```
from typing import Union, Dict, Optional, Any, List, Tuple
import json
import requests
import uuid
from string import Template

from pyspark.sql import DataFrame, SparkSession
from pyspark.sql import functions as F, types as T # noqa

# --- DQX Imports ---
from databricks.labs.dqx.profiler.profiler import DQProfiler
from databricks.labs.dqx.profiler.generator import DQGenerator
from databricks.labs.dqx.metrics_observer import DQMetricsObserver
from databricks.labs.dqx.config import InputConfig, WorkspaceFileChecksStorageConfig, LLMModelConfig
from databricks.labs.dqx.engine import DQEngine
from databricks.sdk import WorkspaceClient

# --- Local Config ---
from dqx_config import (
init_logging,
get_rules_path,
extrayendo_summary_stats,
create_dataframe,
filter_to_return_final_dfs,
DEFAULT_LLM_MODEL,
DQX_NOTIFY_TEMPLATE_STR
)

logger = init_logging("dqx_quality")

#
==========================================================================
# CLASE DQXAlert (Efectos Secundarios: Guardar y Notificar)
#
==========================================================================
```

```
class DQXAlert:
"""
Gestiona las acciones post-validación:
1. Guardado condicional de rechazados (Flag).
2. Notificación condicional (Template JSON) basada en métricas.
"""
def __init__(self, webhook_url: Optional[str] = None):
self.webhook_url = webhook_url

def save_rejected(self, df_rejected: DataFrame, target_table: str, save_flag: int) -> None:
"""
Inserta el df_rejected a una tabla Delta.
Flag: 1 guarda, 0 no guarda.
"""
if save_flag == 1:
try:
logger.info(f"💾 [DQXAlert] Guardando rechazados en: {target_table}")
(df_rejected.write
.format("delta")
.mode("append")
.option("mergeSchema", "true")
.saveAsTable(target_table)
)
logger.info("☑ [DQXAlert] Guardado exitoso.")
except Exception as e:
logger.error(f"❌ [DQXAlert] Error guardando rechazados: {e}")
# No lanzamos excepción para permitir que el pipeline continúe si solo falla el log
else:
logger.info("⏭ [DQXAlert] Flag=0. Se omite guardado de rechazados.")

def evaluate_and_notify(
self,
metrics_dict: Dict[str, Any],
table_name: str,
```

```
pipeline_run_id: str,
dqx_run_id: str,
audit_table: Optional[str] = None
) -> None:
"""
Engatilla la alerta SOLO si hay errors o warnings en el metrics_dict.
"""
# 1. Evaluar Métricas (Gatekeeper)
errors = int(metrics_dict.get("error_row_count", 0))
warnings = int(metrics_dict.get("warning_row_count", 0))

if errors == 0 and warnings == 0:
logger.info(f"☑ [DQXAlert] Calidad OK ({table_name}). No se envía alerta.")
return

if not self.webhook_url:
logger.warning("⚠# [DQXAlert] Hay errores pero no se configuró Webhook URL.")
return

try:
audit_url = "https://adb-tu-workspace.azuredatabricks.net" # ⚠# Configurar Base URL
if audit_table:
audit_url += f"/explore/data/{audit_table.replace('.', '/')}"

template_data = {
"table_name": table_name,
"pipeline_run_id": pipeline_run_id,
"dqx_run_id": dqx_run_id,
"input_row_count": str(metrics_dict.get("input_row_count", 0)),
"error_row_count": str(errors),
"warning_row_count": str(warnings),
"audit_url": audit_url
}
```

```
json_template = Template(DQX_NOTIFY_TEMPLATE_STR)
data_json = json_template.safe_substitute(template_data)

logger.info(f" 🚨 [DQXAlert] Enviando notificación a Teams...")
response = requests.post(
self.webhook_url,
data=data_json.encode("utf-8"),
headers={'Content-Type': 'application/json'},
timeout=10
)

if response.status_code in [200, 202]:
logger.info(f"☑ [DQXAlert] Notificación enviada.")
else:
logger.error(f"❌ [DQXAlert] Error Webhook: {response.status_code} - {response.text}")

except Exception as e:
logger.error(f"❌ [DQXAlert] Fallo crítico en notificación: {e}")



#
# ============================================================================
# CLASE DQXQuality (Generador de Data y Esquemas)
#
# ============================================================================
class DQXQuality:
def __init__(self, observer_name: str = "dq_monitoring_observer") -> None:
self.ws = WorkspaceClient()
self.observer = DQMetricsObserver(name=observer_name)
self.engine = DQEngine(self.ws, observer=self.observer)

# --- MÉTODOS PÚBLICOS ---

def apply_checks_table(
self,
table_name: str,
```

```
pipeline_run_id: str = "-1",
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> Tuple[Dict, DataFrame, DataFrame, DataFrame]:
"""Aplica reglas a una tabla Delta/Hive existente."""
spark = SparkSession.builder.getOrCreate()
return self._execute_check_logic(
df_target=spark.table(table_name),
input_ref=table_name,
dataset_name=table_name,
pipeline_run_id=pipeline_run_id,
quality_check_filename=quality_check_filename,
rules_path=rules_path
)

def apply_checks_dataframe(
self,
df: DataFrame,
dataset_name: str = "dataframe_memory",
pipeline_run_id: str = "-1",
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> Tuple[Dict, DataFrame, DataFrame, DataFrame]:
"""Aplica reglas a un DataFrame en memoria."""
return self._execute_check_logic(
df_target=df,
input_ref=dataset_name,
dataset_name=dataset_name,
pipeline_run_id=pipeline_run_id,
quality_check_filename=quality_check_filename,
rules_path=rules_path
)

# --- LÓGICA INTERNA ---
```

```
def _execute_check_logic(self, df_target: Any, input_ref: Any, dataset_name: str, pipeline_run_id: str, quality_check_filename: Optional[str], rules_path: Optional[str]):

try:

# 1. Cargar Reglas

ref_for_path = input_ref if isinstance(input_ref, str) else dataset_name

final_rules_path = get_rules_path(ref_for_path, dataset_name, quality_check_filename, rules_path)

dq_rules_raw = self.engine.load_checks(WorkspaceFileChecksStorageConfig(final_rules_path))

# 2. Ejecutar DQX

df_enriched, observation = self.engine.apply_checks_by_metadata(df_target, dq_rules_raw)

df_enriched.cache()

# 3. Obtener Métricas y RunID nativo

metrics_dict = observation.get

dqx_run_id = metrics_dict.get("run_id", str(uuid.uuid4()))

# 4. Construir DF SUMMARY

df_summary = self._build_summary_df(metrics_dict, dataset_name, pipeline_run_id, dqx_run_id)

# 5. Construir DF REJECTED

df_valid, df_rejected_raw = filter_to_return_final_dfs(df_enriched)

df_rejected_final = df_rejected_raw.select(

F.col("row_values"),

F.col("_errors").alias("errors"),

F.col("_warnings").alias("warnings"),

F.lit(dataset_name).alias("nombre_tabla"),

F.lit(pipeline_run_id).alias("pipeline_run_id"),

F.lit(dqx_run_id).alias("dqx_run_id")

)

return metrics_dict, df_summary, df_valid, df_rejected_final

except Exception as e:

logger.error(f"Error DQX Quality: {e}")
```

```
if 'df_enriched' in locals(): df_enriched.unpersist()
raise e

def _build_summary_df(self, metrics: Dict, dataset_name: str, pipeline_run_id: str, dqx_run_id: str) -> DataFrame:
rows = [(
int(metrics.get("input_row_count", 0)),
int(metrics.get("valid_row_count", 0)),
int(metrics.get("error_row_count", 0)),
int(metrics.get("warning_row_count", 0))
)]
schema = T.StructType([
T.StructField("input_row_count", T.LongType(), True),
T.StructField("valid_row_count", T.LongType(), True),
T.StructField("error_row_count", T.LongType(), True),
T.StructField("warning_row_count", T.LongType(), True)
])
extra_cols = [
F.current_timestamp().alias("fecha_actual"),
F.lit(dataset_name).alias("nombre_tabla"),
F.lit(pipeline_run_id).alias("pipeline_run_id"),
F.lit(dqx_run_id).alias("dqx_run_id")
]
return create_dataframe(schema, rows, extra_cols)

#
==============================================================================
# CLASE DQXProfiling (Profiling para Tablas y DataFrames)
#
==============================================================================
class DQXProfiling:
def __init__(self, workspace_client: Optional[WorkspaceClient] = None, model_name: str = DEFAULT_LLM_MODEL) -> None:
self.ws = workspace_client or WorkspaceClient()
self.llm_config = LLMModelConfig(model_name=model_name)
```

```
self.profiler = DQProfiler(self.ws, llm_model_config=self.llm_config)

self.generator = DQGenerator(self.ws, llm_model_config=self.llm_config)

self.engine = DQEngine(self.ws)

logger.info(f"DQXProfiling inicializado (Modelo: {model_name})")


def profile_table(self, table_name: str, columns: Optional[List[str]] = None, options:
Optional[Dict[str, Any]] = None, quality_check_filename: Optional[str] = None, save_path:
Optional[str] = None) -> DataFrame:

"""Profiling sobre una Tabla Delta/Hive."""

logger.info(f"Profiling tabla: {table_name}")

stats, profiles = self.profiler.profile_table(input_config=InputConfig(location=table_name),
columns=columns, options=options)

return self._finalize_profiling(table_name, stats, profiles, quality_check_filename, save_path)


def profile_dataframe(self, df: DataFrame, dataset_name: str = "dataframe_memory", columns:
Optional[List[str]] = None, options: Optional[Dict[str, Any]] = None, quality_check_filename:
Optional[str] = None, save_path: Optional[str] = None) -> DataFrame:

"""Profiling sobre un DataFrame en memoria."""

logger.info(f"Profiling dataframe: {dataset_name}")

stats, profiles = self.profiler.profile(df, columns=columns, options=options)

return self._finalize_profiling(dataset_name, stats, profiles, quality_check_filename, save_path)


def _finalize_profiling(self, identifier: str, stats: Dict[str, Any], profiles: Any,
quality_check_filename: Optional[str], save_path: Optional[str]) -> DataFrame:

rules = self.generator.generate_dq_rules(profiles)

final_path = get_rules_path(identifier, identifier, quality_check_filename, save_path)

self.engine.save_checks(rules, WorkspaceFileChecksStorageConfig(final_path))

return self._summary_stats_to_df(stats)


def _summary_stats_to_df(self, summary_stats: Dict[str, Any]) -> DataFrame:

columns_data = summary_stats.get("columns", summary_stats)

rows = extrayendo_summary_stats(columns_data)

schema = T.StructType([

T.StructField("columna", T.StringType(), True), T.StructField("total", T.LongType(), True),

T.StructField("nulos", T.LongType(), True), T.StructField("cardinalidad", T.LongType(), True),

T.StructField("min", T.StringType(), True), T.StructField("max", T.StringType(), True),
```

```
T.StructField("media", T.StringType(), True), T.StructField("desviacion_estandar", T.StringType(), True)

])

extra_cols = [F.when(F.col("total") > 0, F.round((F.col("nulos") / F.col("total")) * 100, 2)).otherwise(F.lit(0.0)).alias("porcentaje_nulos")]

return create_dataframe(schema, rows, extra_cols)
```

```
from typing import Union, Dict, Optional, Any, List, Tuple
import json
import requests
import uuid
from string import Template

from pyspark.sql import DataFrame, SparkSession
from pyspark.sql import functions as F, types as T # noqa

# --- DQX Imports ---
from databricks.labs.dqx.profiler.profiler import DQProfiler
from databricks.labs.dqx.profiler.generator import DQGenerator
from databricks.labs.dqx.metrics_observer import DQMetricsObserver
from databricks.labs.dqx.config import InputConfig, WorkspaceFileChecksStorageConfig, LLMModelConfig
from databricks.labs.dqx.engine import DQEngine
from databricks.sdk import WorkspaceClient

# --- Local Config ---
from dqx_config import (
init_logging,
get_rules_path,
extrayendo_summary_stats,
create_dataframe,
get_rejected_df,
DEFAULT_LLM_MODEL,
DQX_NOTIFY_TEMPLATE_STR
)

logger = init_logging("dqx_quality")

#
==========================================================================
# CLASE INTERNA: _DQXPersistence (I/O Simplificado)
#
==========================================================================
```

```
class _DQXPersistence:
"""
Maneja el guardado de tablas y envío de alertas.
Asume que el Catálogo y Esquema ya existen en el ambiente (ctl_desa/prod).
"""
def __init__(self, webhook_url: Optional[str] = None):
self.webhook_url = webhook_url

def save_summary(self, df_summary: DataFrame, target_table: str) -> None:
"""
Guarda el historial de ejecución.
Spark creará la tabla automáticamente si no existe (siempre que el esquema exista).
"""
try:
logger.info(f"📊 Guardando métricas en: {target_table}")
(df_summary.write
.format("delta")
.mode("append")
.option("mergeSchema", "true")
.saveAsTable(target_table)
)
except Exception as e:
logger.error(f"❌ Error guardando summary en {target_table}: {e}")

def save_rejected(self, df_rejected: DataFrame, target_table: str) -> None:
"""
Guarda registros rechazados.
"""
try:
logger.info(f"💾 Guardando rechazados en: {target_table}")
(df_rejected.write
.format("delta")
.mode("append")
.option("mergeSchema", "true")
```

```
.saveAsTable(target_table)
)
logger.info("☑ Guardado de rechazados exitoso.")
except Exception as e:
logger.error(f"❌ Error guardando rechazados en {target_table}: {e}")

def evaluate_and_notify(self, metrics: Dict, table_name: str, pipeline_run_id: str, dqx_run_id: str, audit_table: Optional[str]) -> None:
"""Envía alerta a Teams si hay errores."""
errors = int(metrics.get("error_row_count", 0))
warnings = int(metrics.get("warning_row_count", 0))

if errors == 0 and warnings == 0:
logger.info(f"☑ Calidad OK. No se envía alerta.")
return

if not self.webhook_url:
logger.warning("⚠# Hay errores pero no hay Webhook URL configurado.")
return

try:
# Construcción dinámica del enlace al Data Explorer
audit_url = "https://adb-6151277180747240.0.azuredatabricks.net" # URL Base Fija
if audit_table:
audit_url += f"/explore/data/{audit_table.replace('.', '/')}"

template_data = {
"table_name": table_name,
"pipeline_run_id": pipeline_run_id,
"dqx_run_id": dqx_run_id,
"input_row_count": str(metrics.get("input_row_count", 0)),
"error_row_count": str(errors),
"warning_row_count": str(warnings),
"audit_url": audit_url
}
```

```
json_template = Template(DQX_NOTIFY_TEMPLATE_STR)
data_json = json_template.safe_substitute(template_data)

logger.info(f" 🚨 Enviando notificación a Teams...")
response = requests.post(
self.webhook_url,
data=data_json.encode("utf-8"),
headers={'Content-Type': 'application/json'},
timeout=10
)

if response.status_code not in [200, 202]:
logger.error(f"❌ Error Webhook: {response.status_code} - {response.text}")
else:
logger.info("☑ Notificación enviada.")

except Exception as e:
logger.error(f"❌ Fallo crítico en notificación: {e}")



#
# ============================================================================
# CLASE PRINCIPAL: DQXQuality (Facade)
#
# ============================================================================
class DQXQuality:
def __init__(self, observer_name: str = "dq_monitoring_observer") -> None:
self.ws = WorkspaceClient()
self.observer = DQMetricsObserver(name=observer_name)
self.engine = DQEngine(self.ws, observer=self.observer)

# --- MÉTODOS PÚBLICOS ---

def apply_checks_table(
```

```
self,
table_name: str,
pipeline_run_id: str,
summary_table: str, # OBLIGATORIO
audit_table: Optional[str] = None, # Opcional (si flag=1)
save_rejected_flag: int = 1, # 1=Guardar, 0=No
webhook_url: Optional[str] = None,
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> DataFrame:
"""
Aplica reglas a una tabla Delta/Hive existente.
Guarda métricas y alertas internamente.
"""
spark = SparkSession.builder.getOrCreate()
return self._run_internal_flow(
df_target=spark.table(table_name),
input_ref=table_name,
dataset_name=table_name,
pipeline_run_id=pipeline_run_id,
summary_table=summary_table,
audit_table=audit_table,
save_rejected_flag=save_rejected_flag,
webhook_url=webhook_url,
quality_check_filename=quality_check_filename,
rules_path=rules_path
)

def apply_checks_dataframe(
self,
df: DataFrame,
dataset_name: str,
pipeline_run_id: str,
summary_table: str,
```

```
audit_table: Optional[str] = None,
save_rejected_flag: int = 1,
webhook_url: Optional[str] = None,
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> DataFrame:
"""
Aplica reglas a un DataFrame en memoria.
"""
return self._run_internal_flow(
df_target=df,
input_ref=dataset_name,
dataset_name=dataset_name,
pipeline_run_id=pipeline_run_id,
summary_table=summary_table,
audit_table=audit_table,
save_rejected_flag=save_rejected_flag,
webhook_url=webhook_url,
quality_check_filename=quality_check_filename,
rules_path=rules_path
)

# --- LÓGICA INTERNA ---

def _run_internal_flow(self, df_target: DataFrame, input_ref: Any, dataset_name:
str, pipeline_run_id: str, summary_table: str, audit_table: str, save_rejected_flag: int,
webhook_url: str, quality_check_filename: str, rules_path: str) -> DataFrame:

# 1. Ejecutar Motor (Metrics + Rejected DF)
metrics, df_summary, df_rejected = self._execute_engine_logic(
