Final Report

This report documents a data curation workflow for NASA's C-MAPSS (Commercial Modular Aero-Propulsion System Simulation) turbofan engine degradation dataset, demonstrating end-to-end application of data curation principles discussed in our CS598 class. The project implements all six stages of the USGS Science Data Lifecycle Model (SDLM) and covers all M1-M15 requirements listed in the project specifications on coursera.

0. Revised Scope

After thoroughly reviewing the M1 - M15, I am going to remove the supplementary data sources that would add the dimension tables to the project scope. I initially thought that this project would be heavily focused on the data cleaning and data pipeline module 6, but as the semester has progressed I have realized that this is just one required step in this project. Reducing the scope of the project will allow me to better focus on the requirements for the project and spend more time on each deliverable.

Module	Requirement	Report Section	Notebook Section
M1	Data lifecycle models	Sections 1 - 7	Entire Notebook
M2	Ethical, legal, policy constraints	Section 1.2	N/A (covered in report)
M3 - M5	Data models & abstractions	Section 4.1	Section 2.3
M6	Data cleaning (quality assurance)	Section 3.1.2 - 3.1.3	Section 3.1, 3.2, 3.3
M7	What is Data?	Section 7.1.1	N/A (covered in report)
M8	Metadata & documentation	Section 3.1.1	Section 2.3
M9	Identity & identifier systems	Section 5	Section 3.3
M10	Preservation	Section 5	N/A (covered in report)
M11	Standards & best practices	Section 7.1.2	N/A (covered in report)
M12	Workflow automation & reproducibility	Section 5	Section 3.3
M13	Data practices (sharing, reuse, reproducibility)	Section 5	Section 3.3, 3.4
M15	Dissemination & communication	Section 6	Section 3.4

1. SDLM Planning Stage

1.1 Lifecycle and Data Model (M1)

In the Plan stage of the USGS Science Data Lifecycle Model (SDLM), I am defining the datasets and also the methods for organizing them. I am selecting both an organizational framework (SDLM) and a technical structure (Star Schema Data Model) to guide how raw data will be curated, integrated, and prepared for analysis in this project.[1] [2]

More specifically, the SDLM provides an organizational framework for end-to-end data stewardship, ensuring that activities such as Plan, Acquire, Process, Analyze, Preserve, and Publish are performed in a transparent and reproducible way.[1]

In contrast, the Star Schema Model is a technical design that defines how data is structured to support efficient storage and usage.[2] The Star Schema is an implementation of the Relational Model that was discussed in class.

1.2 Data & Ethics (M2)

Since this project uses simulated data (C-MAPSS) and public, de-identified data (FAA, ASRS), there are no risks associated with Personally Identifiable Information (PII) and human subjects.

Risk Assessment: The primary risk when synthesizing the data models (linking simulated engine units to real FAA/ASRS data) is misrepresentation. I plan to mitigate this by emphasizing that the linking of datasets will be purely conceptual and clearly documented as a demonstration of technical capability, not as a reflection of real-world provenance.

Compliance: The data selection itself adheres to compliance by avoiding classified or private military data, ensuring the entire project can be openly shared and peer-reviewed.

2. SDLM: Acquire Stage

I have completed this stage by downloading the source data from the relevant websites.

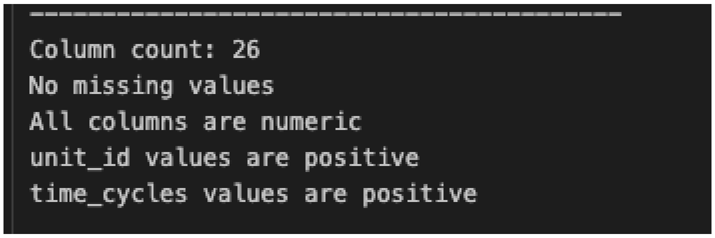
Dataset	Acquisition Activity	Data Format Notes
C-MAPSS	Downloaded the four degradation sub-datasets (FD001-FD004) from the NASA data portal.	The data are provided as a text file with 26 columns of numbers, separated by spaces.[3]

3. SDLM: Process Stage

The **Process** stage prepares the collected data for analysis and transforms the raw files into the integrated Star Schema. I will be using a dataframe to view, store, and manipulate the data in this stage.

3.1.1 C-MAPSS Fact Table Creation (M8)

The C-MAPSS data (columns 1-26) will undergo several transformations to prepare it as the core Fact Table:

- **Initial Loading:** I uploaded the C-MAPSS text files into a dataframe.
- **Schema Validation:** I performed the following checks on the schema of the fact table and all these checks passed without the need to adjust the fact table.:
    - Verified 26-column structure
    - Checked for missing values
    - Validated numeric data types
    - Ensured positive unit_id and time_cycles values
    - 

```
----------------------------------------
Column count: 26
No missing values
All columns are numeric
unit_id values are positive
time_cycles values are positive
```

- **Metadata Mapping (M8):** The author mentioned that each of the four datasets comes with the metadata below, which represents different experimental conditions and failure modes.[3]

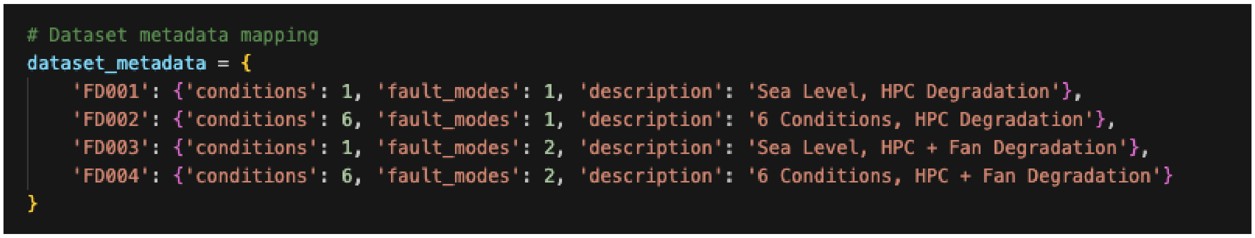
Data Set	Train Trajectories	Test Trajectories	Conditions	Fault Modes
FD001	100	100	ONE (Sea Level)	ONE (HPC Degradation)
FD002	260	259	SIX	ONE (HPC Degradation)
FD003	100	100	ONE (Sea Level)	TWO (HPC Degradation, Fan

				Degradation)
FD004	248	249	SIX	TWO (HPC Degradation, Fan Degradation)

I utilized this table to create 3 metadata columns and I then created an additional 3 metadata columns to enhance the usage of the Fact Table.

