2DAO ML framework. Individual projects within these categories still require Gate 1 authorization. Use cases not appearing in this list require a new C2DAO use case authorization before any development work begins.

Category	Description	Authorized Consumers	Notes
Unit Readiness Prediction	C-rating trend prediction and degradation risk flagging	G3, G4, brigade S3/S4	Personnel-affecting bias review required
Logistics Demand Forecasting	Class IX, Class VII demand forecasting at TDC/BSB level	21st TSC G4, BSB S4	
Parts Failure Prediction	Equipment component failure prediction for maintenance planning	USAREUR-AF G4, unit mechanics	Not for individual MOS assignment
OPDATA Anomaly Detection	Pattern-of-life anomaly scoring for analyst triage	G2, all-source analysts	Analyst-in-loop mandatory; no automated alerting
Training Event Optimization	Course demand forecasting, seat fill optimization	7th ATC G3	
Personnel Strength Forecasting	Organizational fill rate forecasting for force structure planning	G1, HRC coordination	PII-free feature set required; no individual predictions
Contract and Procurement Analytics	Spend analysis, contract performance risk scoring	Resource Management	Not for individual contractor performance
Supply Chain Visibility	In-transit item visibility prediction, delivery window estimation	21st TSC, DPW	
Fuel and Energy Demand Forecasting	POL demand forecasting at theater and installation level	G4, DPW	

Prohibited use cases (not authorized on MSS under any circumstances):

Prohibited Category	Reason
Individual performance prediction for evaluation or promotion	DoD AI Ethics; Army policy
Autonomous targeting or lethal force decision support	DoD Directive 3000.09; LOAC
Individual psychological profile or behavior prediction	Privacy Act; DoD policy
Biometric identification without authorized law enforcement nexus	Army policy
Predictions about individual service members' medical outcomes	HIPAA; Army privacy policy

DRAFT

APPENDIX C — ML QUICK REFERENCE

C-1. Standard Import Block

```
# Standard MLE import block for MSS Code Workspaces
import pandas as pd
import numpy as np
import matplotlib
matplotlib.use("Agg")  # Always headless in Foundry
import matplotlib.pyplot as plt
import seaborn as sns
import joblib
import json
import io
import mlflow

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import (
    classification_report, confusion_matrix, f1_score,
    mean_absolute_error, mean_squared_error
)
from sklearn.ensemble import GradientBoostingClassifier, IsolationForest

from foundry.transforms import Dataset as FoundryDataset
```

C-2. Temporal Split Pattern

```
def temporal_split(df: pd.DataFrame, date_col: str,
                   train_frac: float = 0.70,
                   val_frac: float = 0.15) -> tuple:
    """
    Split a DataFrame by time. Returns (train, val, test) DataFrames.
    Never use random splits for time-series operational data.
    """
    df = df.sort_values(date_col).reset_index(drop=True)
    n = len(df)
    train_end = int(n * train_frac)
    val_end = int(n * (train_frac + val_frac))
    return df.iloc[:train_end], df.iloc[train_end:val_end], df.iloc[val_end:]
```

C-3. Model Artifact Save/Load Pattern

```
def save_model_to_foundry(model, metadata: dict, dataset_path: str) -> None:
    """Save a serialized model and metadata to a Foundry dataset."""
    buf = io.BytesIO()
```

```
    joblib.dump(model, buf)
    dataset = FoundryDataset.get_or_create(dataset_path)
    with dataset.transaction() as tx:
        with tx.write_file("model.joblib") as f:
            f.write(buf.getvalue())
        with tx.write_file("metadata.json") as f:
            json.dump(metadata, f)


def load_model_from_foundry(dataset_path: str) -> tuple:
    """Load a serialized model and metadata from a Foundry dataset."""
    dataset = FoundryDataset.get(dataset_path)
    with dataset.open_file("model.joblib", "rb") as f:
        model = joblib.load(f)
    with dataset.open_file("metadata.json", "r") as f:
        metadata = json.load(f)
    return model, metadata
```

C-4. PSI Drift Detection

```
def compute_psi(expected: np.ndarray, actual: np.ndarray, n_bins: int = 10) -> float:
    """PSI < 0.10: OK. 0.10-0.25: WARN. > 0.25: ALERT."""
    bp = np.unique(np.nanpercentile(expected, np.linspace(0, 100, n_bins + 1)))
    exp_pct = (np.histogram(expected, bins=bp)[0] + 0.0001) / len(expected)
    act_pct = (np.histogram(actual, bins=bp)[0] + 0.0001) / len(actual)
    return float(np.sum((act_pct - exp_pct) * np.log(act_pct / exp_pct)))
```

