Table of Contents

# Code Documentation Guide

## 📚 Overview

This document explains how the Video Analytics codebase is documented and how to maintain clear, helpful documentation as you refactor and extend the code.

## 🎯 Documentation Philosophy

**Goal**: Every engineer or tester should be able to understand **what the code does**, **why it does it**, and **how to use it** without needing to ask someone else.

**Key Principles**: 1. **Clarity over brevity**: It’s better to be verbose and clear than terse and confusing 2. **Examples everywhere**: Show concrete examples, not just abstract descriptions 3. **Explain the “why”**: Don’t just say what the code does, explain why it’s designed that way 4. **Business context**: Connect technical implementation to business value 5. **Expected outcomes**: Tell readers what results to expect

## 📝 Documentation Levels

### Level 1: File-Level Documentation (Top of File)

Every file should start with a comprehensive header explaining:

"""  
File Title - Brief Description  
================================  
  
PURPOSE:  
--------  
What does this file/module do? What problem does it solve?  
  
BUSINESS VALUE:  
--------------  
Why does this exist? What business questions does it answer?  
  
INPUT DATA STRUCTURE:  
--------------------  
What data does this expect? Include:  
- Column names and types  
- Example rows  
- Required vs optional fields  
  
OUTPUT DATA STRUCTURE:  
---------------------  
What does this produce? Include:  
- Column names and types  
- Example output  
- One row per what?  
  
EXAMPLE OUTPUT:  
--------------  
Show a concrete example with real numbers and calculations explained  
  
KEY CONCEPTS:  
------------  
Define any special terms or algorithms used  
  
TYPICAL USAGE:  
-------------  
Show how to run this code with a simple example  
  
PERFORMANCE CONSIDERATIONS:  
--------------------------  
Any scalability concerns or optimization tips  
  
DATA QUALITY HANDLING:  
---------------------  
How does this handle bad data?  
"""

**Example from our codebase**: See [databricks\_video\_aggregation.py:1-128](databricks_video_aggregation.py#L1-L128)

### Level 2: Class-Level Documentation

Document the purpose and design of each class:

class VideoEngagementAggregator:  
 """  
 One-line summary of what this class does.  
  
 Detailed explanation of the class's role in the system,  
 its design philosophy, and how it fits into the bigger picture.  
  
 DESIGN PHILOSOPHY:  
 -----------------  
 Explain the approach taken and why  
  
 EXAMPLE:  
 -------  
 >>> # Show typical usage  
 >>> aggregator = VideoEngagementAggregator(...)  
 >>> results = aggregator.run\_aggregation()  
 """

**Example from our codebase**: See [databricks\_video\_aggregation.py:146-177](databricks_video_aggregation.py#L146-L177)

### Level 3: Method-Level Documentation

Every method should have detailed documentation:

def calculate\_watch\_segments(self, events\_df):  
 """  
 Brief one-line description.  
  
 Detailed explanation of what this method does and why it's important.  
  
 ALGORITHM EXPLANATION:  
 ---------------------  
 Step-by-step breakdown of the algorithm:  
 1. First we do X  
 2. Then we do Y because Z  
 3. Finally we return W  
  
 EXAMPLE:  
 -------  
 Show input data:  
 | timestamp | userId | eventName | currentTime |  
 |-----------|--------|-----------|-------------|  
 | ... | ... | ... | ... |  
  
 Show output data:  
 | timestamp | watchedSeconds | isValidSegment |  
 |-----------|----------------|----------------|  
 | ... | ... | ... |  
  
 Explain calculations:  
 - Row 1: X happened because Y, resulting in Z  
  
 Args:  
 param1 (type): What this parameter is, with examples  
 param2 (type): What this parameter is, with examples  
  
 Returns:  
 type: What is returned, with description of structure  
  
 DEBUGGING TIPS:  
 --------------  
 How to inspect results for troubleshooting  
 """

**Example from our codebase**: See [databricks\_video\_aggregation.py:296-435](databricks_video_aggregation.py#L296-L435)

### Level 4: Inline Comments

Use inline comments to explain **why**, not **what**:

# ❌ BAD: Explains what (obvious from code)  
df = df.filter(col("currentTime") >= 0) # Filter for non-negative current time  
  
# ✅ GOOD: Explains why (provides business context)  
df = df.filter(col("currentTime") >= 0) # Position can't be negative - this filters out data errors

**Guidelines for inline comments**: - Comment complex logic that isn’t immediately obvious - Explain business rules and constraints - Document assumptions and trade-offs - Highlight potential issues or edge cases - Reference related sections of code