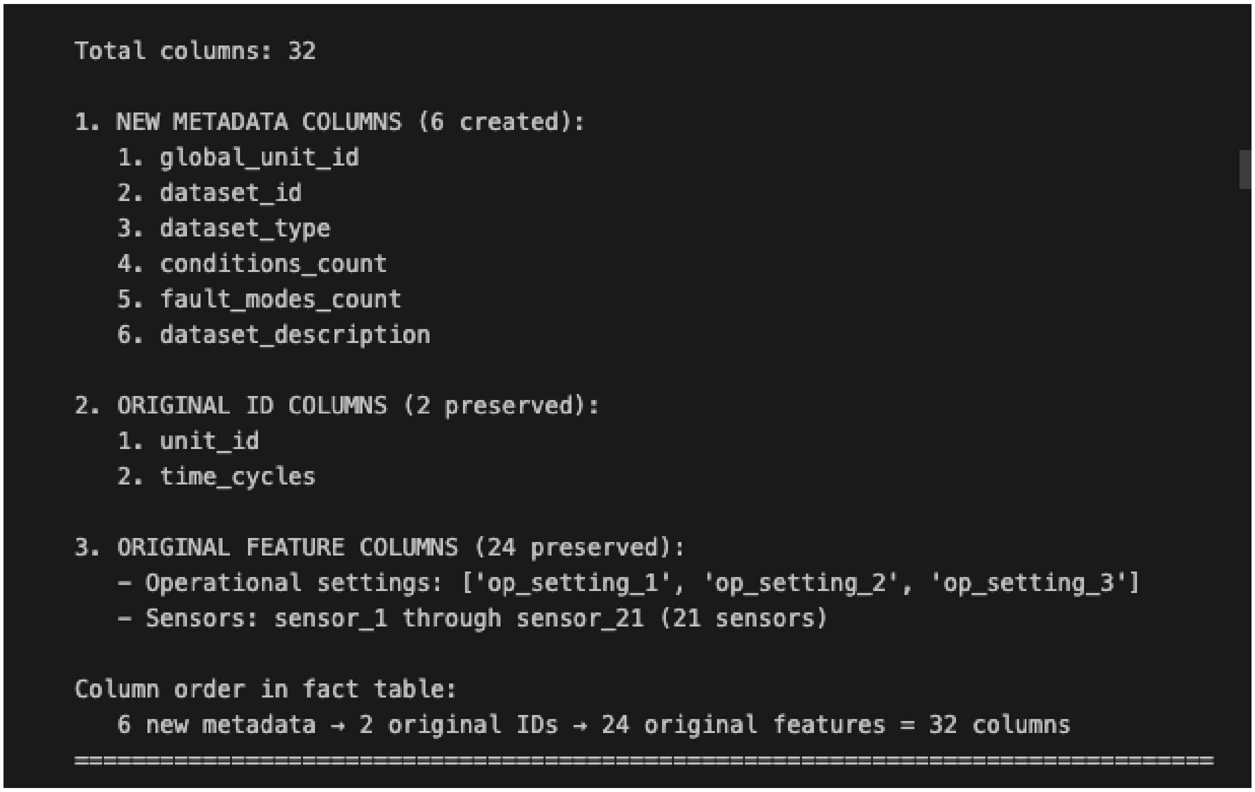
- Global_unit_id (Primary Key)
  - Purpose: Unique identifier across all datasets
  - Format: "{dataset_id}_{unit_id}" or "{dataset_id}_test_{unit_id}"
  - Examples: "FD001_1", "FD002_test_5"
  - Why needed: Original "unit_id" values overlap across datasets (e.g., unit 1 exists in all 8 files)
- Dataset_id
  - Purpose: Identifies which sub-dataset (FD001-FD004) the record belongs to
  - Values: "FD001", "FD002", "FD003", "FD004"
  - Usage: Enables filtering by condition complexity
- Dataset_type
  - Purpose: Distinguishes training vs. test data
  - Values: "train", "test"
  - Usage: Critical for train/test split when I create the RUL variable.
- Conditions_count
  - Purpose: Number of distinct operating conditions in the dataset
  - Values: “1” (FD001, FD003) or "6" (FD002, FD004)
  - Usage: Indicates complexity of flight variation
- Fault_modes_count
  - Purpose: Number of fault modes present in the dataset
  - Values: "1" (FD001, FD002) or "2" (FD003, FD004)
  - Usage: Single fault = HPC degradation only; Dual fault = HPC + Fan degradation
- Dataset_description
  - Purpose: Summary of experimental conditions
  - Format: ""{Operating Conditions}, {Fault Types}""
  - Examples: "Sea Level, HPC Degradation"", ""6 Conditions, HPC + Fan Degradation""

Below is the data dictionary that I created to add this metadata to the dataframe.



```
# Dataset metadata mapping
dataset_metadata = {
    'FD001': {'conditions': 1, 'fault_modes': 1, 'description': 'Sea Level, HPC Degradation'},
    'FD002': {'conditions': 6, 'fault_modes': 1, 'description': '6 Conditions, HPC Degradation'},
    'FD003': {'conditions': 1, 'fault_modes': 2, 'description': 'Sea Level, HPC + Fan Degradation'},
    'FD004': {'conditions': 6, 'fault_modes': 2, 'description': '6 Conditions, HPC + Fan Degradation'}
}
```

Below is the structure of the Fact Table dataframe with metadata added.

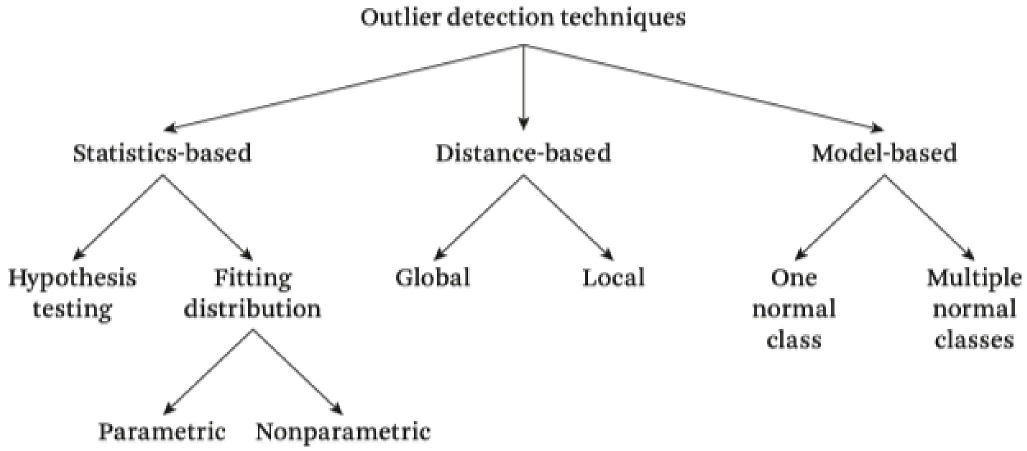


```
Total columns: 32

1. NEW METADATA COLUMNS (6 created):
   1. global_unit_id
   2. dataset_id
   3. dataset_type
   4. conditions_count
   5. fault_modes_count
   6. dataset_description

2. ORIGINAL ID COLUMNS (2 preserved):
   1. unit_id
   2. time_cycles

3. ORIGINAL FEATURE COLUMNS (24 preserved):
   - Operational settings: ['op_setting_1', 'op_setting_2', 'op_setting_3']
   - Sensors: sensor_1 through sensor_21 (21 sensors)

Column order in fact table:
   6 new metadata → 2 original IDs → 24 original features = 32 columns
================================================================================
```

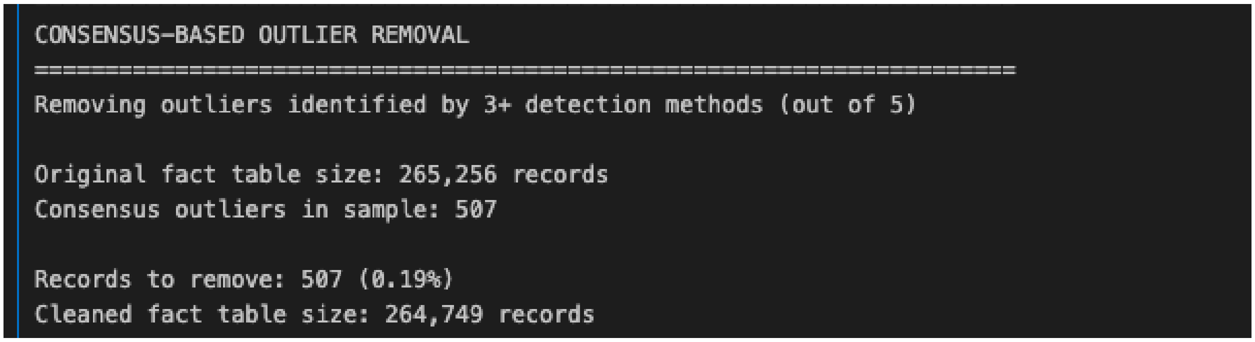
3.1.2 Outlier Removal (M6)[6]

I used both statistics and distance based approaches for outlier detection. I did not use model-based because there was no independent variable mapped to the training data and I needed labeled data sets for model-based outlier detection. I honestly didn't know there were this many options to test for outliers so this will be a boon for future pipelines. I generally always used statistics based outlier detections. I ended up following the 5 approaches to outlier detection below and in order for an outlier to be removed it had to have 2 of these methods deem it as an outlier. This resulted in only 566 or 0.21% of records falling into the outlier category. I excluded the op_setting columns from the outlier analysis because they are controlled experimental inputs and not subject to sensor noise contamination the author mentioned in the dataset. The 5 approaches I used are below for evaluating outliers in the 21 sensor columns:

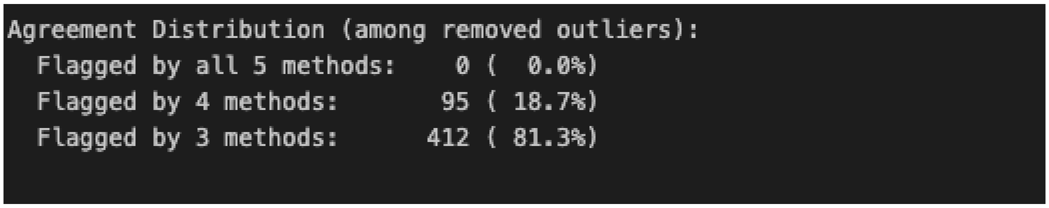
- Statistics Based
  - Multivariate analysis - there are 21 sensors so I felt I should try to look at these as groupings.
  - Robust Statistics: I used the median and MAD (Median Absolute Deviation) for 50% breakdown point
  - Z-Score Method: I used the z-score to Identify points beyond 3 standard deviations from the mean
- Distance Based
  - Local Outlier Factor (LOF): I used LOF to detect outliers based on local density deviation
  - Isolation Forest: I used isolation forest to use random partitioning to isolate anomalies



[6]