df_target, input_ref, dataset_name, pipeline_run_id, quality_check_filename, rules_path
)

# 2. Persistencia
persistor = _DQXPersistence(webhook_url=webhook_url)
```

```
# A. Guardar Summary (Obligatorio)
persistor.save_summary(df_summary, summary_table)

# B. Guardar Rechazados (Condicional)
if save_rejected_flag == 1 and audit_table:
persistor.save_rejected(df_rejected, audit_table)

# C. Notificar
dqx_run_id = metrics.get("run_id", "unknown")
persistor.evaluate_and_notify(metrics, dataset_name, pipeline_run_id, dqx_run_id,
audit_table)

return df_rejected

def _execute_engine_logic(self, df_target: DataFrame, input_ref: Any, dataset_name: str,
pipeline_run_id: str, quality_check_filename: Optional[str], rules_path: Optional[str]):
try:
ref_for_path = input_ref if isinstance(input_ref, str) else dataset_name
final_rules_path = get_rules_path(ref_for_path, dataset_name, quality_check_filename,
rules_path)

dq_rules_raw = self.engine.load_checks(WorkspaceFileChecksStorageConfig(final_rules_path))

# Ejecución Lazy
df_enriched, observation = self.engine.apply_checks_by_metadata(df_target, dq_rules_raw)

# Materialización Obligatoria (Stall prevention)
df_enriched.cache()
_ = df_enriched.count()

metrics_dict = observation.get
dqx_run_id = metrics_dict.get("run_id", str(uuid.uuid4()))

# Construcción DFs
df_summary = self._build_summary_df(metrics_dict, dataset_name, pipeline_run_id,
dqx_run_id)
df_rejected_raw = get_rejected_df(df_enriched)

df_rejected_final = df_rejected_raw.select(
F.col("row_values"),
```

```
F.col("_errors").alias("errors"),
F.col("_warnings").alias("warnings"),
F.lit(dataset_name).alias("nombre_tabla"),
F.lit(pipeline_run_id).alias("pipeline_run_id"),
F.lit(dqx_run_id).alias("dqx_run_id")
)

return metrics_dict, df_summary, df_rejected_final

except Exception as e:
logger.error(f"Error Engine Interno: {e}")
if 'df_enriched' in locals(): df_enriched.unpersist()
raise e

def _build_summary_df(self, metrics: Dict, dataset_name: str, pipeline_run_id: str, dqx_run_id: str) -> DataFrame:
rows = [(int(metrics.get("input_row_count", 0)), int(metrics.get("valid_row_count", 0)), int(metrics.get("error_row_count", 0)), int(metrics.get("warning_row_count", 0)))]
schema = T.StructType([T.StructField("input_row_count", T.LongType(), True), T.StructField("valid_row_count", T.LongType(), True), T.StructField("error_row_count", T.LongType(), True), T.StructField("warning_row_count", T.LongType(), True)])
extra_cols = [F.current_timestamp().alias("fecha_actual"), F.lit(dataset_name).alias("nombre_tabla"), F.lit(pipeline_run_id).alias("pipeline_run_id"), F.lit(dqx_run_id).alias("dqx_run_id")]
return create_dataframe(schema, rows, extra_cols)

# ==============================================================================
# CLASE DQXProfiling (Se mantiene igual)
# ==============================================================================
class DQXProfiling:
def __init__(self, workspace_client: Optional[WorkspaceClient] = None, model_name: str = DEFAULT_LLM_MODEL) -> None:
self.ws = workspace_client or WorkspaceClient()
self.llm_config = LLMModelConfig(model_name=model_name)
self.profiler = DQProfiler(self.ws, llm_model_config=self.llm_config)
self.generator = DQGenerator(self.ws, llm_model_config=self.llm_config)
```

```
self.engine = DQEngine(self.ws)


def profile_table(self, table_name: str, columns: Optional[List[str]] = None, options: Optional[Dict[str, Any]] = None, quality_check_filename: Optional[str] = None, save_path: Optional[str] = None) -> DataFrame:

stats, profiles = self.profiler.profile_table(input_config=InputConfig(location=table_name), columns=columns, options=options)

return self._finalize_profiling(table_name, stats, profiles, quality_check_filename, save_path)


def profile_dataframe(self, df: DataFrame, dataset_name: str = "dataframe_memory", columns: Optional[List[str]] = None, options: Optional[Dict[str, Any]] = None, quality_check_filename: Optional[str] = None, save_path: Optional[str] = None) -> DataFrame:

stats, profiles = self.profiler.profile(df, columns=columns, options=options)

return self._finalize_profiling(dataset_name, stats, profiles, quality_check_filename, save_path)


def generate_rules_from_prompt(self, user_prompt: str, input_data: Union[str, Any], options: Optional[Dict[str, Any]] = None, quality_check_filename: Optional[str] = None, save_path: Optional[str] = None) -> None:

target_name = input_data if isinstance(input_data, str) else "dataframe_memory"

if isinstance(input_data, str):

checks = self.generator.generate_dq_rules_ai_assisted(user_input=user_prompt, input_config=InputConfig(location=input_data))

else:

stats, _ = self.profiler.profile(input_data, options=options)

checks = self.generator.generate_dq_rules_ai_assisted(user_input=user_prompt, summary_stats=stats)

final_path = get_rules_path(target_name, target_name if isinstance(input_data, str) else None, quality_check_filename, save_path)

self.engine.save_checks(checks, WorkspaceFileChecksStorageConfig(final_path))


def _finalize_profiling(self, identifier: str, stats: Dict[str, Any], profiles: Any, quality_check_filename: Optional[str], save_path: Optional[str]) -> DataFrame:

rules = self.generator.generate_dq_rules(profiles)

final_path = get_rules_path(identifier, identifier, quality_check_filename, save_path)

self.engine.save_checks(rules, WorkspaceFileChecksStorageConfig(final_path))

return self._summary_stats_to_df(stats)


def _summary_stats_to_df(self, summary_stats: Dict[str, Any]) -> DataFrame:

columns_data = summary_stats.get("columns", summary_stats)
```

```
rows = extrayendo_summary_stats(columns_data)

schema = T.StructType([

T.StructField("columna", T.StringType(), True), T.StructField("total", T.LongType(), True),

T.StructField("nulos", T.LongType(), True), T.StructField("cardinalidad", T.LongType(), True),

T.StructField("min", T.StringType(), True), T.StructField("max", T.StringType(), True),

T.StructField("media", T.StringType(), True), T.StructField("desviacion_estandar", T.StringType(),
True)

])

extra_cols = [F.when(F.col("total") > 0, F.round((F.col("nulos") / F.col("total")) * 100,
2)).otherwise(F.lit(0.0)).alias("porcentaje_nulos")]

return create_dataframe(schema, rows, extra_cols)
```

```
from typing import Union, Dict, Optional, Any, List, Tuple

# --- Spark Imports ---
from pyspark.sql import DataFrame, SparkSession
import pyspark.sql.functions as F
from pyspark.sql.types import (
StructType,
StructField,
StringType,
LongType,
NumericType
)

# --- DQX Imports ---
from databricks.labs.dqx.profiler.profiler import DQProfiler
from databricks.labs.dqx.profiler.generator import DQGenerator
from databricks.labs.dqx.metrics_observer import DQMetricsObserver
from databricks.labs.dqx.config import (
InputConfig,
WorkspaceFileChecksStorageConfig,
LLMModelConfig
)
from databricks.labs.dqx.engine import DQEngine
from databricks.sdk import WorkspaceClient

# --- Local Config ---
from dqx_config import (
init_logging,
get_rules_path,
extrayendo_summary_stats,
create_dataframe
)

logger = init_logging("dqx_quality")
```

```
#
# ============================================================================
# CLASE DQXProfiling
#
# ============================================================================
class DQXProfiling:
"""
Capa central de profiling y generación de reglas de calidad (DQX).
"""

def __init__(self, workspace_client: Optional[WorkspaceClient] = None) -> None:
self.ws = workspace_client or WorkspaceClient()
self.profiler = DQProfiler(self.ws)
self.generator = DQGenerator(self.ws)
self.engine = DQEngine(self.ws)
logger.info("DQXProfiling inicializado correctamente.")

def profile_table(
self,
table_name: str,
columns: Optional[List[str]] = None,
options: Optional[Dict[str, Any]] = None,
quality_check_filename: Optional[str] = None,
save_path: Optional[str] = None,
):
logger.info(f"Profiling tabla: {table_name}")

stats, profiles = self.profiler.profile_table(
input_config=InputConfig(location=table_name),
columns=columns,
options=options,
)

return self._finalize_profiling(
```

```
identifier=table_name,
stats=stats,
profiles=profiles,
quality_check_filename=quality_check_filename,
save_path=save_path,
)

def profile_dataframe(
self,
df: Any,
columns: Optional[List[str]] = None,
options: Optional[Dict[str, Any]] = None,
quality_check_filename: Optional[str] = None,
save_path: Optional[str] = None,
):
logger.info("Profiling DataFrame en memoria")

stats, profiles = self.profiler.profile(
df,
columns=columns,
options=options,
)

return self._finalize_profiling(
identifier="dataframe_memory",
stats=stats,
profiles=profiles,
quality_check_filename=quality_check_filename,
save_path=save_path,
)

def _finalize_profiling(
self,
identifier: str,
```

```
stats: Dict[str, Any],
profiles: Any,
quality_check_filename: Optional[str],
save_path: Optional[str],
):
rules = self.generator.generate_dq_rules(profiles)

final_path = get_rules_path(
identifier,
identifier,
quality_check_filename,
save_path,
)

self.engine.save_checks(
rules,
WorkspaceFileChecksStorageConfig(final_path),
)

logger.info(f"Reglas de profiling guardadas en: {final_path}")
return self._summary_stats_to_df(stats)

def _summary_stats_to_df(self, summary_stats: Dict[str, Any]):
try:
columns_data = summary_stats.get("columns", summary_stats)
rows = extrayendo_summary_stats(columns_data)

schema = StructType([
StructField("columna", StringType(), True),
StructField("total", LongType(), True),
StructField("nulos", LongType(), True),
StructField("cardinalidad", LongType(), True),
StructField("min", StringType(), True),
StructField("max", StringType(), True),
```

```
StructField("media", StringType(), True),
StructField("desviacion_estandar", StringType(), True),
])

extra_cols = [
F.when(F.col("total") > 0,
F.round((F.col("nulos") / F.col("total")) * 100, 2))
.otherwise(F.lit(0.0)).alias("porcentaje_nulos")
]

return create_dataframe(schema, rows, extra_cols)

except Exception as e:
raise e

def generate_rules_from_prompt(
self,
user_prompt: str,
input_data: Union[str, Any],
model_name: str = "databricks/databricks-claude-sonnet-4-5",
options: Optional[Dict[str, Any]] = None,
quality_check_filename: Optional[str] = None,
save_path: Optional[str] = None,
) -> None:
target_name = input_data if isinstance(input_data, str) else "dataframe_memory"
logger.info(f"IA Rules Generation | Target={target_name} | Model={model_name}")

try:
llm_config = LLMModelConfig(model_name=model_name)
ai_generator = DQGenerator(self.ws, llm_model_config=llm_config)

if isinstance(input_data, str):
checks = ai_generator.generate_dq_rules_ai_assisted(
user_input=user_prompt,
```

```
input_config=InputConfig(location=input_data),
)
else:
stats, _ = self.profiler.profile(
input_data,
options=options,
)
checks = ai_generator.generate_dq_rules_ai_assisted(
user_input=user_prompt,
summary_stats=stats,
)

final_path = get_rules_path(
target_name,
target_name if isinstance(input_data, str) else None,
quality_check_filename,
save_path,
)

self.engine.save_checks(checks, WorkspaceFileChecksStorageConfig(final_path))
logger.info(f"Reglas IA guardadas en: {final_path}")

except Exception as e:
logger.error(f"Error generando reglas con IA: {e}")
raise e



#
==============================================================================
# CLASE DQXQuality (Lógica Estricta: Error OR Warning = Fallo)
#
==============================================================================
class DQXQuality:

def __init__(self, observer_name: str = "dq_monitoring_observer") -> None:
self.ws = WorkspaceClient()
```

```
self.observer = DQMetricsObserver(name=observer_name)
self.engine = DQEngine(self.ws, observer=self.observer)
logger.info(f"DQXQuality (DQX Native) inicializado.")

def apply_checks_table(
self,
table_name: str,
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> Tuple[DataFrame, DataFrame, DataFrame]:

spark = SparkSession.builder.getOrCreate()
logger.info(f"Applying checks on Table: {table_name}")

return self._execute_check_logic(
spark.table(table_name),
table_name,
table_name,
quality_check_filename,
rules_path
)

def apply_checks_dataframe(
self,
df: Any,
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> Tuple[DataFrame, DataFrame, DataFrame]:

logger.info(f"Applying checks on DataFrame en Memoria")

return self._execute_check_logic(
df,
"dataframe_memory",
"dataframe_memory",
quality_check_filename,
rules_path
```

```
)

def _execute_check_logic(
self,
df_target: Any,
input_ref: Any,
dataset_name: str,
quality_check_filename: Optional[str],
rules_path: Optional[str]
) -> Tuple[DataFrame, DataFrame, DataFrame]:

try:
spark = SparkSession.builder.getOrCreate()
ref_for_path = input_ref if isinstance(input_ref, str) else dataset_name

final_rules_path = get_rules_path(
ref_for_path,
dataset_name,
quality_check_filename,
rules_path
)

dq_rules_raw = self.engine.load_checks(
WorkspaceFileChecksStorageConfig(final_rules_path)
)

# 1. EJECUCIÓN (Lazy)
df_enriched, observation = self.engine.apply_checks_by_metadata(
df_target,
dq_rules_raw
)

df_enriched.cache()
count_res = df_enriched.count()
logger.info(f"Checks procesados. Total filas: {count_res}")
```

```
# 2. SUMMARY (KPIs)
metrics_dict = observation.get
df_summary = self._build_summary_df(spark, metrics_dict, dataset_name)

# 3. SPLIT INTELIGENTE (MODO ESTRICTO: Error OR Warning = Rechazo)

dqx_cols_to_drop = ["_errors", "_warnings"]

# --- DEFINICIÓN DE LÓGICA (Strict Null-based) ---

# RECHAZADO: Tiene errores O tiene warnings.
# Usamos | (OR Bitwise) y paréntesis obligatorios.
condicion_fallido = (F.col("_errors").isNotNull()) | (F.col("_warnings").isNotNull())

# VÁLIDO: No tiene errores Y no tiene warnings.
# Usamos & (AND Bitwise) y paréntesis obligatorios.
condicion_valido = (F.col("_errors").isNull()) & (F.col("_warnings").isNull())

# --- BIFURCACIÓN ---

# A) Datos Válidos: Completamente limpios
df_valid = df_enriched.filter(condicion_valido).drop(*dqx_cols_to_drop)

# B) Datos Rechazados: Tienen algún defecto (Error o Warning)
df_failed_raw = df_enriched.filter(condicion_fallido)

# Identificamos columnas de negocio
business_cols = [c for c in df_enriched.columns if c not in dqx_cols_to_drop]

# Transformación VARIANT (DBR 15.3+)
df_rejected = df_failed_raw.select(
F.parse_json(
F.to_json(F.struct(*[F.col(c) for c in business_cols]))
).alias("row_values"),
F.col("_errors"),
F.col("_warnings"),
F.current_timestamp().alias("fecha_auditoria")
)

return df_summary, df_valid, df_rejected
```

```
except Exception as e:
logger.error(f"Error aplicando checks: {e}")
if 'df_enriched' in locals(): df_enriched.unpersist()
raise e


def _build_summary_df(self, spark: SparkSession, metrics: Dict, dataset_name: Optional[str] = None):
rows = [(
int(metrics.get("input_row_count", 0)),
int(metrics.get("valid_row_count", 0)),
int(metrics.get("error_row_count", 0)),
int(metrics.get("warning_row_count", 0))
)]

schema = StructType([
StructField("total_filas", LongType(), True),
StructField("filas_validas", LongType(), True),
StructField("error_filas", LongType(), True),
StructField("advertencias_filas", LongType(), True)
])

extra_cols = [
F.current_timestamp().alias("fecha_actual"),
F.lit(dataset_name).alias("nombre_tabla"),
F.lit(-1).alias("pipeline_run_id")
]

return create_dataframe(schema, rows, extra_cols)
```

```
from typing import Union, Dict, Optional, Any, List, Tuple

# Best Practice 1.1: Standard PySpark imports with aliases
from pyspark.sql import DataFrame, SparkSession
from pyspark.sql import functions as F, types as T # noqa

# --- DQX Imports ---
from databricks.labs.dqx.profiler.profiler import DQProfiler
from databricks.labs.dqx.profiler.generator import DQGenerator
from databricks.labs.dqx.metrics_observer import DQMetricsObserver
from databricks.labs.dqx.config import (
InputConfig,
WorkspaceFileChecksStorageConfig,
LLMModelConfig
)
from databricks.labs.dqx.engine import DQEngine
from databricks.sdk import WorkspaceClient

# --- Local Config ---
from dqx_config import (
init_logging,
get_rules_path,
extrayendo_summary_stats,
create_dataframe,
filter_to_return_final_dfs
)

logger = init_logging("dqx_quality")

#
==============================================================================
# CLASE DQXProfiling
#
==============================================================================
class DQXProfiling:
```