C-5. MLflow Run Pattern

```
def log_training_run(run_name: str, params: dict, metrics: dict,
                     model, artifact_name: str = "model") -> str:
    """Log a complete training run to MLflow. Returns the run ID."""
    with mlflow.start_run(run_name=run_name) as run:
        mlflow.log_params(params)
        mlflow.log_metrics(metrics)
        mlflow.sklearn.log_model(model, artifact_path=artifact_name)
        return run.info.run_id
```

C-6. Evaluation Report Pattern

```
def generate_evaluation_report(y_true, y_pred, y_proba,
                                class_names: list, output_dir: str = ".") -> dict:
    """Generate and save a standard evaluation report package."""
    report = classification_report(y_true, y_pred,
                                    target_names=class_names, output_dict=True)

    # Confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    fig, ax = plt.subplots(figsize=(6, 5))
```

```
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
                xticklabels=class_names, yticklabels=class_names)
    ax.set_xlabel("Predicted"); ax.set_ylabel("Actual")
    plt.tight_layout()
    plt.savefig(f"{output_dir}/confusion_matrix.png", dpi=150)
    plt.close()

    return {
        "classification_report": report,
        "confusion_matrix": cm.tolist(),
        "accuracy": report["accuracy"],
        "macro_f1": report["macro avg"]["f1-score"],
        "weighted_f1": report["weighted avg"]["f1-score"],
    }
```

C-7. Common Pitfalls Reference

Pitfall	Symptom	Prevention
Time leakage	Unrealistically high train accuracy; poor real-world performance	Always use `.shift(1)` on lagged features; temporal train/test split
Training-serving skew	Model performs well in eval, poorly in production	Feature Transform code identical between training and inference paths
Silent null imputation	Model learns imputation artifacts as signal	Log null counts before and after imputation; validate in Transform
Stale model in production	Accuracy degrades over time without alerts	Drift monitoring required before deployment
Class imbalance ignored	High accuracy but poor recall on rare classes	Always report per-class metrics; use weighted F1 as primary metric
Random split on time-series data	Optimistic test results; model memorizes future	Temporal split required; document split method in model card
Unbounded compute usage	Runaway training job exhausts cluster	Set training timeouts; use early stopping; right-size compute profile

APPENDIX D — PROFESSIONAL READING LIST

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

Source	Title	Date	Relevance
Military Review	"The Military Needs Frontier Models"	Aug 2025	Large-scale ML models
Army AL&T	"Commoditizing AI/ML Models"	2024-25	ML model lifecycle management
Army Sustainment	"Predictive Logistics: Reimagining Sustainment on the 2040 Battlefield"	Winter 2025	ML in predictive logistics
Military Review	"Transforming the Multidomain Battlefield with AI"	2024	ML for object detection

GLOSSARY

Anomaly Detection — An ML task that identifies observations that deviate significantly from a learned baseline distribution. Used for OPDATA triage in MSS. Outputs are triage scores, not operational assessments.

Artifact — A file produced by an ML workflow: a serialized model, evaluation report, SHAP plot, or metadata JSON. Artifacts are versioned and stored in Foundry datasets.

Batch Inference — Running a trained model against a full dataset on a schedule to produce a complete set of predictions. The standard MSS deployment pattern.

Bias Assessment — Systematic analysis of model performance across demographic, organizational, or geographic subgroups to detect disparate impact. Required for all personnel-affecting models.

C-Rating — The US Army unit readiness rating system: C1 (fully capable), C2 (mostly capable), C3 (marginally capable), C4 (not capable), C5 (not assessed). The standard target variable for unit readiness prediction models.

C2DAO — USAREUR-AF Command and Control Data Architecture Office. The theater-level governance authority for all MSS data products, including ML models. No ML model deploys to production without C2DAO authorization.

Concept Drift — A change in the statistical relationship between features and the target variable over time, as opposed to data drift (a change in feature distributions). Concept drift renders a model stale even if feature distributions are unchanged.

Contamination Parameter — In Isolation Forest, the expected proportion of anomalies in the dataset. Tune with analyst feedback; too high produces excessive false positives; too low misses genuine anomalies.

Cross-Validation — A model evaluation technique that trains and evaluates on multiple data splits. For time-series operational data, walk-forward (TimeSeriesSplit) cross-validation is required.

Data Drift — A change in the statistical distribution of input features over time. Detected via PSI. A PSI > 0.25 is a retrain trigger.

Early Stopping — A regularization technique that halts training when validation performance stops improving. Prevents overfitting and controls training runtime.

Experiment Tracking — Systematic logging of hyperparameters, metrics, and model artifacts for each training run. MLflow is the standard experiment tracker on MSS.

Feature Engineering — The process of transforming raw operational data into numeric or categorical input vectors for ML models. The highest-leverage step in the ML lifecycle.

Feature Importance — A measure of how much each input feature contributes to a model's predictions. Required for model interpretability in operational contexts. Computed via SHAP for tree models.