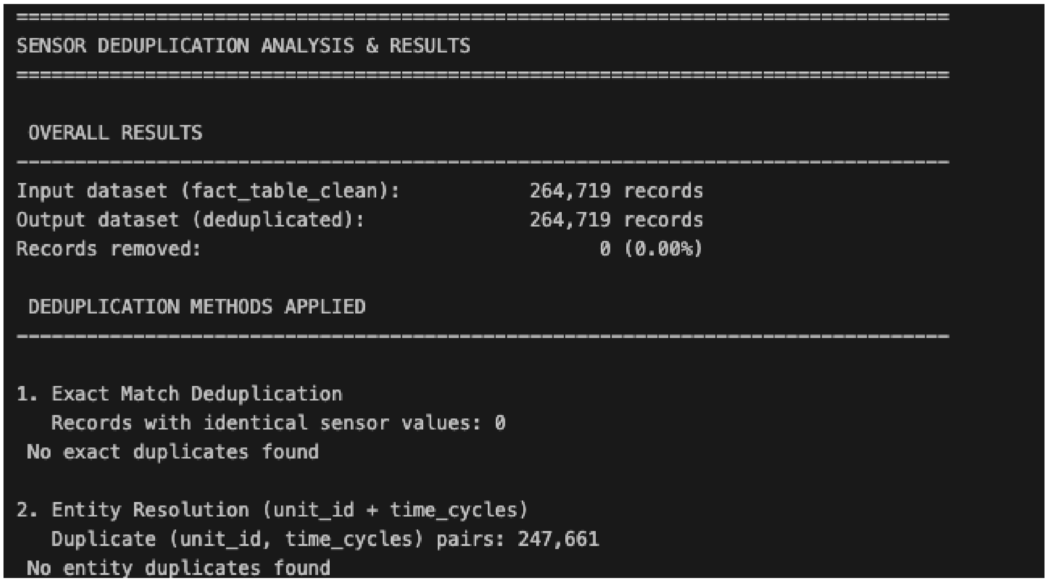
```
CONSENSUS-BASED OUTLIER REMOVAL
======================================================================
Removing outliers identified by 3+ detection methods (out of 5)

Original fact table size: 265,256 records
Consensus outliers in sample: 507

Records to remove: 507 (0.19%)
Cleaned fact table size: 264,749 records
```



```
Agreement Distribution (among removed outliers):
  Flagged by all 5 methods:    0 (  0.0%)
  Flagged by 4 methods:       95 ( 18.7%)
  Flagged by 3 methods:      412 ( 81.3%)
```

3.1.3 Data Deduplication (M6)

Data deduplication is the process of identifying distinct records that refer to the same real-world entity. Although the C-MAPSS time-series data is simulated the author did mention that there is a lot of noise, which held true with the data deduplication testing where I found out that individual sensors had 90% duplicate values. For example, Sensors 1, 16, and 19 had only 6, 2, and 2 respective unique values! However, unique value analysis by sensor is column based and is not true deduplication which is done at the row level and is across all columns in the fact table. As a result there was no *exact match* or *entity resolution deduplication* where sensors had identical records and therefore no record consolidation. [6]



```
================================================================================
SENSOR DEDUPLICATION ANALYSIS & RESULTS
================================================================================

 OVERALL RESULTS
--------------------------------------------------------------------------------
Input dataset (fact_table_clean):          264,719 records
Output dataset (deduplicated):             264,719 records
Records removed:                                 0 (0.00%)

 DEDUPLICATION METHODS APPLIED
--------------------------------------------------------------------------------

1. Exact Match Deduplication
   Records with identical sensor values: 0
 No exact duplicates found

2. Entity Resolution (unit_id + time_cycles)
   Duplicate (unit_id, time_cycles) pairs: 247,661
 No entity duplicates found
```

3.1.4 C-MAPSS Data Transformation (M12)

3.1.4.1 The Actual Transformation (M12)

The data transformation methodology implemented in this project follows the framework described by Ilyas and Chu in Data Cleaning.[6] Their approach consists of both syntactic and semantic transformations.

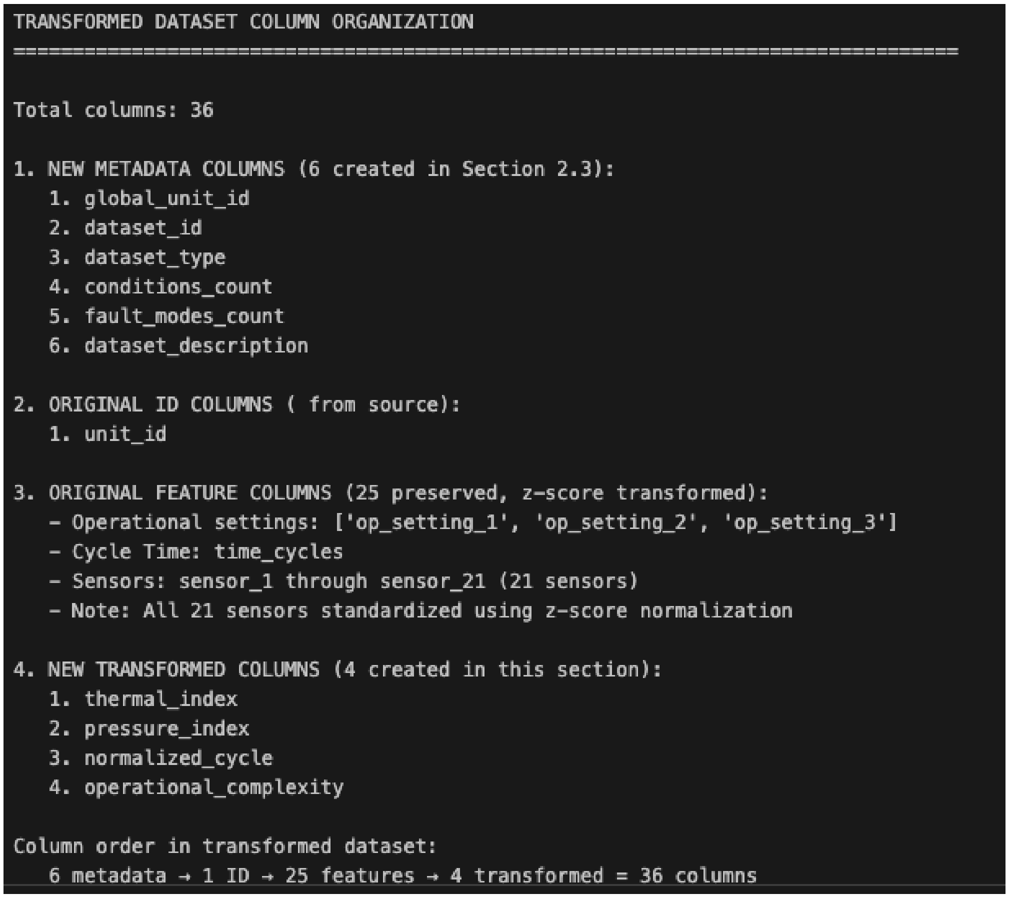
1. Semantic Transformations: Operations that embed domain understanding into the data, such as deriving health indicators, modeling degradation trends, and normalizing

operational conditions. These transformations require domain expertise. I performed the following semantic transformations based on a hypothesis of the dataset and my domain knowledge. These semantic transformations are inferred from correlation analysis and general turbofan engine domain knowledge rather than explicit documentation in the C-MAPSS dataset:

- Thermal Index (Hypothesis): I aggregated the mean of sensors 2, 3, and 4 to capture overall thermal health. These sensors were selected based on their correlated behavior patterns observed during exploratory analysis, suggesting they likely measure temperature-related parameters at different engine stages. The mean aggregation provides a single thermal health indicator representing the engine's thermodynamic state.
- Pressure Index (Hypothesis): I aggregated the mean of sensors 7, 8, and 9 to represent pressure system health. Similar to the thermal index, these sensors exhibit strong inter-correlation, indicating they may capture pressure-related measurements across the compression system.
- Normalized Cycle: I scaled time cycle position to [0,1] range for relative lifecycle positioning. This transformation converts absolute cycle counts into a normalized position within each engine's operational life, enabling direct comparison across engines with different total lifespans. A value of 0 represents the beginning of life, while values approaching 1 indicate end-of-life conditions.
- Operational Complexity (Hypothesis): I combined the magnitude of operational settings 1 and 2 to quantify flight condition complexity. This feature captures the intensity of operating conditions by adding the absolute values of the primary operational settings (abs(op_setting_1) + abs(op_setting_2)). Higher values indicate more demanding flight conditions (e.g., high altitude, high throttle), which accelerate component degradation. This metric serves as a proxy for operational stress on the engine.