"""

Capa central de profiling y generación de reglas de calidad (DQX).

Utiliza IA y estadísticas para generar reglas YAML automáticamente.

"""

def __init__(self, workspace_client: Optional[WorkspaceClient] = None) -> None:

self.ws = workspace_client or WorkspaceClient()

self.profiler = DQProfiler(self.ws)

self.generator = DQGenerator(self.ws)

self.engine = DQEngine(self.ws)

logger.info("DQXProfiling inicializado correctamente.")

def profile_table(

self,

table_name: str,

columns: Optional[List[str]] = None,

options: Optional[Dict[str, Any]] = None,

quality_check_filename: Optional[str] = None,

save_path: Optional[str] = None

) -> DataFrame:

"""

Perfila una tabla registrada en Unity Catalog/Hive.

Parameters

----

table_name : str

Nombre completo de la tabla (catálogo.esquema.tabla).

columns : List[str], optional

Lista de columnas específicas a perfilar.

options : Dict[str, Any], optional

Opciones avanzadas para DQX Profiler.

quality_check_filename : str, optional

Nombre base para el archivo YAML de reglas.

save_path : str, optional

```
Ruta forzada para guardar el YAML.

Returns
-------
DataFrame
DataFrame con las estadísticas sumarizadas del perfilamiento.
"""
logger.info(f"Profiling tabla: {table_name}")
stats, profiles = self.profiler.profile_table(
input_config=InputConfig(location=table_name),
columns=columns,
options=options,
)
return self._finalize_profiling(table_name, stats, profiles, quality_check_filename, save_path)

def profile_dataframe(
self,
df: Any,
columns: Optional[List[str]] = None,
options: Optional[Dict[str, Any]] = None,
quality_check_filename: Optional[str] = None,
save_path: Optional[str] = None
) -> DataFrame:
"""
Perfila un DataFrame en memoria (útil para archivos crudos o transformaciones).
"""
logger.info("Profiling DataFrame en memoria")
stats, profiles = self.profiler.profile(
df,
columns=columns,
options=options,
)
return self._finalize_profiling("dataframe_memory", stats, profiles, quality_check_filename,
save_path)
```

```
def _finalize_profiling(
self,
identifier: str,
stats: Dict[str, Any],
profiles: Any,
quality_check_filename: Optional[str],
save_path: Optional[str]
) -> DataFrame:
"""Genera reglas y guarda el resultado."""
rules = self.generator.generate_dq_rules(profiles)
final_path = get_rules_path(identifier, identifier, quality_check_filename, save_path)

self.engine.save_checks(rules, WorkspaceFileChecksStorageConfig(final_path))
logger.info(f"Reglas de profiling guardadas en: {final_path}")

return self._summary_stats_to_df(stats)


def _summary_stats_to_df(self, summary_stats: Dict[str, Any]) -> DataFrame:
"""Convierte el diccionario de estadísticas de DQX a un DataFrame Spark."""
columns_data = summary_stats.get("columns", summary_stats)
rows = extrayendo_summary_stats(columns_data)

# Best Practice: Use T. prefix for types
schema = T.StructType([
T.StructField("columna", T.StringType(), True),
T.StructField("total", T.LongType(), True),
T.StructField("nulos", T.LongType(), True),
T.StructField("cardinalidad", T.LongType(), True),
T.StructField("min", T.StringType(), True),
T.StructField("max", T.StringType(), True),
T.StructField("media", T.StringType(), True),
T.StructField("desviacion_estandar", T.StringType(), True),
])

# Best Practice: Use F. prefix
extra_cols = [
F.when(F.col("total") > 0,
```

```
F.round((F.col("nulos") / F.col("total")) * 100, 2))
.otherwise(F.lit(0.0)).alias("porcentaje_nulos")
]
return create_dataframe(schema, rows, extra_cols)

def generate_rules_from_prompt(
self,
user_prompt: str,
input_data: Union[str, Any],
model_name: str = "databricks/databricks-claude-sonnet-4-5",
options: Optional[Dict[str, Any]] = None,
quality_check_filename: Optional[str] = None,
save_path: Optional[str] = None
) -> None:
"""Genera reglas de calidad utilizando GenAI basado en un prompt de usuario."""
target_name = input_data if isinstance(input_data, str) else "dataframe_memory"
logger.info(f"IA Rules Generation | Target={target_name} | Model={model_name}")

try:
llm_config = LLMModelConfig(model_name=model_name)
ai_generator = DQGenerator(self.ws, llm_model_config=llm_config)

if isinstance(input_data, str):
checks = ai_generator.generate_dq_rules_ai_assisted(
user_input=user_prompt,
input_config=InputConfig(location=input_data)
)
else:
stats, _ = self.profiler.profile(input_data, options=options)
checks = ai_generator.generate_dq_rules_ai_assisted(
user_input=user_prompt,
summary_stats=stats
)

final_path = get_rules_path(
target_name,
```

```
target_name if isinstance(input_data, str) else None,
quality_check_filename,
save_path
)
self.engine.save_checks(checks, WorkspaceFileChecksStorageConfig(final_path))
logger.info(f"Reglas IA guardadas en: {final_path}")

except Exception as e:
logger.error(f"Error generando reglas con IA: {e}")
raise e


# ============================================================================
# CLASE DQXQuality
# ============================================================================
class DQXQuality:
"""
Motor de ejecución de Calidad de Datos.
Aplica reglas, genera métricas de control y separa datos válidos de rechazados.
"""

def __init__(self, observer_name: str = "dq_monitoring_observer") -> None:
self.ws = WorkspaceClient()
self.observer = DQMetricsObserver(name=observer_name)
self.engine = DQEngine(self.ws, observer=self.observer)
logger.info(f"DQXQuality (DQX Native) inicializado.")

def apply_checks_table(
self,
table_name: str,
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> Tuple[DataFrame, DataFrame, DataFrame]:
"""Aplica reglas a una tabla Delta/Hive."""
spark = SparkSession.builder.getOrCreate()
```

```
logger.info(f"Applying checks on Table: {table_name}")
return self._execute_check_logic(
spark.table(table_name),
table_name,
table_name,
quality_check_filename,
rules_path
)

def apply_checks_dataframe(
self,
df: Any,
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> Tuple[DataFrame, DataFrame, DataFrame]:
"""Aplica reglas a un DataFrame en memoria."""
logger.info(f"Applying checks on DataFrame en Memoria")
return self._execute_check_logic(
df,
"dataframe_memory",
"dataframe_memory",
quality_check_filename,
rules_path
)

def _execute_check_logic(
self,
df_target: Any,
input_ref: Any,
dataset_name: str,
quality_check_filename: Optional[str],
rules_path: Optional[str]
) -> Tuple[DataFrame, DataFrame, DataFrame]:
"""
```