**Example**:

# Valid segment must start with play/resume and end with pause/ended  
# This ensures we only count actual watch time, not paused periods  
df = df.withColumn(  
 "isValidSegment",  
 (col("prevEvent").isin(["video\_play", "video\_resume"])) & # Segment start  
 (col("eventName").isin(["video\_pause", "video\_ended"])) & # Segment end  
 (col("timeDelta") > 0) & # Forward progress  
 (col("timeDelta") < 7200) & # Max 2 hours (data quality)  
 (col("timeDelta").between(0, col("timestampDelta") + 5)) # Plausibility check  
)

## 🎓 Jupyter Notebook Documentation

Notebooks should be heavily documented because they’re used for learning and exploration.

### Structure

# Notebook Title  
  
## 📚 Purpose  
What this notebook does and why it exists  
  
## 🎯 Learning Objectives  
What you'll understand after running this  
  
## 🚀 Quick Start  
Minimal steps to get started  
  
## 📊 What You'll Build  
Visual diagram or summary  
  
## ⏱️ Estimated Time  
How long this takes  
  
---  
  
## Section 1: Section Name  
  
### What We're Doing  
High-level explanation  
  
### Why This Matters  
Business context  
  
### Understanding the Data  
Detailed explanation of data structures  
  
### Expected Results  
What output to expect

### Code Cells

Add markdown cells **before** each code cell explaining: - What the code does - Why we’re doing it this way - What to look for in the output

**Example**:

# MAGIC %md  
# MAGIC ### TC-001: Perfect Viewing  
# MAGIC  
# MAGIC This test case simulates a user who watches a video from start to finish  
# MAGIC without any pauses or interruptions. This is the "ideal" case.  
# MAGIC  
# MAGIC \*\*Expected Behavior\*\*:  
# MAGIC - watchPercentage: 100%  
# MAGIC - completionPercentage: 100%  
# MAGIC - sessionCount: 1  
# MAGIC - pauseCount: 0  
  
# COMMAND ----------  
  
tc001 = [  
 (base\_time, "anna", "session\_tc001", "video\_001", "video\_play", 0.0),  
 (base\_time + timedelta(seconds=300), "anna", "session\_tc001", "video\_001", "video\_ended", 300.0),  
]

## 📊 Test Documentation

### Test Scenario Documentation

Each test scenario should be documented with:

# ============================================================================  
# TC-XXX: Scenario Name - Brief Description  
# ============================================================================  
# Video: video\_XXX (duration details)  
# Behavior: What the user does in plain English  
#  
# Timeline:  
# HH:MM:SS - Event description (calculation)  
# HH:MM:SS - Event description (calculation)  
#  
# Expected Metrics:  
# - metric1: X (explanation of calculation)  
# - metric2: Y (explanation of calculation)  
#  
# Why This Test Matters:  
# Business or technical reason this scenario is important

**Example**:

# ============================================================================  
# TC-005: Skip Backward (Rewind)  
# ============================================================================  
# Video: video\_001 (5 minutes = 300 seconds)  
# Behavior: Watches beginning, pauses, continues, then rewinds 10 seconds  
#  
# Timeline:  
# 10:00:00 - Plays from 0s  
# 10:00:30 - Pauses at 30s (watched 30s)  
# 10:00:35 - Resumes at 30s  
# 10:02:05 - Pauses at 120s (watched 90s more)  
# 10:02:10 - Resumes at 110s (REWIND 10 seconds)  
# 10:02:20 - Pauses at 120s (watched 10s more)  
#  
# Expected Metrics:  
# - totalWatchTime: 130 seconds (30 + 90 + 10)  
# - uniqueSecondsWatched: 120 seconds (0-120, but 110-120 watched twice)  
# - backwardSkipCount: 1  
#  
# Why This Test Matters:  
# Tests that we correctly handle overlapping time segments when users  
# rewind to re-watch content. Unique seconds should not double-count.

## 🔍 Documentation Checklist

Before committing code, verify:

### File Level

* File has comprehensive header documentation
* Purpose and business value are clear
* Input/output data structures are documented
* Example with calculations is provided
* Key concepts are defined

### Class Level

* Class purpose is documented
* Design philosophy is explained
* Typical usage example is shown

### Method Level

* Method has docstring with description
* Algorithm steps are explained
* Parameters are documented with types and examples
* Return value is documented
* Example with input/output is provided
* Edge cases and gotchas are mentioned

### Inline Level

* Complex logic has explanatory comments
* Business rules are documented
* Magic numbers have comments explaining their meaning
* Assumptions are stated

### Test Level

* Each test scenario has a description
* Expected behavior is documented
* Timeline/sequence is explained
* Expected metrics are calculated and explained

## 💡 Documentation Examples

### ❌ Bad Documentation

def calculate\_metrics(df):  
 """Calculate metrics"""  
 df = df.groupBy("userId").agg(sum("watchTime"))  
 return df

**Problems**: - No explanation of what metrics - No parameter types - No return description - No examples - No context

### ✅ Good Documentation

def calculate\_metrics(self, events\_df):  
 """  
 Calculate user-level engagement metrics from video events.  
  