2. Syntactic Transformations: Operations that modify data structure without requiring domain knowledge. Examples include normalization, scaling, time-series reformatting, and statistical feature generation. These transformations ensure data consistency and analytical readiness.
    - I applied Z-score standardization to all 21 sensor columns (mean=0, std=1)
    - I ensured uniform data scaling across sensors, improving feature comparability

The final transformed dataset contains 36 total columns as referenced below.



```
TRANSFORMED DATASET COLUMN ORGANIZATION
================================================================================

Total columns: 36

1. NEW METADATA COLUMNS (6 created in Section 2.3):
   1. global_unit_id
   2. dataset_id
   3. dataset_type
   4. conditions_count
   5. fault_modes_count
   6. dataset_description

2. ORIGINAL ID COLUMNS ( from source):
   1. unit_id

3. ORIGINAL FEATURE COLUMNS (25 preserved, z-score transformed):
   - Operational settings: ['op_setting_1', 'op_setting_2', 'op_setting_3']
   - Cycle Time: time_cycles
   - Sensors: sensor_1 through sensor_21 (21 sensors)
   - Note: All 21 sensors standardized using z-score normalization

4. NEW TRANSFORMED COLUMNS (4 created in this section):
   1. thermal_index
   2. pressure_index
   3. normalized_cycle
   4. operational_complexity

Column order in transformed dataset:
   6 metadata → 1 ID → 25 features → 4 transformed = 36 columns
```

4. SDLM: Analyze Stage

4.1 Data Structure and Data Model (M3-M5)

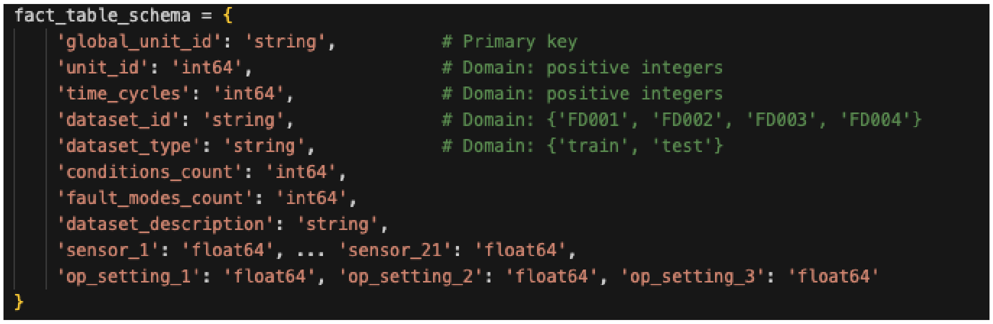
4.1.1 Relational Model (M3)

Following M3 course material on "The Relational Model," this project is grounded in E.F. Codd's 1970 findings. More specifically, the project implements a Kimball Star Schema with embedded dimensions. The Relational Model parameters are below for the schema:

A. **Constraints Implemented:**
    - Primary key: “global_unit_id” uniquely identifies engine units across datasets.
    - Composite key: “(global_unit_id, time_cycles)” uniquely identifies observation
    - time_cycles > 0 (engines start at cycle 1)
    - dataset_id is in {'FD001', 'FD002', 'FD003', 'FD004'}
    - conditions_count is {1, 6}
    - dataset_id in fact table references dataset metadata
    - Foreign key (dataset_id): Each observation belongs to exactly one dataset
    - These constraints are based on data set assumptions, domain knowledge and helps ensure that the data can be validated.

B. Relational Model Characteristics of Fact Table:

- Tuples: Each row represents an (engine unit, time cycle) observation
- Attributes: Each column represents a measured or derived property
- Values: Data elements (int64, float64, string)
- Constraint: Primary key uniqueness, foreign key integrity (dataset_id), domain constraints (positive cycles)
- Fact Table: The integrated C-MAPSS time-series data will form the central Fact Table.
    - **i.** Columns: 32 (26 original sensor/setting measurements + 6 derived metadata)
    - **ii.** Rows: 132,332 observations (tuples)
    - **iii.** Primary Key: “global_unit_id” (composite: dataset_id + unit_id)
    - iv. Schema

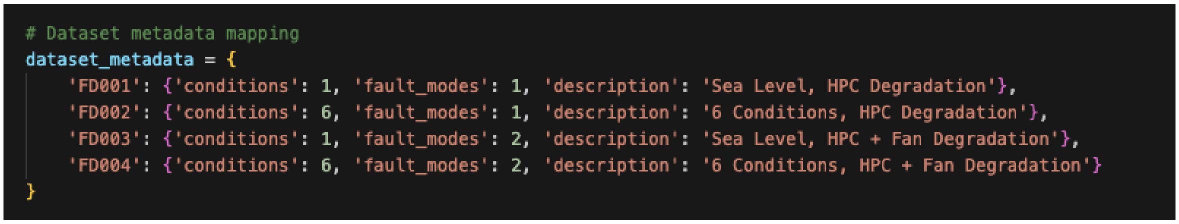


```
fact_table_schema = {
    'global_unit_id': 'string',            # Primary key
    'unit_id': 'int64',                    # Domain: positive integers
    'time_cycles': 'int64',                # Domain: positive integers
    'dataset_id': 'string',                # Domain: {'FD001', 'FD002', 'FD003', 'FD004'}
    'dataset_type': 'string',              # Domain: {'train', 'test'}
    'conditions_count': 'int64',
    'fault_modes_count': 'int64',
    'dataset_description': 'string',
    'sensor_1': 'float64', ... 'sensor_21': 'float64',
    'op_setting_1': 'float64', 'op_setting_2': 'float64', 'op_setting_3': 'float64'
}
```

    - v.

C. Functional Dependencies:

- The Foreign key dataset_id has functional dependencies below:
    - i. dataset_id → conditions_count
        1. If two observations have same `dataset_id`, they have same `conditions_count`
        2. Example: All FD001 observations have `conditions_count = 1`
    - ii. dataset_id → fault_modes_count
        1. FD001: 1 fault mode, FD002: 1 fault mode, FD003: 2 fault modes, FD004: 2 fault modes
    - iii. dataset_id → dataset_description



```
# Dataset metadata mapping
dataset_metadata = {
    'FD001': {'conditions': 1, 'fault_modes': 1, 'description': 'Sea Level, HPC Degradation'},
    'FD002': {'conditions': 6, 'fault_modes': 1, 'description': '6 Conditions, HPC Degradation'},
    'FD003': {'conditions': 1, 'fault_modes': 2, 'description': 'Sea Level, HPC + Fan Degradation'},
    'FD004': {'conditions': 6, 'fault_modes': 2, 'description': '6 Conditions, HPC + Fan Degradation'}
}
```

D. Data Independence:

This project demonstrates **physical** data independence, which means that physical schema (storage method) can change without affecting data interactions. For example, if this project switches from CSV files to Parquet, pandas DataFrame operations remain unchanged. This project also demonstrates **logical** data independence, which means that logical schema can change without breaking applications. For example, when I added new metadata columns (e.g., `dataset_description`) it doesn't break existing queries that reference original 26 sensors

4.1.2 Tree Model (M4)

Following M4 course material, this project models three specific documents using the tree data structure by applying the OHCO (Text is an Ordered Hierarchy of Content Objects) model. The principles of OHCO are:

- **Content Objects**: Logical components (e.g., chapters, sections, paragraphs).
- **Hierarchy**: Objects nest within each other without overlaps (e.g., paragraphs inside sections).
- **Ordered**: Objects follow one another in a specific sequence.

This project's documentation artifacts exemplify the OHCO tree model as follows:

A. **Jupyter Notebook** (hierarchical tree structure):
   - **Content Objects**: The Notebook (root) contains Section and Subsection objects, which contain Paragraph (markdown cell) and Code cell objects.
   - **Hierarchy**: The objects are nested. For example, Subsection 1.1 is a child of Section 1, which is a child of the Notebook (root).
   - **Ordered**: The objects are sequential. Section 1 precedes Section 2, and Subsection 1.1 precedes Subsection 1.2.

B. **RDF Ontology** (cmapss_ontology.ttl) - serialized tree of triples:
   - **Content Objects**: The Ontology (root) contains high-level objects like Classes, Object Properties, and Datatype Properties.
   - **Hierarchy**: These objects contain child objects. Example classes contain Dataset, EngineUnit, and TimeCycle. Object Properties contain hasEngineUnit and hasTimeCycle.
   - **Ordered**: In its serialized .ttl file format, these definitions are listed in a specific,

sequential order.

C. **ER Diagram** (cmapss_erd.html) - conceptual tree as schema

- **Content Objects**: The ER Model (root) contains Entities, Relationships, and Constraints as high-level objects. Entities contain Attributes (e.g., DATASET contains dataset_id, dataset_description, conditions_count, fault_modes_count).

- **Hierarchy**: Objects are nested within categories. There are five entities which are children of the Entities container. Each Entity has Attributes as children. Relationships connect Entities and are children of the Relationships container.

- **Ordered**: Entities are declared first (with their attributes listed sequentially), followed by Relationships listed in order of conceptual dependency (DATASET → ENGINE_UNIT → TIME_CYCLE → SENSOR_MEASUREMENT, then RUL_TARGET).