Lógica central de validación.

Returns

Tuple[DataFrame, DataFrame, DataFrame]

1. df_summary: Estadísticas de ejecución (1 fila).

2. df_valid: Datos limpios (esquema original).

3. df_rejected: Datos rechazados (esquema VARIANT + metadata).

"""

try:

spark = SparkSession.builder.getOrCreate()

ref_for_path = input_ref if isinstance(input_ref, str) else dataset_name

final_rules_path = get_rules_path(ref_for_path, dataset_name, quality_check_filename, rules_path)

dq_rules_raw = self.engine.load_checks(WorkspaceFileChecksStorageConfig(final_rules_path))

df_enriched, observation = self.engine.apply_checks_by_metadata(df_target, dq_rules_raw)

df_enriched.cache()

count_res = df_enriched.count()

logger.info(f"Checks procesados. Total filas: {count_res}")

metrics_dict = observation.get

df_summary = self._build_summary_df(metrics_dict, dataset_name)

SPLIT OPTIMIZADO (Lógica externa en dqx_config)

df_valid, df_rejected = filter_to_return_final_dfs(df_enriched)

return df_summary, df_valid, df_rejected

except Exception as e:

logger.error(f"Error aplicando checks: {e}")

if 'df_enriched' in locals(): df_enriched.unpersist()

raise e

def _build_summary_df(self, metrics: Dict, dataset_name: Optional[str] = None) -> DataFrame:

"""Construye el DataFrame de resumen de ejecución."""

rows = [(

```
int(metrics.get("input_row_count", 0)),
int(metrics.get("valid_row_count", 0)),
int(metrics.get("error_row_count", 0)),
int(metrics.get("warning_row_count", 0))
)]

# Best Practice: T. prefix for types
schema = T.StructType([
T.StructField("total_filas", T.LongType(), True),
T.StructField("filas_validas", T.LongType(), True),
T.StructField("error_filas", T.LongType(), True),
T.StructField("advertencias_filas", T.LongType(), True)
])

extra_cols = [
F.current_timestamp().alias("fecha_actual"),
F.lit(dataset_name).alias("nombre_tabla"),
F.lit(-1).alias("pipeline_run_id")
]
return create_dataframe(schema, rows, extra_cols)
```

🧊 DQX Utilities: El Estándar de Calidad de Datos en Pacífico

Bienvenido. Si estás leyendo esto, es porque te importa que los datos que mueves por nuestro Lakehouse (RDV, UDV, DDV) sean confiables.

DQX Utilities es el módulo centralizado dentro de `pacifico_utils`[^1] que democratiza el uso de Databricks Labs DQX. Su objetivo es eliminar la fricción de escribir validaciones manuales, delegando esa tarea a dos agentes especializados potenciados por IA.

🤔 ¿Qué es y por qué lo usamos?

Esta librería divide la gestión de calidad en dos roles claros para cubrir todo el ciclo de vida del dato:

- **El Arquitecto (DQXProfiling):** Se usa en Desarrollo. Te ayuda a definir las reglas. Puede analizar tus datos automáticamente o traducir tus requerimientos de negocio a código técnico.
- **El Guardián (DQXQuality):** Se usa en Producción. Vive dentro de tus Workflows de Databricks[^2] y se encarga de ejecutar las reglas, frenar datos malos y reportar métricas.

🏗️# Integración con el Framework Pacífico

Esta utilidad ha sido diseñada para encajar nativamente en nuestra arquitectura moderna:

- **Modularidad:** Vive dentro de `pacifico_utils`, lista para ser importada en cualquier Notebook de transformación (`nb_*.py`)[^3].
- **Eficiencia (Smart Sampling):** Al perfilar DataFrames gigantes, la herramienta aplica un muestreo inteligente automático para obtener estadísticas representativas sin disparar los costos de cómputo.
- **Seguridad:** Al usar la generación con IA, el sistema envía solo metadatos y estadísticas, minimizando la exposición de datos sensibles.

Importación

En cualquier notebook de tu flujo de trabajo:

```python
from pacifico_utils.dqx_utils import DQXProfiling, DQXQuality
```

🛠️# Fase 1: Desarrollo (Diseñando las Reglas)

Estás creando un nuevo proceso o explorando una fuente. Necesitas definir qué es "buena calidad" para estos datos antes de automatizar nada.

Opción A: "El Negocio me dio las reglas" (Generación con IA)

Si tienes los criterios de aceptación claros (ej: ticket en Jira), no pierdas tiempo escribiendo YAML a mano.

Usa `generate_rules_from_prompt`. La herramienta es **híbrida**:

- Si le pasas un **Nombre de Tabla**, consulta el esquema en Unity Catalog.
- Si le pasas un **DataFrame** (ej. lectura de un Parquet en RDV), calcula estadísticas rápidas para darle contexto a la IA.

```python
profiler = DQXProfiling()

# Escribe las reglas tal como te las dieron (Lenguaje Natural)
PROMPT = """
1. El ID de cliente debe ser único y obligatorio.
2. El correo electrónico debe tener formato válido.
3. El monto de transacción no puede ser negativo.
4. Si el estado es 'ACTIVO', la fecha de baja debe ser nula.
"""

# Genera el archivo YAML de configuración en tu Repo
profiler.generate_rules_from_prompt(
    user_prompt=PROMPT,
    input_data="hive_metastore.udv_clientes.maestra",  # O un objeto DataFrame
    quality_check_filename="reglas_udv_clientes"
)
```

¿Qué ocurre aquí? La IA analiza la estructura de tus datos + tu pedido y crea un archivo técnico estándar listo para ser versionado.

Opción B: "No conozco la data" (Perfilamiento Automático)

Ideal para fuentes nuevas, ya sean archivos sueltos en RDV/Landing o tablas ya registradas en UDV. Deja que la herramienta escanee los datos y te diga qué patrones encuentra.

```python
# CASO 1: Archivos en Data Lake (Landing / RDV)
# Leemos el archivo Parquet (recordar que en RDV todo es texto/parquet)
df_raw = spark.read.parquet("abfss://landing@pacificolake.../archivos/")

profiler.profile_dataframe(
    df=df_raw,
    quality_check_filename="reglas_base_landing"
)

# CASO 2: Tablas Registradas (UDV / Unity Catalog / Hive Metastore)
# Perfilamos directamente usando el nombre de la tabla
profiler.profile_table(
```

```
    table_name="hive_metastore.udv_ventas.fact_transacciones",
    quality_check_filename="reglas_base_ventas"
)

# CASO 3: Tablas Delta por Ruta (UDV / Time Travel)
# Útil si accedes a la tabla Delta por su ruta física o una versión específica
df_delta = spark.read.format("delta").load("abfss://udv@pacificolake.../fact_transacciones")

profiler.profile_dataframe(
    df=df_delta,
    quality_check_filename="reglas_base_delta_path"
)
```

🏭 Fase 2: Producción (Ejecutando el Control)

Ya tienes tus reglas (el archivo YAML generado). Ahora es momento de operacionalizarlas en nuestro entorno moderno.

En nuestra arquitectura, la lógica de transformación vive exclusivamente en Azure Databricks. Data Factory actúa únicamente como disparador a través de sus Mallas (carpeta 4-SCH)[^4], ejecutando Workflows de Databricks[^5].

Tu notebook productivo (`nb_*.py`) es una Task dentro de ese Workflow[^6]. Ahí es donde debes invocar al motor de calidad.

Implementación en el Notebook (nb_*.py)

El control de calidad (DQXQuality) actúa como un **Quality Gate**: valida los datos transformados en memoria antes de que se escriban en la siguiente capa.

```python
# ... [Bloque previo: Lectura de parámetros y Transformación con scripts .py] ...
# df_final = transform_logic(...)

# ============================================================
# 🛡️ QUALITY GATE (DQX)
# ============================================================
from pacifico_utils.dqx_utils import DQXQuality

logger.info("Iniciando validación de calidad DQX...")

quality_engine = DQXQuality()

# Ejecuta las reglas sobre el DataFrame transformado (en memoria)
# Esto asegura que NUNCA escribamos basura en UDV/DDV
df_resumen, df_validado = quality_engine.apply_checks_table(
```

```
    table_name="hive_metastore.udv_finanzas.reporte_final",  # O pasar df_final
    quality_check_filename="reglas_udv_finanzas",  # Archivo YAML creado en Dev
    return_enriched=True
)

# 1. Observabilidad:
# Muestra los KPIs en los logs del Driver del Workflow para revisión rápida.
display(df_resumen)

# 2. Circuit Breaker (Cortacircuitos):
# Si detectamos errores críticos, fallamos la tarea para que ADF detenga la malla.
cnt_errores = df_resumen.filter("error_filas > 0").count()

if cnt_errores > 0:
    logger.error(f"🚫 CALIDAD FALLIDA: Se detectaron {cnt_errores} reglas rotas.")

    # Detener el Workflow protegiendo la integridad de la capa siguiente
    # raise Exception("DQX Validation Failed: Data Integrity Issues Detected")

logger.info("☑️ Calidad validada exitosamente. Procediendo a escritura.")
```

🌐 Guía de Uso por Fuente de Datos

Nuestro framework se adapta a la capa donde estés trabajando:

Escenario	Tipo de Input	Método Recomendado	Ejemplo
Tablas Unity Catalog (Capas UDV / DDV)	String (Nombre de la tabla)	Pasa el nombre directo. La herramienta consulta el esquema en el metastore.	`input_data="hive_metastore.schema.tabla"`
Archivos en Data Lake (Capas Landing / RDV)	DataFrame (Spark Object)	Lee el archivo con Spark primero y pasa el objeto DataFrame.	`df = spark.read.parquet(...)` `input_data=df`
DataFrames en Memoria (Transformaciones In-flight)	DataFrame (Spark Object)	Ideal para validar pasos intermedios dentro del Workflow.	`input_data=df_transformado`

🧠 Capacidades "Power User"

1. Detección de Llaves Primarias (Auto-PK)

¿Recibiste un dataset en RDV sin documentación y no sabes cuál es el ID único? DQXProfiling usa algoritmos inteligentes para sugerirte la llave primaria.

```python
# Funciona leyendo el parquet directo
df_rdv = spark.read.parquet("abfss://landing@...")
pk_sugerida = profiler.detect_primary_keys(df_rdv)

print(f"La PK probable es: {pk_sugerida}")
```

2. Modelos de IA a la Carta

Por defecto, usamos modelos optimizados de Databricks. Si necesitas probar otro modelo, puedes parametrizarlo usando Widgets en tu notebook de desarrollo:

```python
# Usar Llama 3 para este análisis específico
model_name = dbutils.widgets.get("llm_model")

profiler.generate_rules_from_prompt(
    user_prompt=...,
    model_name=model_name
)
```

💡 Mejores Prácticas

1. **Versionamiento:** Guarda los archivos YAML de reglas generados en tu repositorio Git (carpetas `framework_*/quality_rules`)[^7]. Son parte de tu código.