 This method aggregates individual video events into summary statistics  
 for each user, including total watch time, number of videos watched,  
 and average engagement score.  
  
 BUSINESS VALUE:  
 --------------  
 These metrics power our user segmentation and personalization features.  
 Marketing uses them to identify highly engaged users for campaigns.  
  
 ALGORITHM:  
 ---------  
 1. Group events by userId  
 2. Sum all watchTime values  
 3. Count distinct videoIds  
 4. Calculate average engagement score  
  
 Args:  
 events\_df (DataFrame): Video events with columns:  
 - userId (str): User identifier  
 - videoId (str): Video identifier  
 - watchTime (float): Seconds watched  
 - engagementScore (float): Event-level engagement  
  
 Returns:  
 DataFrame: User-level metrics with columns:  
 - userId (str): User identifier  
 - totalWatchTime (float): Total seconds watched across all videos  
 - videoCount (int): Number of distinct videos watched  
 - avgEngagement (float): Average engagement score  
  
 Example:  
 >>> # Input  
 >>> events\_df.show()  
 | userId | videoId | watchTime | engagementScore |  
 |--------|---------|-----------|-----------------|  
 | user1 | vid1 | 100 | 75 |  
 | user1 | vid2 | 50 | 60 |  
 | user2 | vid1 | 200 | 90 |  
  
 >>> # Output  
 >>> metrics = aggregator.calculate\_metrics(events\_df)  
 >>> metrics.show()  
 | userId | totalWatchTime | videoCount | avgEngagement |  
 |--------|----------------|------------|---------------|  
 | user1 | 150 | 2 | 67.5 |  
 | user2 | 200 | 1 | 90.0 |  
  
 Performance:  
 This method triggers a shuffle operation due to groupBy.  
 For large datasets (>10M events), consider pre-filtering  
 by date range to reduce data volume.  
 """  
 # Group by user and aggregate metrics  
 # Using distinct count for videoId to handle duplicate events  
 user\_metrics = events\_df.groupBy("userId").agg(  
 sum("watchTime").alias("totalWatchTime"),  
 countDistinct("videoId").alias("videoCount"),  
 avg("engagementScore").alias("avgEngagement")  
 )  
  
 return user\_metrics

## 🚀 Quick Tips for Writing Good Documentation

1. **Write documentation FIRST**: Write the docstring before implementing the function
2. **Use concrete examples**: Don’t just say “aggregate by user” - show sample input/output
3. **Explain calculations**: Show the math: watchPercentage = (130 / 300) \* 100 = 43.3%
4. **Link to business value**: Connect technical details to business impact
5. **Update when refactoring**: If you change code, update documentation immediately
6. **Test your examples**: Make sure code examples actually run
7. **Use formatting**: Bold, tables, and sections make documentation scannable
8. **Think like a new person**: Pretend you’re seeing this code for the first time

## 📚 Additional Resources

* [PEP 257 – Docstring Conventions](https://peps.python.org/pep-0257/)
* [Google Python Style Guide](https://google.github.io/styleguide/pyguide.html#38-comments-and-docstrings)
* [NumPy Documentation Guide](https://numpydoc.readthedocs.io/en/latest/format.html)

## ✅ Benefits of Good Documentation

When documentation is comprehensive:

1. **Onboarding is faster**: New team members can understand code without constant questions
2. **Refactoring is safer**: Clear intent makes it easier to change code without breaking it
3. **Debugging is quicker**: Examples help identify where things went wrong
4. **Testing is easier**: Expected behavior is clearly defined
5. **Collaboration improves**: Team members can work independently
6. **Technical debt reduces**: Well-documented code is more maintainable

## 📞 Questions?

If you’re unsure how to document something: 1. Look at existing examples in this codebase 2. Ask yourself: “If I saw this code in 6 months, would I understand it?” 3. Have a teammate review your documentation 4. Err on the side of over-documenting rather than under-documenting

*Last updated: 2025-10-06* *Documentation standard version: 1.0*