Feature Leakage — The unintentional inclusion of information that would not be available at prediction time in the training feature set. Produces optimistic train/test metrics and models that fail in production.

Feature Store — A shared repository of precomputed, versioned feature datasets accessible to multiple models or analysts. Reduces redundant feature computation and ensures consistency.

Feature Pipeline — A Foundry Transform or sequence of Transforms that reads raw data and produces the feature dataset consumed by model training and inference.

Gradient Boosting — An ensemble ML algorithm that builds a sequence of weak learners (typically decision trees) where each learner corrects the errors of its predecessor. The primary algorithm for tabular MSS use cases (scikit-learn GBM, LightGBM, XGBoost).

Human-in-the-Loop (HITL) — A design constraint requiring that a qualified human operator reviews and approves model outputs before they take effect operationally. Mandatory for all MSS ML deployments.

Hyperparameter — A configuration setting of a model that is set before training and not learned from data (e.g., learning rate, tree depth, number of estimators). Tuned via Optuna or grid search.

Idempotent Transform — A Foundry Transform that produces the same output regardless of how many times it is run against the same input. Required for all production inference Transforms.

Inference — Running a trained model against new, unlabeled data to produce predictions. Distinct from training.

Isolation Forest — An unsupervised anomaly detection algorithm that isolates anomalies by randomly partitioning the feature space. Standard MSS approach for OPDATA anomaly triage.

LightGBM — A gradient boosting framework optimized for speed and large datasets. Preferred for demand forecasting use cases with sparse NSN-level data.

MLflow — An open-source platform for ML experiment tracking, model versioning, and artifact management. The standard experiment tracker in MSS Code Workspaces.

 DRAFT — UNOFFICIAL

MLOps — Machine Learning Operations: the set of practices for reliably deploying, monitoring, versioning, and maintaining ML models in production. Encompasses model registry, drift detection, retraining, and incident response.

Model Card — A structured document describing a model's intended use, training data, evaluation results, limitations, and monitoring requirements. Required for all production MSS deployments.

Model Registry — A Foundry dataset containing metadata about all trained model versions for a given use case: version, status, artifact path, evaluation metrics, and deployment history.

NSN — National Stock Number. A 13-digit identifier for items in the US military supply system. The entity-level identifier for logistics demand forecasting models.

Ontology-Backed Property — An Object Type property in the Foundry Ontology whose value is computed from a model's output and automatically synced from a prediction dataset. Allows model outputs to appear natively in Workshop applications.

Optuna — A hyperparameter optimization framework using Bayesian search and pruning. The standard hyperparameter tuner for MSS ML workflows.

Overfitting — A condition in which a model performs well on training data but poorly on new data because it has learned noise rather than signal. Detected by a large gap between training and validation metrics.

Population Stability Index (PSI) — A statistical measure of the shift in a feature's distribution between two time periods. PSI < 0.10: stable; 0.10–0.25: moderate drift; > 0.25: significant drift requiring investigation.

PyTorch — An open-source deep learning framework. Used for sequence models (LSTM) and neural architectures where GBM methods are insufficient.

Retraining Trigger — A documented condition that initiates a model retraining workflow: drift alert, scheduled interval, performance degradation, or operational event. Retraining produces a CANDIDATE model, not an automatic production update.

SHAP (SHapley Additive exPlanations) — A game-theoretic method for explaining individual model predictions by assigning each feature an importance value for each prediction. The standard interpretability method for MSS tree models.

Training-Serving Skew — A discrepancy between the feature engineering logic used during training and the feature engineering logic used during inference. A primary cause of silent model degradation in production.

UDRA — Unified Data Reference Architecture, version 1.1 (February 2025). The Army's authoritative reference architecture for enterprise data management. ML models are data products subject to UDRA governance.

UIC — Unit Identification Code. The standard identifier for US Army units. The entity-level key for readiness prediction models.

DRAFT

Walk-Forward Validation — A time-series cross-validation method that simulates real operational conditions by training on historical data and validating on the subsequent time period, advancing the window forward for each fold.

Weighted F1-Score — A classification metric that computes the F1-score for each class and weights by class support. The standard primary metric for imbalanced multi-class problems such as C-rating prediction.

SL 4M, Maven Smart System (MSS) Machine Learning Engineer Technical Manual Headquarters, United States Army Europe and Africa, Wiesbaden, Germany Reference: learn-data.armydev.com

DoD and Army Strategic References:

- **DoD Responsible AI Strategy & Implementation Pathway (June 2024 update)** — DoD framework for responsible AI development, testing, and fielding
- **DoD AI Cybersecurity Risk Management Guide (CDAO)** — Risk management guidance for AI systems in DoD environments
- **DoD Directive 3000.09, Autonomy in Weapon Systems (January 2023 update)** — Policy on autonomous and semi-autonomous functions in weapon systems

DRAFT