2. **Iteración:** La IA es una asistente, no magia. Revisa siempre el YAML generado (`reglas_ia_*.yml`) y ajústalo si es necesario.

3. **Validación en Memoria:** Siempre prefiere validar el DataFrame antes de escribir (`apply_checks_dataframe`). Es más eficiente y seguro que escribir y luego validar.

Equipo de Ingeniería de Datos - Pacífico Seguros

[^1]: Módulo de utilidades compartidas del equipo
[^2]: Orquestación nativa de Databricks
[^3]: Convención de nomenclatura para notebooks productivos
[^4]: Estructura de carpetas en Azure Data Factory
[^5]: Jobs programados en Databricks
[^6]: Unidad de ejecución dentro de un Workflow
[^7]: Convención de versionamiento de reglas de calidad

Azure Data Factory & Databricks

Git, Github & Azure Devops

Microsoft Excel y VBA

EDGAR QUISPE

CONTÁCTAME

+51 908 848 299

edgar.quispeq10@gmail.com

https://www.linkedin.com/in/edgar-quispe-151051167/

Chorrillos, Lima (Perú)

IDIOMAS

●●●●● INGLES **Intermedio**

CERTIFICACIONES

Fabric Analytics Engineer Associate
Microsoft

Power BI Data Analyst Associate
Microsoft

Azure Fundamentals
Microsoft

Azure Data Fundamentals
Microsoft

Databricks Fundamentals
Databricks

HERRAMIENTAS

Microsoft Fabric & Power BI

SQL Server, Oracle

SQL Server Analysis Services

Python for ETL y Web Scraping

Blob Storage & Azure SQL

RESUMEN

Dinámico y habituado al trabajo individual y en equipo; con gran sentido de responsabilidad, así como alta vocación de servicio, capaz de generar valor agregado a la organización a través de ideas para la resolución de problemas y automatización de tareas recurrentes.

EDUCACIÓN

Universidad mayor de San Marcos 03/2015 - 12/2021
Ingeniería de Sistemas e Informática, Ingeniería de Sistemas.

EXPERIENCIA

PROSEGUR PERÚ Lima, PER **Azure Data Engineer** 09/2025 – 01/2026
Encargado de la migración y mantenimiento de una arquitectura de datos moderna en Azure, utilizando Azure Databricks, Azure Data Factory, Synapse Analytics y Azure Analysis Services, así como el desarrollo de reportes en Power BI para la explotación de información.

INDRA PERÚ Lima, PER **Data Analyst Engineer** 10/2024 – 09/2025
Encargado del modelado y procesamiento de datos con Azure Data Factory y Databricks. Desarrollo de reportes en Power BI y optimización de bases de datos. Validación y definición de soluciones en conjunto con el negocio.

UMA CONSULTING Lima, PER **Especialista de Soluciones BI** 09/2023 – 09/2024 Diseñé, creé y desplegué un Data Warehouse en SQL Server para Unibanca. Implementé el proceso de carga histórica e incremental con SQL Server Integration Services. Desarrollo de reportes en Power BI para la explotación de datos y toma de decisiones.

CALA PROJECTS Montevideo, URY
Consultor Business Intelligence 11/2022 - 09/2023
Encargado de la automatización del proceso de extracción, transformación y carga masiva de datos a un Data Warehouse, asi como la implementación de informes interactivos en Power BI para el cliente Marriot.

ITACAMBA Santa Cruz, BOL
Analista de Business Intelligence 03/2022-11/2022
Gestioné el Proyecto BI que integró las diferentes fuentes de información de la empresa en un Data Warehouse para

ser explotado a través de varios informes en Power BI, optimizando la toma de decisiones basada en datos.

ADDC PERU Lima, PER
Instructor de cursos y capacitaciones 09/2020 - 06/2022

SODEXO PERU Lima, PER
Especialista de Indicadores 09/2019 – 11/2020

Arquitectura del Lakehouse

Bienvenido al equipo de **Data Engineering** en Pacífico Seguros.
En esta guía encontrarás lo **básico y necesario** para desenvolverte en nuestro ecosistema, entender cómo funcionan nuestras capas de datos y conocer las herramientas que utilizamos.

Utilitarios importantes para el Data Engineer en Pacífico

Para trabajar en nuestro ecosistema, es necesario contar con ciertos aplicativos y herramientas instaladas en tu equipo.

Aplicativos a instalar

- **Azure Data Studio** → Para conectarte y consultar bases de datos SQL.
- **Azure Storage Explorer** → Para gestionar archivos en Azure Blob Storage.
- **Visual Studio Code** → Editor de código para desarrollo y debugging.
- **Git** → Control de versiones y manejo de repositorios.

¿Cómo solicitar la instalación?

Puedes usar como referencia el siguiente ticket para realizar tu solicitud en el Service Desk: **Ejemplo de ticket:**
RITM0546081

Arquitectura del Lakehouse

A continuación se muestra la Arquitectura de nuestro Lakehouse.

Las capas principales de nuetro Lakehouse son las siguientes:

- **Landing:** Aplica unicamente para Data Entries y Fuentes Externas (MiBanco, BCP, Equifax, etc)
- **RDV (Raw Data Vault):** En la capa RDV aterrizan todos los datos sin ninguna modificacion, solo se convierte a texto todos sus campos.
- **UDV (Universal Data Vault):** En la capa UDV se tienen entidades de datos en función a reglas de negocio y necesidades generales. Asi mismo, a estas entidades se les aplica un proceso de calidad de datos.
- **DDV (Dimensional Data Vault):** En la capa DDV se tienen productos datos enfocados a la necesidad especifica de una area de negocio, estos pueden ser: Datamart's, Dataset's, Tablas con fin particular.

Los principales servicios cloud que usamos dentro de Ingeniería son los siguientes:

- **Azure Data Factory:** Es nuestra herramienta de ingesta de datos y orquestación de pipelines.
- **Azure Data Databricks:** Es nuestra herramienta de transfo**rmación de datos y orquestación de pipelines (en algunos casos, principalmente entidades DDV).
- **Azure SQL:** Es la herramienta que utilizamos para registrar los pipelines de datos que se crean y tambien la metadata principal de los mismos. Es el corazón de los pipelines de datos puesto que todos los pipelines de datos tienen comunicación con la misma durante sus ejecuciones. De ocurrir un error con este servicio ninguno de los pipelines de datos del lakehouse podría ejecutarse.

Nuestro Framework

En el Chapter de Ingeniería contamos con un framework de datos el cuál nos permite realizar ingestas desde distintas fuentes como: Base de Datos (Oracle, Sql Server), SFTP's, Data Entries, etc. Así como tambien procesarlas y aplicarles reglas de limpieza para que luego sean disponibilizadas a usuarios finales.

Modelo de control

El corazón de este Framework vive en una BD Azure SQL que es la encargada de orquestar toda la metadata de los pipelines. A continuación se presenta el modelo de datos que sostiene el framework:

Donde:

1. **tbl_pipeline:** Es la tabla principal donde se registran todos los pipelines.
    - **pipeline_id:** campo incremental autogenerado.
    - **src_name:** nombre de la tabla o fuente origen a ingestar.
    - **src_group_type:** nos sirve para identificar el origen, toma valores como: SqlServer, Oracle, Data entries.
    - **trg_name:** nombre destino de la fuente ingestada.
    - **trg_path:** ruta dentro del contenedor donde se almacenará los datos ingestados.
2. **tbl_pipeline_parameter:** Esta tabla sirve de apoyo para registrar todos los parametros de configuración que se necesitan para determinado pipeline.
    - **pipeline_parameter_id:** campo incremental autogenerado.
    - **pipeline_id:** identificador del pipeline y tambien sirve como llave foranea.
    - **paramter_name:** nombre del parámetro.
    - **parameter_value:** valor para el prámetro.
3. **tbl_predecessor_udv:** En esta tabla se registran todos los predecesores (inputs) que necesita un pipeline, por ejemplo: Una tabla DDV puede necesitar 3 o 4 predecesores UDV.
    - **src_pipeline_id:** identificador del pipeline predecesor.
    - **udv_pipeline_id:** identificador del pipeline destino, por un tema de modelamiento este campo se creo con el prefijo "udv" en lugar de "trg" que haga referencia a destino.
    - **udv_path_checkpoint:** es una ruta temporal a donde se van a copiar los datos de los pipelines predecesores, para que luego puedan ser utilizados.
4. **tbl_ctrl_pipeline_paths:** En esta tabla se registran las rutas de las fuentes ingestadas a nuestra capa RDV unicamente.
    - **pipeline_id:** identificador del pipeline.
    - **trg_path:** ruta donde se han almacenado los datos dentro de la capa RDV.
    - **pipeline_id:** identificador del pipeline predecesor.
    - **udv_pipeline_id:** identificador del pipeline destino, por un tema de modelamiento este campo se creo con el prefijo "udv" en lugar de "trg" que haga referencia a destino.
    - **flg_udv_load:** es un flag que utiliza nuestro framework para indicar si esa información ya fue utilizada por algun proceso sucesor (PREUDV, UDV, DDV).

5. **tbl_ctrl_pipeline_hist:** En esta tabla se registran todas las ejecuciones que pueda tener un pipeline. Almacena el historial de ejecuciones de toodos los pipelines.
    - **pipeline_id:** identificador del pipeline
    - **process_load_type:** indica el tipo de carga: FULL, INCREMENTAL.
    - **last_available_date:** indica la fecha mas reciente de la información del pipeline. Este es un campo opcional ya que en procesos FULL no suele tener valor, se asigna uno por defecto.
    - **execution_date:** indica cuando se ejecuto el pipeline.
6. **tbl_ctrl_pipeline:** En esta tabla se registran solo las ULTIMAS ejecuciones de cada pipeline.

De las tablas mencionadas el Data Engineer tiene que hacer registros solo en 3 de estas: **tbl_pipeline, tbl_pipeline_parameter y tbl_predecessor_udv**.

(*) En algunos casos el src_name y trg_name pueden tener el mismo valor. (**) Solo se ha mencionado los campos principales de las tablas de control.

Tipos de Carga en RDV

- **FULL:** son cargas donde ingestamos todos los registros, segun su frecuencia definida diaria, mensual, anual.
- **INCREMENTAL:** son cargas donde solo cargamos el diferencial con respecto a la ultima carga segun frecuencia definida.
- **INCREMENTAL (LAST N DAYS):** son cargas generalmente programadas diarias pero donde nos traemos no solo los nuevos registros, sino que retrocedemos "N" (dias, meses o años) para extraer la información, esto se hace debido a que en los CORE's generalmente se hacen modificaciones manuales.

Tipos de Carga en UDV, DDV

- **FULL:** son cargas donde reprocesamos todos los registros, segun su frecuencia definida diaria, mensual, anual.
- **INCREMENTAL:** son cargas donde solo actualizamos e insertamos el diferencial con respecto a la ultima carga segun frecuencia definida.
- **MERGE:** no es un tipo de carga como tal, pero se utiliza para indicar que el proceso ya hace uso de este tipo de configuración en nuestro flujo.

Proceso de Ingesta Capa RDV

Para la ingesta a la capa RDV ya se cuenta con multiples plantillas genéricas en Azure DataFactory las cuales solo requieren un nivel de parametrización, a nivel de las tablas de control. Estas plantillas según los parámetros registrados ya se encargan de realizar un tipo de carga específico sea FULL o INCREMENTAL.

Una particularidad que tenemos en el caso de la capa RDV es que todos los datos se almacenan en formato parquet y convertidos a texto, esto para evitar que se pierda la integridad del dato con respecto a su origen.

Proceso de Ingesta Capa UDV

De cara a la ingesta hacia UDV nuestros procesos utilizarán la información que se ha cargado a RAW. Nuestros procesos UDV legados se ejecutan desde Data Factory, sin embargo, ya nuestros nuevos procesos se ejecutan mediante workflows en Databricks que son disparados desde Data Factory.

Dentro del flujo UDV existen 2 casuísticas particulares de nuestro framework.

1. Relacion 1 a 1 entre RDV y UDV

Cuando tenemos un proceso UDV que se alimenta de entidades RDV no lo hace directamente de la ruta donde estas se encuentran, sino lo que hace nuestro framework es copiar estos datos a una ruta temporal y desde esta ruta temporal es donde se consumen. A este flujo le hemos llamado **CHECKPOINT**.

La particularidad del Checkpoint es que solo funciona cuando la entidad RDV es consumida por unicamente un proceso UDV, es decir, si existen 2 procesos que necesitan utilizar la misma RDV, esto no va ser factible a través del **CHECKPOINT**.

2. Relacion 1 a muchos entre RDV y UDV

Si tenemos 2 entidades UDV que necesitan consumir una misma entidad RDV, primero tenemos que construir una entidad en una **pseudo-capa a la cual llamamos PRE-UDV**, su ventaja es que se encuentra en **formato DELTA**, y esto permite que pueda ser utilizada por mas de una entidad UDV. En este caso el Checkpoint solo sería aplicado de RAW a PRE-UDV, sin embargo, para UDV solo se utilizaría PRE-UDV.