4.1.3 Data Models: Entity Relationship (ER) Model and Ontology (M5)

A. My perspective on ER and Ontology in the industry

It is very interesting to me that the lecture in Module 5 mentions that “I don’t see any useful distinction between conceptual models and ontologies and so will be using ‘ontologies’ to include models like ER.” I have actually struggled with how these concepts differ, so I can try to articulate how I see them used in the aviation industry.

In my experience, Entity Relationship (ER) models are most often used in Model-Based Systems Engineering (MBSE) to support database architecture. I have primarily built ERs in Cameo, where they serve as the foundation for structuring how data is stored and related. When I use the Department of Defense Architecture Framework (DODAF) for MBSE, I typically start by creating the ER diagram within the Data and Information Viewpoint (DIV).[8] This viewpoint captures the logical structure of data (the entities, attributes, and relationships) and ensures that teams across disciplines work from a **shared definition of the data**.

What I find most powerful about DODAF is how these data models can extend beyond their original viewpoint. Once an ER diagram is created in the DIV, those same entities and relationships can be reused in other viewpoints, such as the Operational Viewpoint (OV). For instance, an entity like Engine Unit in the data model might also appear in an operational context that describes how maintainers interact with the aircraft or how sensor data flows through a maintenance support system. This linkage allows the ER model to serve as a bridge between technical data and operational processes, ensuring that information defined in the data layer remains meaningful to the teams working in mission or maintenance environments. The MBSE approach also simplifies updates across multiple teams, since common data elements are shared and changes in one viewpoint automatically propagate to others, maintaining consistency throughout the architecture.

From an ontology perspective Palantir Foundry builds directly on the principles of the ER model.[9] Foundry provides a platform for constructing end-to-end data transformation pipelines (ETL) that remain tied to the underlying data model. In many organizations, these transformations are performed manually in Excel or through undocumented scripts, making them difficult to validate or reproduce. Foundry addresses this challenge by embedding those workflows into a transparent, version-controlled environment where each transformation step is connected to the entities and relationships defined in the data model.

What makes Foundry particularly interesting in the context of this course is how it uses the term ontology. In Foundry, the ontology layer defines how curated and modeled data maps to real-world operational concepts such as Engine Unit, Sensor Reading, or Maintenance Event. This is closely aligned with the academic definition of ontology introduced by Tom Gruber (Stanford), who defined it as "an explicit specification of a conceptualization of a domain." Gruber explains that a conceptualization is an abstract, simplified view of the world that identifies the objects, concepts, and relationships relevant for a particular purpose.

Viewed through that lens, Foundry's ontology can be seen as the practical, operational form of Gruber's idea because it specifies how an organization conceptualizes its world and makes that conceptualization executable through data. In Foundry, entities from the ER model become the foundational concepts of the ontology, and the ETL pipelines provide the formalized relationships that connect them. The result is a shared, traceable understanding of data that is both semantically meaningful and technically reproducible across teams.

B. Entity Relationship Model - Database Design

I found this section to be very interesting and applicable to my responsibilities at work. For example, I have spent a lot of time utilizing the entity relationship model to conceptualize the aviation domain in terms of physical things and then to map that conceptualization to a schema. I have discovered that being able to articulate the data model into real world things that stakeholders understand is very important to architecting the **proper database design**. Conceptual schemas provide wonderful documentation that can be tailored to stakeholders viewpoint and needs. In the appendix I have included an ERD diagram for the fact table dataset, but a table is below.

C. Ontology

Following the Module 5 course material, an ontology is “an explicit specification of a conceptualization of a domain” (Gruber, 1993). I need an ontology to answer the question of: What entities exist in the turbofan engine monitoring dataset, and how do they relate to each other?

This project created a formal RDF/RDFS ontology file (cmapss_ontology.ttl) that

Entity	Relationship	Cardinality	Example	Key
DATASET → ENGINE_UNIT	One-to-Many	Each dataset contains multiple engine units	FD001 contains 100 training engines and 100 test engines	ENGINE_UNIT.dataset_id references DATASET.dataset_id
ENGINE_UNIT → TIME_CYCLE	One-to-Many	Each engine operates over multiple time cycles	Engine FD001_1 operates from cycle 1 to cycle 192	TIME_CYCLE.global_unit_id references ENGINE_UNIT.global_unit_id
ENGINE_UNIT → SENSOR	One-to-Many	Each engine generates multiple sensor observations	Engine FD001_1 has 192 sensor measurement records (one per cycle)	SENSOR.global_unit_id references ENGINE_UNIT.global_unit_id
TIME_CYCLE ↔ SENSOR	One-to-One	Each time cycle has exactly one sensor measurement record	Global_unit_id + time_cycles + sensor	(global_unit_id, time_cycles) uniquely identifies observation
ENGINE_UNIT → RUL_TARGET	One-to-One	Each engine has a RUL target value	The fact table only has RUL for test, I will create the training RUL	RUL_TARGET.global_unit_id references ENGINE_UNIT.global_unit_id

defines nine core classes and their relationships:

Core Classes

1. Dataset – Experimental configuration (FD001–FD004)
2. EngineUnit – Individual turbofan undergoing degradation
3. TimeCycle – Discrete operational cycle in the engine's lifecycle
4. FactRecord – A single observation capturing state at one cycle
5. SensorMeasurement – Collection of all 21 sensor readings
6. SensorReading – Individual sensor value (e.g., temperature, pressure, speed)
7. OperationalSetting – Collection of three operating parameters
8. OperatingSetting – Individual setting value
9. RULTarget – Remaining Useful Life ground truth (target variable)

Relationships

1. hasEngineUnit: Dataset (1) → EngineUnit (many)
2. hasTimeCycle: EngineUnit (1) → TimeCycle (many)
3. hasFactRecord: EngineUnit (1) → FactRecord (many)
4. capturedAt: FactRecord (1) → TimeCycle (1)
5. hasSensorMeasurement: FactRecord (1) → SensorMeasurement (1)
6. hasRULTarget: EngineUnit (1) → RULTarget (0 or 1)

An ontology can be seen in the appendix.

D. Academic Definition: Ontology versus ERD

I agree with the M5 course material that there's no fundamental distinction between the RDF ontology and ER diagram. I believe the only difference is really just with their implementation. Both model identical entities (Dataset, EngineUnit, TimeCycle, Sensors, RUL), relationships (contains, operates over, generates, captured at), and cardinalities (1:M, 1:1, 1:0..1).

The differences are in implementation format and modeling approach:

1. Implementation format: The RDF ontology uses Turtle syntax (.ttl) following RDFS/OWL standards, which was the format referenced in the lecture for ontologies. The ER diagram uses Mermaid syntax (.md) following Chen's 1976 ER model, enabling visual diagramming and SQL schema generation.

2. Modeling Approach: The RDF ontology models each sensor reading (sensor_1, sensor_2, etc.) as potentially separate SensorReading instances with a general class hierarchy (sensor readings are instances of measurement concepts). The ER diagram treats them as 21 separate attributes of the SENSOR_MEASUREMENT entity, reflecting a database orientation (attributes = columns in a table).

Both serve the same M5 purpose: providing a conceptual schema that abstracts away from physical implementation (CSV files, pandas DataFrames) and documents the domain independent of storage technology. As the M5 instructor states: "I don't see any useful distinction between conceptual models and ontologies" - they're complementary representations optimized for different tool ecosystems (semantic web vs. relational databases).

5. SDLM: Preserve Stage

5.1.1 DVC (M9, M10)

I implemented Data Version Control (DVC) to provide identity verification for each stage of the data curation pipeline. After creating four pipeline stages (fact_table, fact_table_clean, fact_table_sensor_deduplicated, fact_table_transformed), I exported each as CSV files and tracked them with DVC, which automatically computed MD5 hashes for each version. The DVC tracking revealed that Stage 2 (after outlier removal) and Stage 3 (after deduplication) had identical hashes (39bc59e1a964b477f67c0775901c4bc1), confirming that zero duplicate sensor readings were found, which is what I expected. Stage 1 and Stage 4 had different hashes, confirming that outlier removal successfully modified the dataset and that feature engineering added 4 derived columns (thermal_index, pressure_index, normalized_cycle, operational_complexity). This DVC implementation provides content-addressed storage using hashes,which helps with dataset verification. The DVC hashes are also uploaded to github for verification and tracking: https://github.com/djhavera/fdc/tree/main