Proceso de Ingesta Capa DDV

En el caso de la construcción de entidades DDV, estas se alimentan principalmente de entidades UDV, pero en ciertas ocasiones muy puntuales lo pueden hacer tambien de PREUDV.

DataOps

Imagina que los datos son como agua que fluye por tuberías. DataOps es la práctica que asegura que ese flujo sea rápido, limpio y seguro, conectando a quienes construyen las tuberías (ingenieros de datos) con quienes necesitan el agua (usuarios de negocio). Su misión: automatizar, orquestar y monitorear todo el ciclo para que la información llegue donde debe, sin fricciones.

¿Qué es DataOps en pocas palabras?

Es una forma ágil y colaborativa de trabajar con datos que combina:

- Automatización para reducir tareas manuales.
- Observabilidad para saber qué pasa en cada paso.
- Gobierno y calidad para que los datos sean confiables.
- Colaboración entre equipos técnicos y de negocio.

Nuestro Ecosistema DataOps

Plantillas ingesta Data Factory

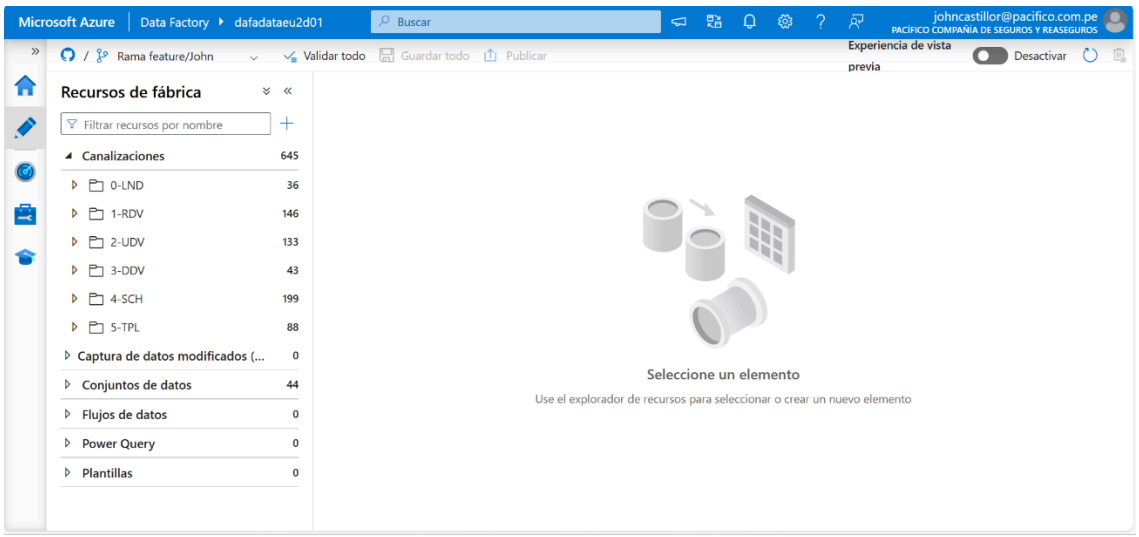
En nuestro ecosistema, **ADF (Azure Data Factory)** es el servicio principal para la **ingesta de datos desde los distintos cores hacia nuestro Data Lake**.

Este componente se organiza en **pipelines** que definen los flujos de carga y transformación inicial.

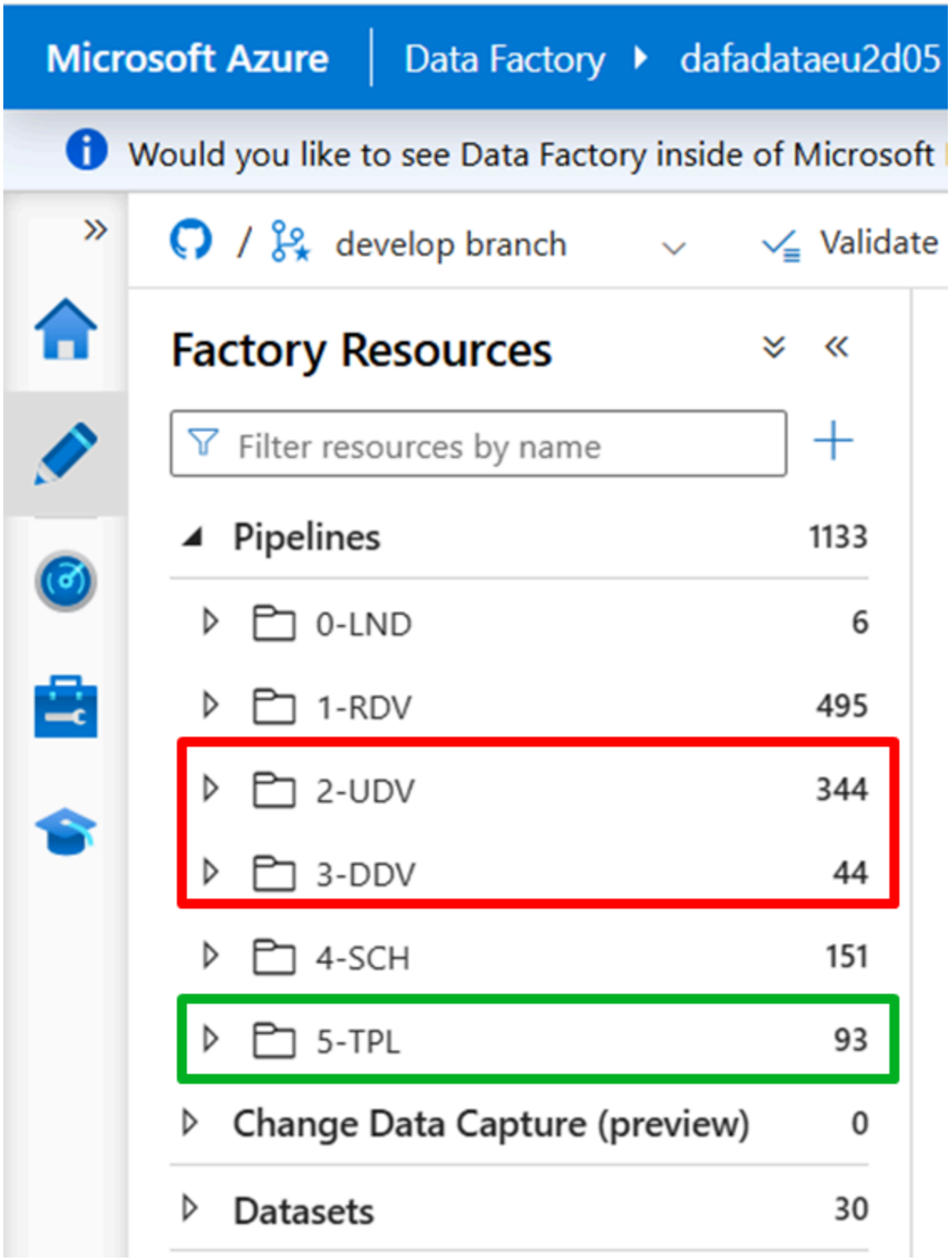
Contexto técnico

- Cada pipeline está diseñado para ser **modular y reutilizable**, evitando duplicación de lógica.
- La estructura se organiza por **capas**:
    - **0-LND** → Landing: datos crudos tal como llegan.
    - **1-RDV** → Raw Data Vault: datos estructurados con mínima transformación.
    - **2-UDV** → Unified Data Vault.
    - **3-DDV** → Data Driven Vault.
    - **4-SCH** → Scheduling y control.
    - **5-TPL** → **Plantillas genéricas** para ingesta, reutilizadas por diferentes cores.

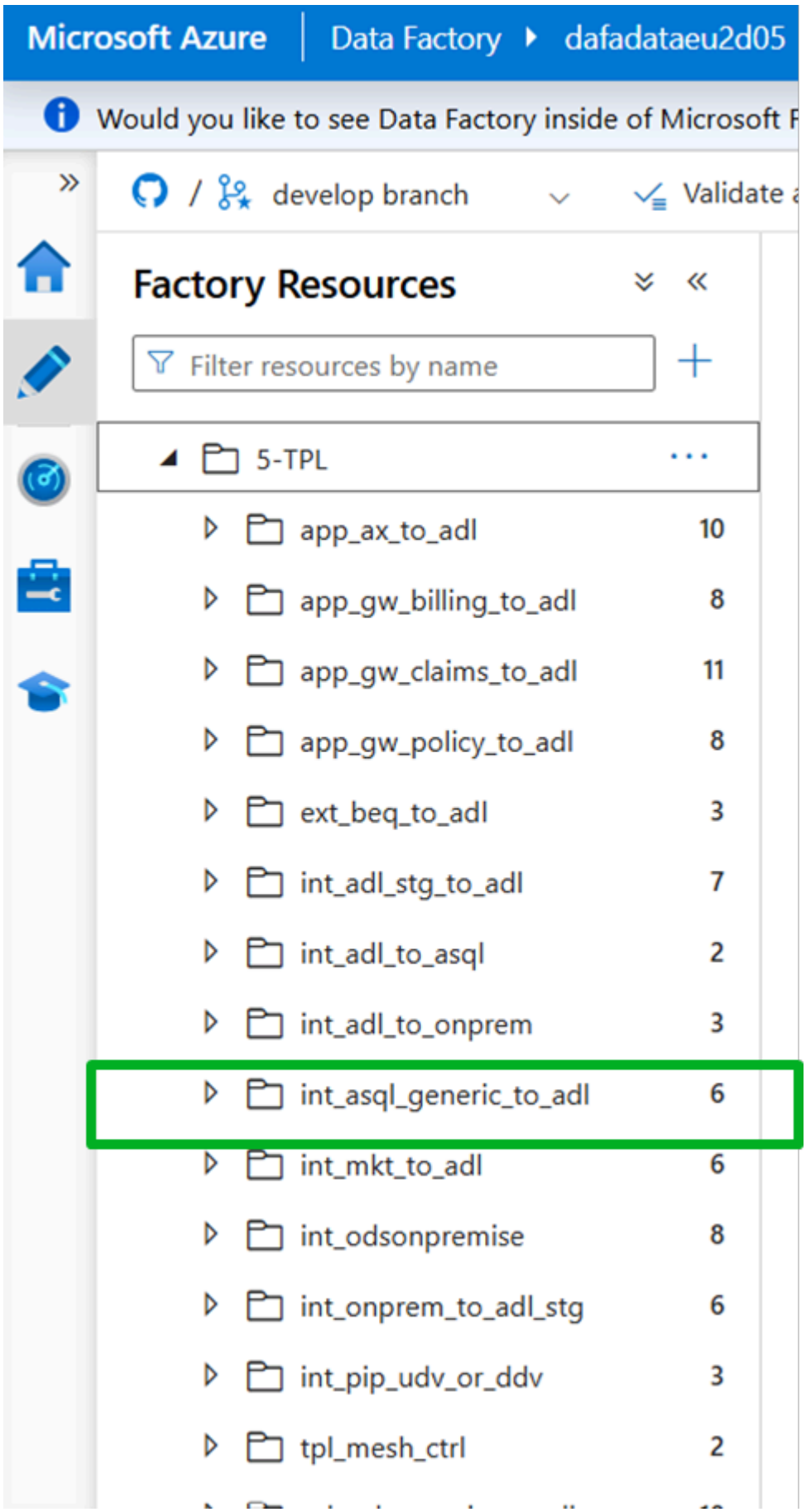
Estas capas permiten **separación de responsabilidades** y escalabilidad.



Por otro lado, en esta vista se aprecia la cantidad de pipelines por capa.
Nota: Las carpetas en **verde** son plantillas activas, mientras que las **rojas** indican pipelines que están en proceso de deprecación.



Detalle de la carpeta TPL


Microsoft Azure
Data Factory
dafadataeu2d05
Would you like to see Data Factory inside of Microsoft F
develop branch
Validate a
Factory Resources
Filter resources by name
5-TPL
app_ax_to_adl 10
app_gw_billing_to_adl 8
app_gw_claims_to_adl 11
app_gw_policy_to_adl 8
ext_beq_to_adl 3
int_adl_stg_to_adl 7
int_adl_to_asql 2
int_adl_to_onprem 3
int_asql_generic_to_adl 6
int_mkt_to_adl 6
int_odsonpremise 8
int_onprem_to_adl_stg 6
int_pip_udv_or_ddv 3
tpl_mesh_ctrl 2

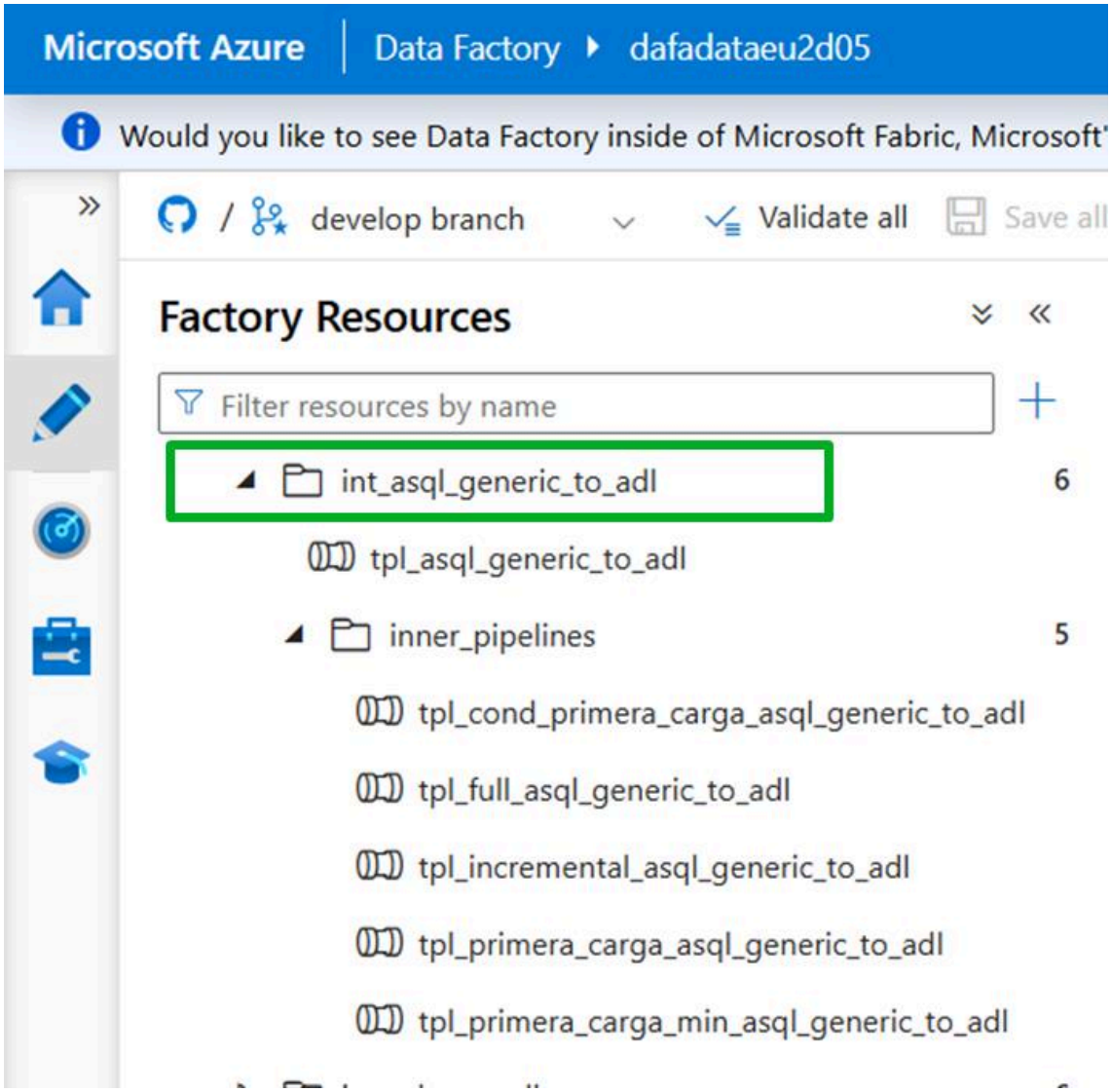
Dentro de la carpeta **5-TPL** se encuentran las **plantillas genéricas** que usamos para la ingesta de datos desde diferentes orígenes hacia el Data Lake.
Estas plantillas son **reutilizables** y están diseñadas para reducir la duplicación de lógica, permitiendo que con solo parametrización se adapten a distintos escenarios.

Ejemplo:

- int_asql_generic_to_adl → Plantilla para ingestar datos desde **Azure SQL** hacia **ADL (Azure Data Lake)**.
- int_onprem_to_adl_stg → Plantilla para ingestar datos desde **entornos OnPremise** hacia ADL.

¿Por qué es importante?
Estas plantillas son la base de la automatización en la capa de ingesta, asegurando consistencia y escalabilidad en los procesos.

Estructura interna de una plantilla



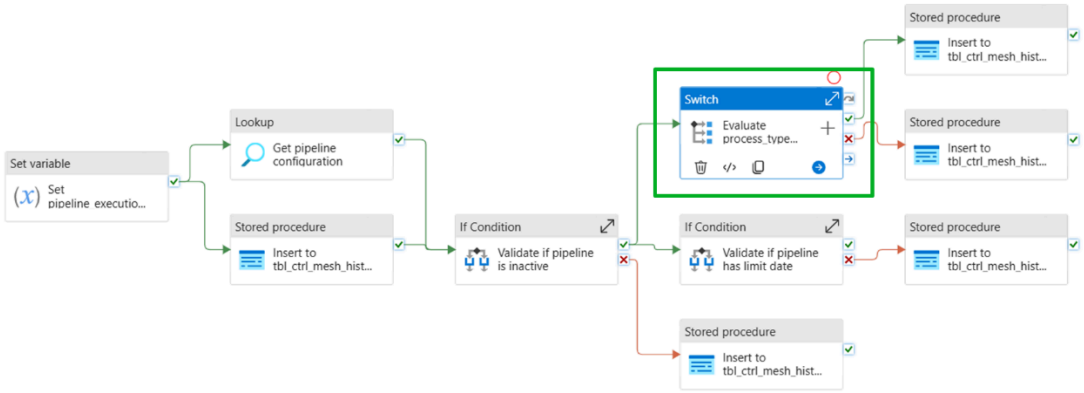
Cada plantilla está compuesta por:

- **Pipeline principal** → Orquesta la lógica general del proceso.

- **Inner pipelines (subplantillas)** → Se ejecutan según la condición del tipo de carga definida en los parámetros:
    - **FULL** → Carga completa.
    - **INCREMENTAL** → Solo datos nuevos o modificados.
    - **PRIMERA CARGA** → Inicialización del dataset.

¿Por qué esta estructura?
Permite que un solo pipeline maneje múltiples escenarios de carga sin necesidad de crear pipelines separados, lo que mejora la **modularidad** y **mantenibilidad** del ecosistema.

Tipos de carga soportados

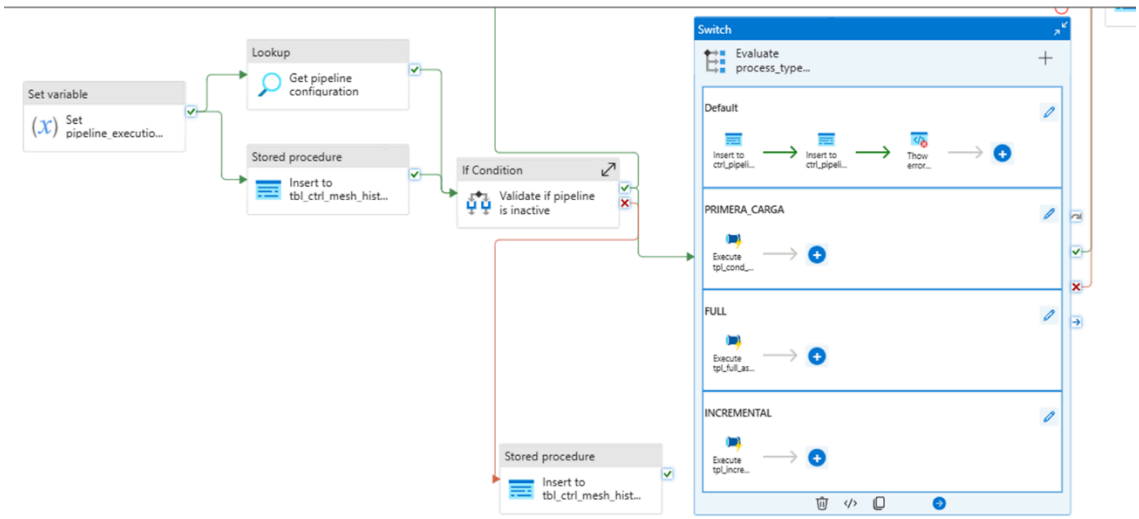


El **switch** dentro del pipeline principal evalúa el parámetro process_type y decide qué subplantilla ejecutar:

- **PRIMERA_CARGA** → Inicialización completa del dataset.
- **FULL** → Carga completa (sobrescribe).
- **INCREMENTAL** → Solo datos nuevos o modificados.

Esta lógica permite **flexibilidad y control**, evitando duplicación de pipelines para cada escenario.

Flujo visual del pipeline



El flujo incluye:

- **Lookup** → Obtiene configuración del pipeline.

- **Validaciones** → Activo/inactivo, límites de fecha.
- **Switch** → Determina el tipo de carga.
- **Stored Procedures** → Inserción en tablas de control para trazabilidad.

Malla Data Factory

Las **mallas** en ADF representan la capa de **orquestación** dentro de este ecosistema. Su función principal es **coordinar la ejecución de múltiples plantillas de ingesta** y, posteriormente, **disparar procesos de transformación en Databricks**.

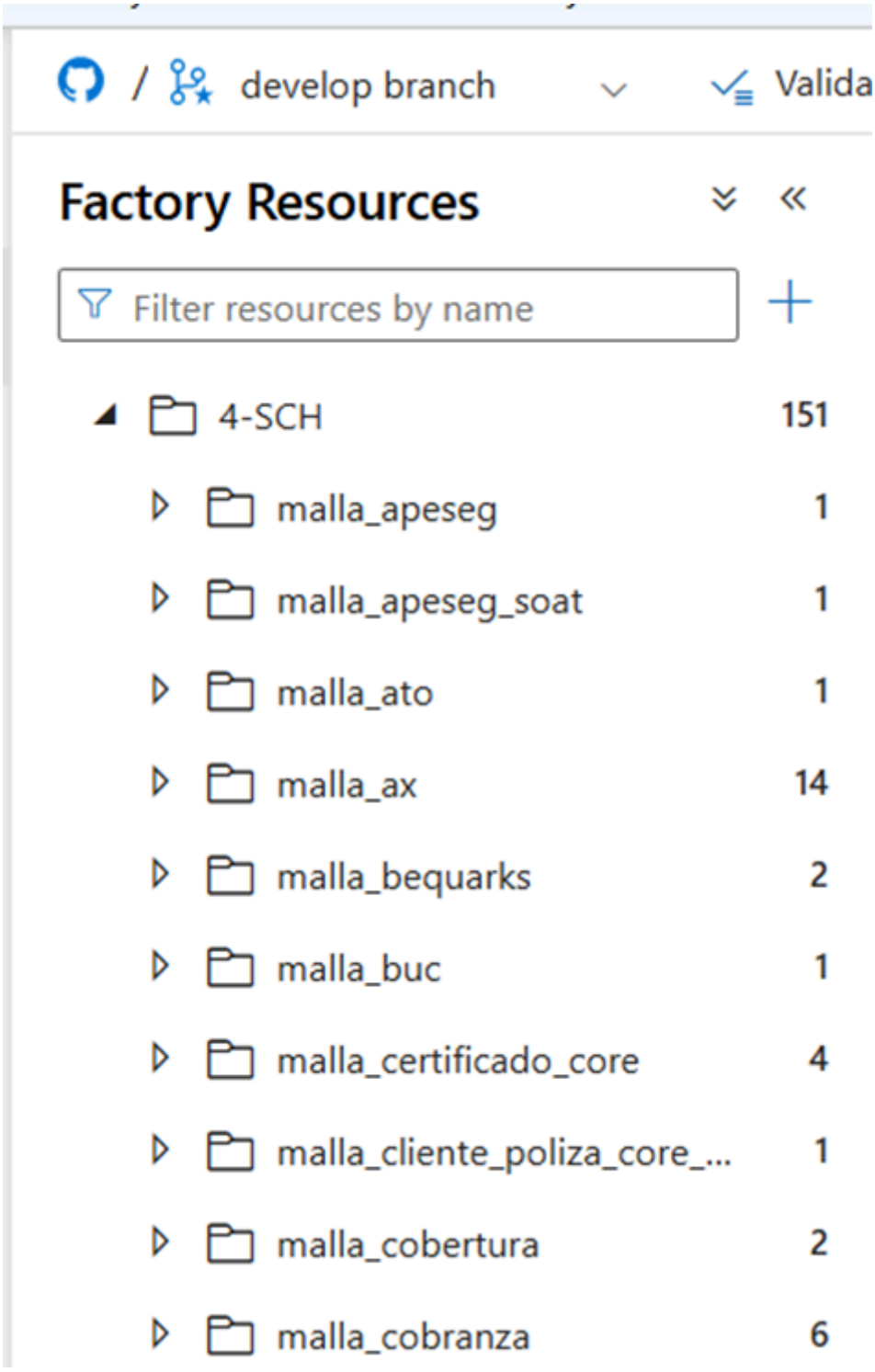
Contexto técnico

- Las mallas se encuentran en la carpeta **4-SCH** dentro de ADF.
- Cada malla agrupa pipelines que:
    - **Inicializan el control del flujo** (validaciones y registros).
    - **Invocan plantillas de ingesta** para traer datos desde diferentes fuentes.
    - **Ejecutan workflows en Databricks** para procesamiento avanzado.
    - **Finalizan el control**, asegurando trazabilidad y cierre del proceso.