5.1.2 W&B (M9, M10, M12, & M13)

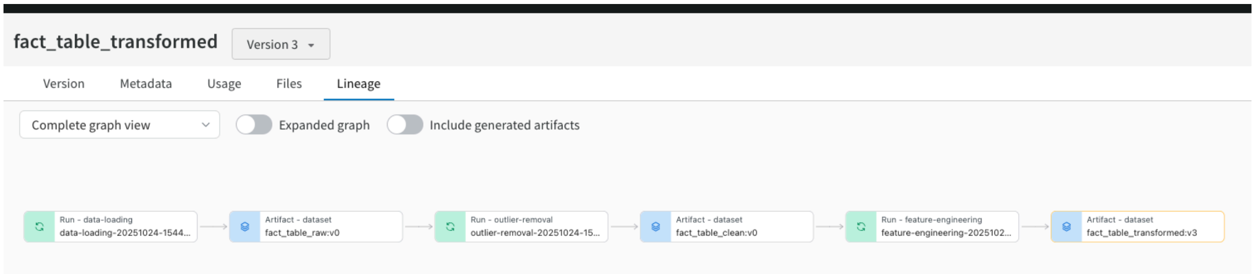
To complement the DVC tracking of the dataset, I used Weights and Biases (W&B) to track the transformations on the datasets, so that I could understand why the datasets were changing. I chose to use W&B because data transformation represents the start of the machine learning (ML) process and W&B allows me to log data sets as artifacts and easily run them through the upstream ML process. The directed acyclic graph (DAG) of runs and artifacts is very beneficial as well during the ML cycle. I actually believe this can be a nice alternative to the ETL capability that Palantir foundry offers. More specifically, you can find the W&B artifacts at the links below:

- W&B Run:
https://wandb.ai/djhavera/cmapss-data-curation/runs/44bae6u4/overview?nw=nwuserdjhavera

- Dataset Artifact:
https://wandb.ai/djhavera/cmapss-data-curation/artifacts/dataset/fact_table_transformed/v4/metadata

My W&B implementation establishes a complete 3-step data provenance chain: fact_table_raw:v0, fact_table_clean:v1, fact_table_transformed:v4. The lineage graph (https://wandb.ai/djhavera/cmapss-data-curation/artifacts/dataset/fact_table_transformed/v4/lineage) shows that the final dataset was created through: (1) outlier removal eliminating 507 records, and (2) feature engineering applying z-score normalization and deriving 4 semantic features (thermal_index, pressure_index, normalized_cycle, operational_complexity).

This automated documentation directly addresses Merali's call for better scientific record-keeping (M13) by providing full traceability without manual effort. Each artifact version has a unique hash enabling dataset identity verification (M9), and all transformation parameters are logged for exact reproducibility (M12). In a production ML system, this lineage would extend to model training artifacts and deployment versions, enabling complete audit trails from raw sensor data to RUL predictions.



6. SDLM: Publish/Share Stage

I implemented modern scientific communication practices that extend beyond traditional PDF publications by deploying two complementary MLOps systems for transparent, reproducible data curation documentation.

First, I used DVC (Data Version Control) to track four pipeline stages with cryptographic MD5 hashes, providing mathematical proof of transformation correctness: Stage 1 to 2 hash changed (confirming 507 outliers removed), Stage 2 to 3 hash remained identical (confirming zero duplicates found), and Stage 3 to 4 hash changed (confirming 4 features added).

Second, I implemented Weights & Biases (W&B) to create a complete 3-step artifact lineage chain (fact_table_raw:v0 to fact_table_clean:v1 to fact_table_transformed:v4) with logged metrics documenting every transformation parameter, enabling exact reproducibility.

Third, I used a public git repository where you can access the jupyter notebook: https://github.com/djhavera/fdc/tree/main

This implementation directly addresses M15 lecture themes by:

1. Supporting strategic reading through structured W&B dashboards (Config/Summary/Lineage tabs)
2. Enhancing research integrity via cryptographic verification and public access
3. Creating a "research object" that bundles executable code, versioned data, and

auto-generated provenance, which embodies Buckheit & Donoho's principle that "the actual scholarship is the complete software development environment."

The public W&B lineage graph (https://wandb.ai/djhavera/cmapss-data-curation/artifacts/dataset/fact_table_transformed/v3/lineage) provides visual proof of complete data provenance from raw sensor readings through quality improvements to final engineered features, addressing Merali's critique about poor computational documentation by automatically tracking the entire transformation pipeline without manual record-keeping.

7. Cross-Cutting Activities (Describing, Managing Quality, Backup & Secure)

7.1 Describing

7.1.1 Data Models: Entity Relationship (ER) Model and Ontology (M7)

This section uses the Basic Representation Model (BRM) to understand what "data" is and how the C-MAPSS dataset is represented across multiple levels. The BRM is a nice continuation of our ERD because it is a summary of the entities and relationships involved in the representation of the data set's digital objects.[10] The BRM distinguishes data by propositional content (what the data means), symbol structure (how it's organized), and physical instantiation (where it's stored).

A. Propositional content (what the data means)
   a. Represents the language-independent meaning expressed by symbol structures.
   b. Propositions are things that can be true or false.
   c. An example would be the ENGINE_UNIT → RUL_TARGET relationship:
      i. English: "Test engine FD001_100 has a remaining useful life target of 112 cycles".
      ii. Relational tuple in Dataframe: `(global_unit_id='FD001_100', rul_target=112)
      iii. ER notation: `FD001_100 → [RUL_TARGET: 112]`
   d. The representation in the example has the same propositional content, but all reference the one-to-one relationship of engine unit to rul target.

B. Symbol structure (how it's organized)[10]
   a. This refers to the abstract arrangements of symbols used to express the propositional content. This project's C-MAPSS dataset illustrates a "cascade" of representational layers such:
      i. Relational structure: Rows and columns with 32 attributes per observation.
      ii. RDF triple structure in ontology .ttl file: Subject-predicate-object format

in ontology .ttl file.

iii. The same propositional content (e.g., "sensor_2 = 643.53 for FD001_1 at cycle 192") can be expressed by different symbol structures (graph, table)

C. Physical instantiation (where it's stored)
   a. This refers to the physical hardware where the data is saved to.
   b. Electric charge states in SSD flash memory cells
   c. Magnetic orientations on hard disks
   d. Git repository blob objects (SHA-1 hash-addressed)

7.1.2 Standards (M11)

Most of the standards that I used were informal by storing data in a dataframe and manipulating in pandas, therefore there was no strict approval process to follow. I did depart from standards when I created custom metadata columns, but since these can easily be deleted they don't impact conformance to a standard.

7.2 Managing Quality / Backup & Secure

This is already captured in section 5 and section 6 above.

8. Conclusion

This project began with a challenge familiar from my 15 years at GE Aerospace: data pipelines that fail to deliver transparency, reproducibility, or long-term usefulness, which are issues that often drive organizations toward expensive enterprise platforms like Palantir Foundry. Through a complete data curation workflow for NASA's C-MAPSS turbofan dataset, I demonstrated that open-source MLOps tools can achieve the same capabilities with greater flexibility and at a fraction of the cost.

The workflow proved that three core pillars of transparency, automation, and reproducibility can be fully realized using open source tools like Jupyter, pandas, DVC, W&B, and GitHub.

- Transparency: W&B provided interactive data lineage graphs that mirror Foundry's visual pipeline tracking, but without licensing costs.
- Automation: W&B logged 114 transformation metrics and DVC tracked versioned datasets with cryptographic hashes, eliminating manual documentation drift.
- Reproducibility: DVC's hash-based verification and W&B's parameter logging ensured every dataset version and transformation step can be exactly reproduced.

While Foundry offers enterprise features like SSO and managed infrastructure, this open-source stack provides functional equivalence for core data curation needs. For aviation predictive maintenance systems, where data transparency and auditability are critical, these open tools deliver the same rigor for much less cost

The project reinforced that effective data curation depends less on software and more on disciplined application of foundational principles: version control, experiment tracking, and cryptographic verification.

Ultimately, this work provides a blueprint for aerospace data science teams seeking enterprise-grade transparency with open, reproducible, and verifiable workflows—proving that high-quality data stewardship is achievable without enterprise budgets.

9. Next Steps

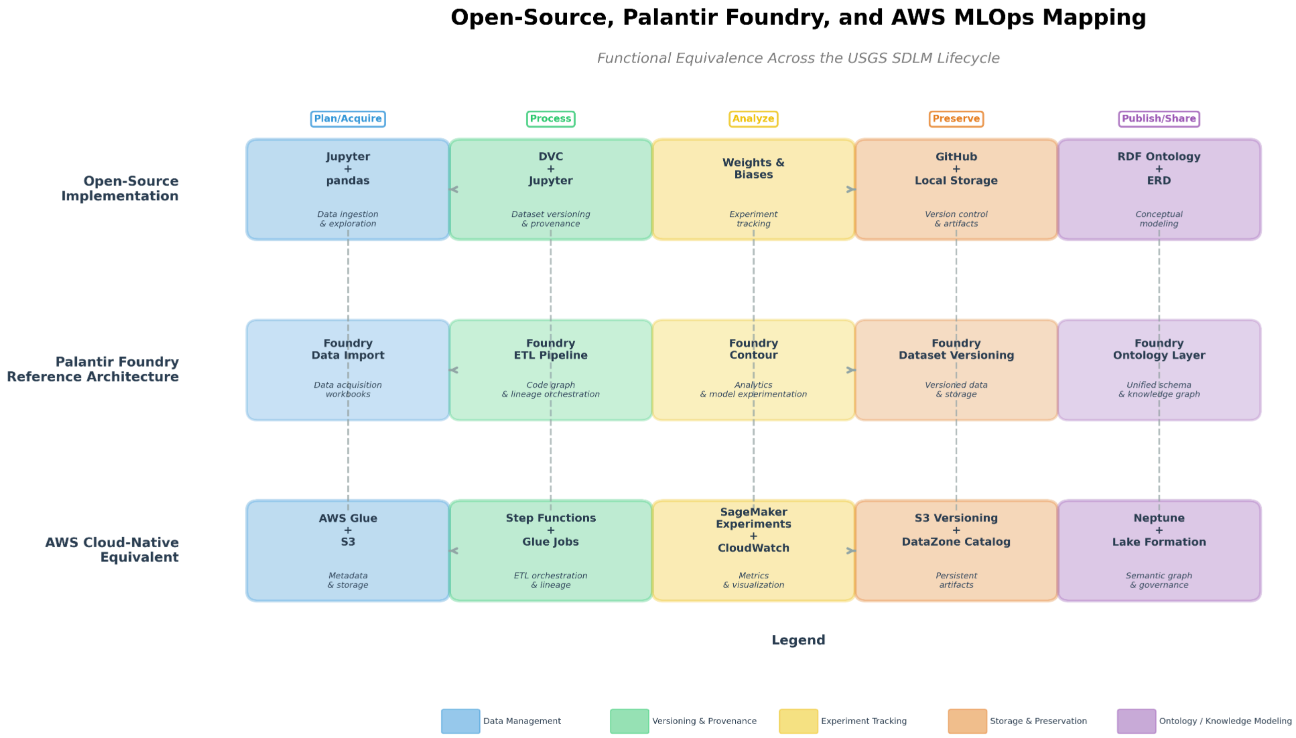
Rather than viewing these ecosystems as competing or mutually exclusive, it is more accurate to see them as interchangeable options along a spectrum of control, automation, and cost. The open-source stack (DVC, Weights & Biases, pandas, GitHub) offers full transparency and flexibility, which is ideal for research, prototyping, and organizations that value traceability and reproducibility without licensing costs. AWS services (Glue, SageMaker, Step Functions, S3, Neptune) provide a middle ground, automating many of the same functions while maintaining open standards and scalability for production systems. Palantir Foundry represents the enterprise-managed end of the spectrum, which emphasizes integrated governance, access control, and semantic consistency across teams, but at higher cost and less configurability.

Ultimately, these approaches align with different organizational needs: open-source for experimentation and learning, AWS for scalable and governed implementation, and Foundry for end-to-end enterprise data management. Teams can select the combination, or single ecosystem, that best fits their maturity, budget, and transparency requirements.

Open-Source vs AWS vs Palantir Foundry

Concept / Function	**Open-Source Implementation**	**AWS Equivalent**	**Palantir Foundry Equivalent**	**Purpose / Integration Notes**
Data Versioning & Provenance	DVC (Data Version Control)	S3 Versioning + AWS Glue Data Catalog +	Foundry Data Lineage / Code Workbook	Foundry and AWS both formalize lineage at the schema and transformation level, similar to DVC's

		SageMaker Lineage Tracking	Pipelines	commit-based workflow.
Experiment Tracking & Visualization	Weights & Biases (W&B)	SageMaker Experiments + CloudWatch Metrics + Model Dashboard	Foundry Machine Learning (ML) Modules + Ontology Workflows	AWS offers granular lineage and metric dashboards, while Foundry integrates ML results directly into its ontology for enterprise visibility.
Ontology / Conceptual Model	RDF Ontology (cmapss_ontology.ttl)	Amazon Neptune + AWS Glue Data Catalog	Foundry Ontology Layer	AWS and Foundry both treat ontology as the semantic layer connecting datasets, pipelines, and operational views.
Entity Relationship Model (ERD)	ERD (Cameo / Star Schema)	AWS Glue Schema Registry + AWS Lake Formation	Foundry Object Model (Entity Schema Designer)	Foundry's schema designer and AWS Glue's schema registry both provide governed, reusable entity structures.
Pipeline Orchestration	Python + Jupyter + DVC Pipelines	AWS Step Functions + SageMaker Pipelines	Foundry Code Workbooks + Transformation Graphs	Manages ETL, feature engineering, and model training flows. Step Functions mirror Foundry's visual pipeline orchestration, integrating with AWS Glue and SageMaker.
Artifact Storage / Versioned Data Lake	Local Storage + GitHub	Amazon S3 (Versioned Buckets)	Foundry Data Lineage-Enabled Object Store	Central repository for artifacts and intermediate datasets. S3 and Foundry both maintain historical versions for full auditability.



Open-Source, Palantir Foundry, and AWS MLOps Mapping

Functional Equivalence Across the USGS SDLM Lifecycle

	Plan/Acquire	Process	Analyze	Preserve	Publish/Share
Open-Source Implementation	**Jupyter + pandas** *Data ingestion & exploration*	**DVC + Jupyter** *Dataset versioning & provenance*	**Weights & Biases** *Experiment tracking*	**GitHub + Local Storage** *Version control & artifacts*	**RDF Ontology + ERD** *Conceptual modeling*
Palantir Foundry Reference Architecture	**Foundry Data Import** *Data acquisition workbooks*	**Foundry ETL Pipeline** *Code graph & lineage orchestration*	**Foundry Contour** *Analytics & model experimentation*	**Foundry Dataset Versioning** *Versioned data & storage*	**Foundry Ontology Layer** *Unified schema & knowledge graph*
AWS Cloud-Native Equivalent	**AWS Glue + S3** *Metadata & storage*	**Step Functions + Glue Jobs** *ETL orchestration & lineage*	**SageMaker Experiments + CloudWatch** *Metrics & visualization*	**S3 Versioning + DataZone Catalog** *Persistent artifacts*	**Neptune + Lake Formation** *Semantic graph & governance*

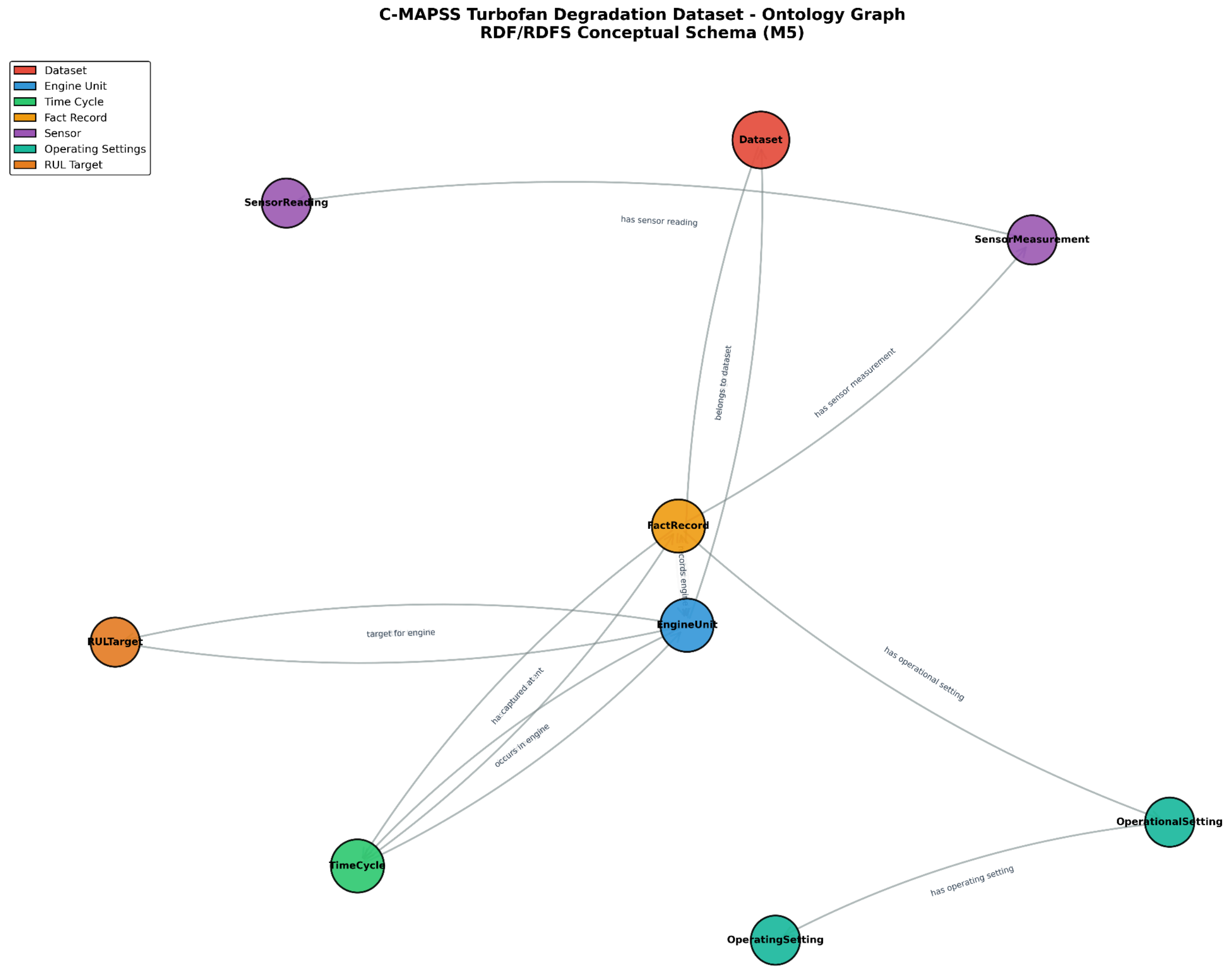
Legend

🟦 Data Management 🟩 Versioning & Provenance 🟨 Experiment Tracking 🟧 Storage & Preservation 🟪 Ontology / Knowledge Modeling

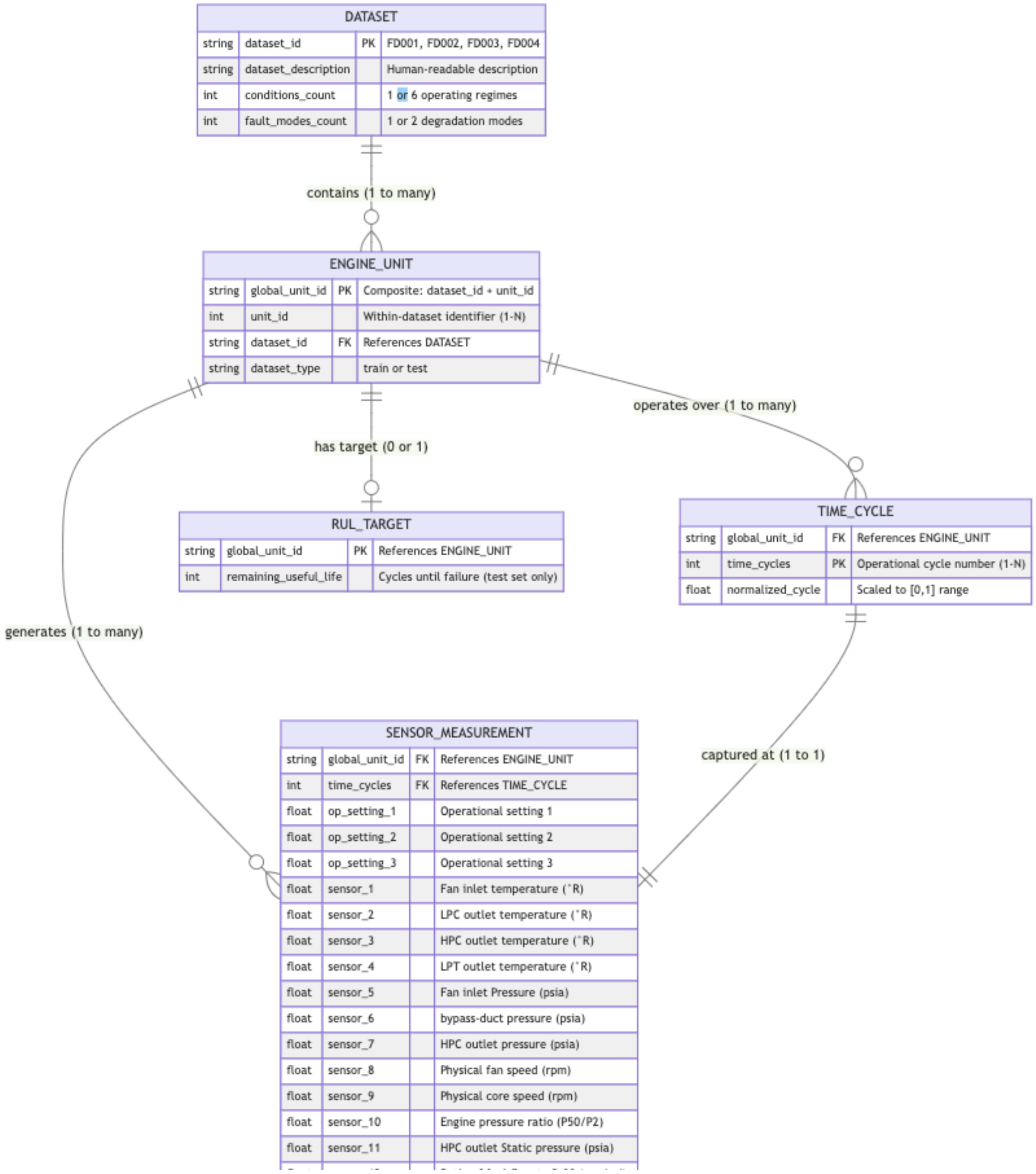
Works cited

1. Plale, B., & Kouper, I. (2017). The Centrality of Data: Data Lifecycle and Data Pipelines. In C. R. Sugimoto, H. Ekbia, & M. Mattioli (Eds.), *Big Data is Not a Monolith: Policies, Practices and Problems*. MIT Press.
2. Star Schema, accessed September 29, 2025, https://en.wikipedia.org/wiki/Star_schema
3. CMAPSS Jet Engine Simulated Data - Dataset - Catalog, accessed August 29, 2025, https://catalog.data.gov/dataset/cmapss-jet-engine-simulated-data
4. Download the aircraft registration database - Federal Aviation Administration, accessed August 29, 2025, https://www.faa.gov/licenses_certificates/aircraft_certification/aircraft_registry/releasable_aircraft_download
5. ASRS Database Online - Aviation Safety Reporting System, accessed August 29, 2025, https://asrs.arc.nasa.gov/search/database.html
6. Ihab, A., & Chu, X. (2019). Data cleaning. Association for Computing Machinery.
7. Saxena, A., Goebel, K., Simon, D., & Eklund, N. (2008). "Damage Propagation Modeling for Aircraft Engine Run-to-Failure Simulation." *Proceedings of the 1st International Conference on Prognostics and Health Management (PHM08)*, Denver, CO.

8. DoDAF Viewpoints and Models, Accessed October 22, https://dodcio.defense.gov/Library/DoD-Architecture-Framework/dodaf20_standards/
9. The Foundry Ontology, Accessed October 22, https://www.palantir.com/platforms/foundry/foundry-ontology/
10. Wickett, K. M., Sacchi, S., Dubin, D., & Renear, A. H. (2012). Identifying Content and Levels of Representation in Scientific Data, 1-10. https://doi.org/10.1002/meet.14504901199



Entity-Relationship Diagram (M5 Conceptual Model)



DATASET

string	dataset_id	PK	FD001, FD002, FD003, FD004
string	dataset_description		Human-readable description
int	conditions_count		1 or 6 operating regimes
int	fault_modes_count		1 or 2 degradation modes

ENGINE_UNIT

string	global_unit_id	PK	Composite: dataset_id + unit_id
int	unit_id		Within-dataset identifier (1-N)
string	dataset_id	FK	References DATASET
string	dataset_type		train or test

RUL_TARGET

string	global_unit_id	PK	References ENGINE_UNIT
int	remaining_useful_life		Cycles until failure (test set only)

TIME_CYCLE

string	global_unit_id	FK	References ENGINE_UNIT
int	time_cycles	PK	Operational cycle number (1-N)
float	normalized_cycle		Scaled to [0,1] range

SENSOR_MEASUREMENT

string	global_unit_id	FK	References ENGINE_UNIT
int	time_cycles	FK	References TIME_CYCLE
float	op_setting_1		Operational setting 1
float	op_setting_2		Operational setting 2
float	op_setting_3		Operational setting 3
float	sensor_1		Fan inlet temperature (°R)
float	sensor_2		LPC outlet temperature (°R)
float	sensor_3		HPC outlet temperature (°R)
float	sensor_4		LPT outlet temperature (°R)
float	sensor_5		Fan inlet Pressure (psia)
float	sensor_6		bypass-duct pressure (psia)
float	sensor_7		HPC outlet pressure (psia)
float	sensor_8		Physical fan speed (rpm)
float	sensor_9		Physical core speed (rpm)
float	sensor_10		Engine pressure ratio (P50/P2)
float	sensor_11		HPC outlet Static pressure (psia)