Esta estructura permite **automatización completa del ciclo de ingesta y transformación**, reduciendo intervención manual y errores.

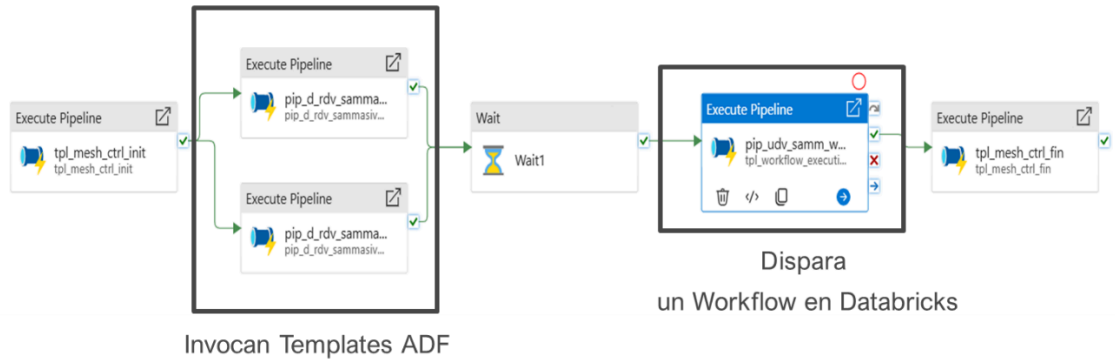
Vista de la carpeta de mallas

Las mallas están organizadas en la carpeta **4-SCH** y cada una corresponde a un **core o proceso específico del negocio**.



Detalle de mallas por core

Estas mallas son responsables de **agrupar pipelines relacionados** y definir el orden en que se ejecutan las plantillas y workflows.



Descripción:

- Cada carpeta representa una **malla** asociada a un dominio funcional.
- Dentro de cada malla se encuentran pipelines que:
  - **Inicializan el control** (validación y registro de ejecución).
  - **Invocan plantillas genéricas (TPL)** para ingestar datos.
  - **Disparan workflows en Databricks** para transformación avanzada.
  - **Finalizan el control**, asegurando trazabilidad.

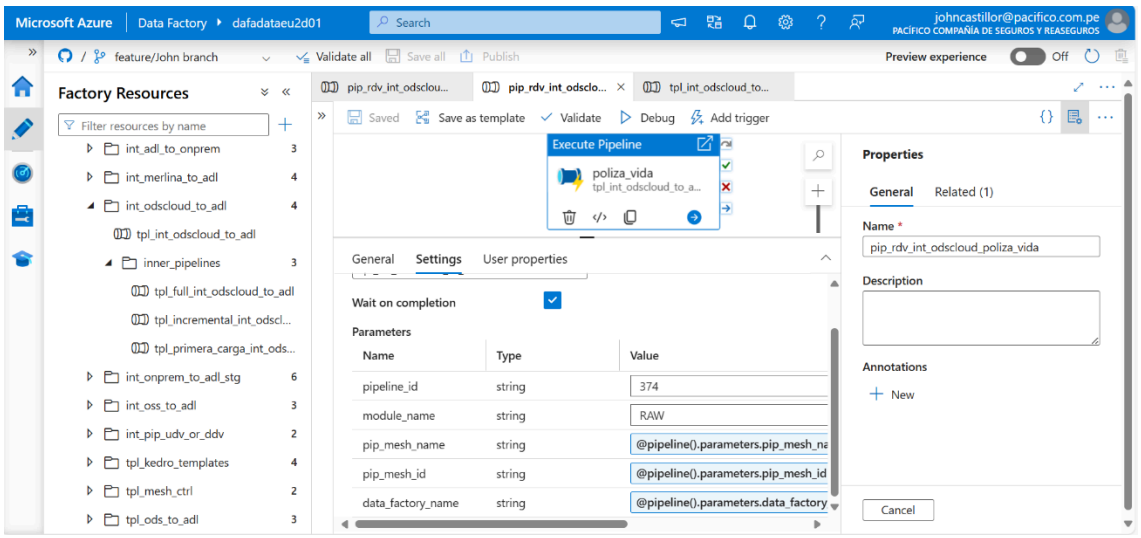
Configuración pipeline RDV

En esta sección vemos cómo se configura esta dentro de la capa **RDV (Raw Data Vault)**.
Este pipeline invoca una **plantilla genérica** y utiliza parámetros dinámicos para adaptarse a diferentes escenarios sin duplicar lógica.

Contexto técnico

- Las plantillas en ADF son **reutilizables** gracias a parámetros que permiten personalizar su ejecución.
- Ejemplo de parámetros:
  - pipeline_id → Identificador único del pipeline.
  - module_name → Nombre del módulo o core (por ejemplo, RAW).
  - Otros parámetros
    como pip_mesh_name, pip_mesh_id y data_factory_name se usan para trazabilidad y control (se les puede colocar cualquier valor o definirlos correctamente para garantizar lo mencionado).

Ejemplo visual



En la imagen se observa:

- Pipeline pip_rdv_int_odscloud_poliza_vida que invoca la plantilla tpl_int_odscloud_to_adl.
- Configuración en la pestaña **Settings**, donde se definen los parámetros mencionados.
- Uso de expresiones dinámicas (@pipeline().parameters...) para obtener valores desde el contexto de ejecución. Por ejemplo: **pipeline_id**, **module_name**, pip_mesh_name, etc (las dos primeras deben estar sí o sí, correctamente definidas)

Detalle importante:
Este pipeline pertenece a la malla que orquesta procesos relacionados con **polizas de vida**, y gracias a los parámetros, la misma plantilla puede ser usada para otros productos como **vehículos, movimientos, solicitudes**, etc.

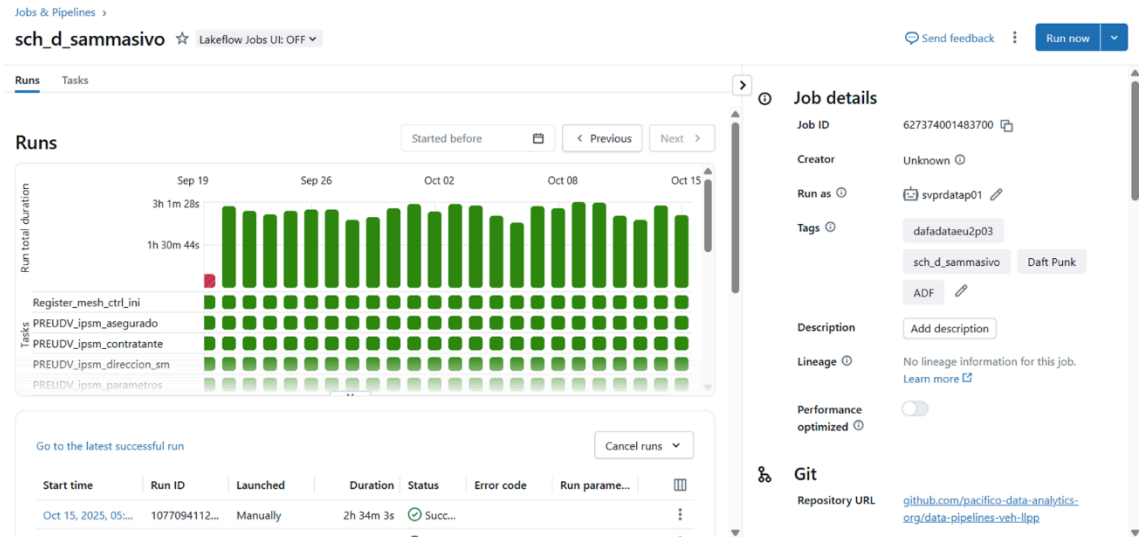
Workflow Databricks

Los **workflows en Databricks** son la parte del ecosistema donde se ejecuta la **lógica de transformación y procesamiento avanzado** sobre los datos ingeridos. Estos workflows son invocados desde las **mallas en ADF** y permiten integrar la potencia de **Spark** con la orquestación definida en Data Factory.

Contexto técnico

- **Importante:** Los pipelines en ADF que antes llamaban directamente a workflows están **deprecados**.
- Para las capas **UDV (Silver)** y **DDV (Gold)**, la mayoría de los procesos se ejecutan **directamente en Databricks**

Monitoreo y seguimiento



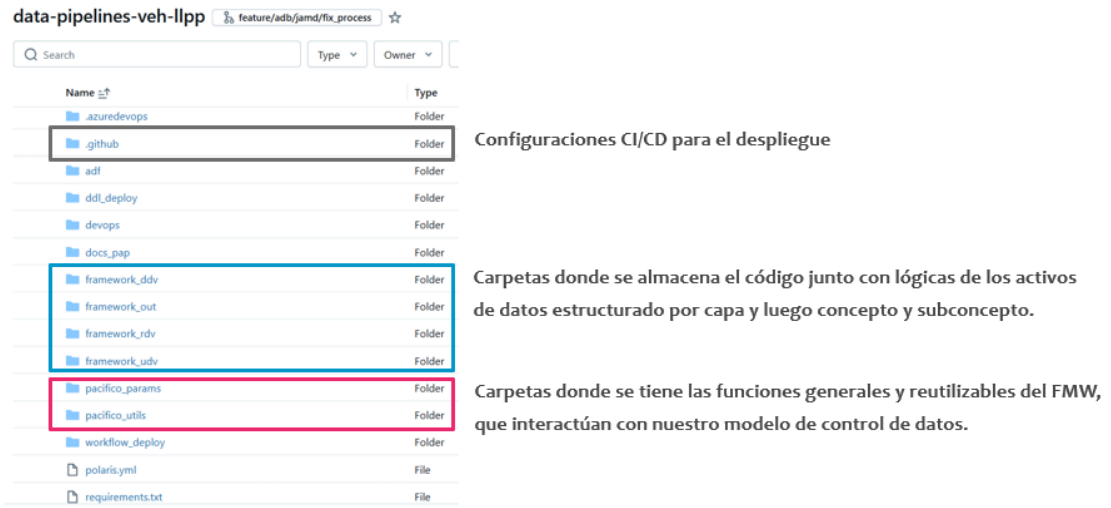
En la imagen se observa:

- **Historial de ejecuciones (Runs):** Cada barra verde indica una ejecución exitosa; las rojas, fallidas.
- **Duración por ejecución:** Permite identificar cuellos de botella.
- **Detalle del Job:** Incluye:
  - Job ID → Identificador único.
  - Tags → Contexto del flujo (ADF, malla asociada).
  - Git Repository URL → Código fuente vinculado para trazabilidad.
- **Última ejecución:** Fecha, duración y estado (ejemplo: Success).

Repositorios de trabajo

En nuestros repositorios mantenemos una estructura clara y organizada para garantizar **escalabilidad, reutilización y trazabilidad** del código.

Vista general



Descripción de las carpetas principales

1. **Configuraciones CI/CD para el despliegue**
   - Carpeta: .github

- Contiene pipelines y configuraciones para la **automatización del despliegue** (CI/CD), asegurando que los cambios en el código se integren y desplieguen de forma controlada.

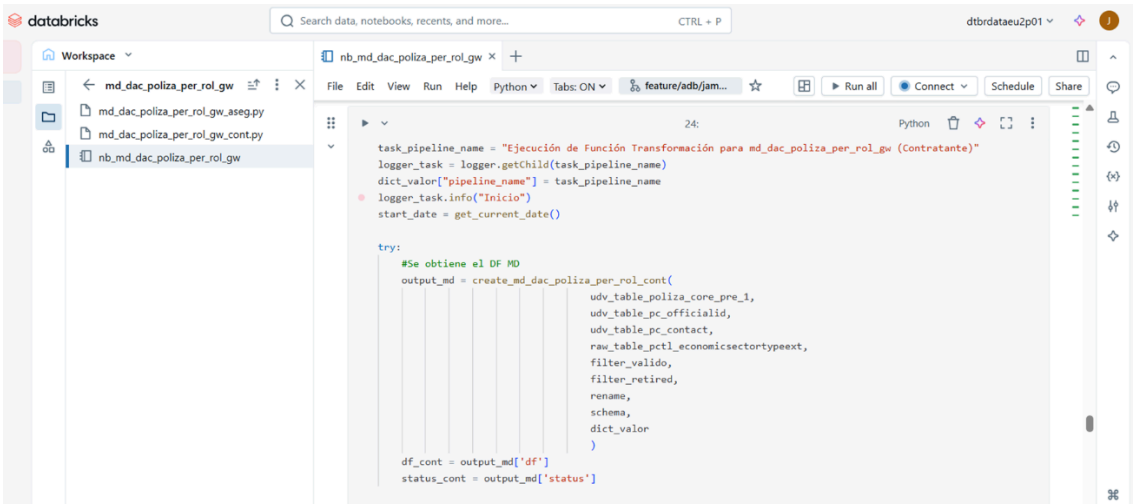
2. **Frameworks por capa**
   - Carpetas: framework_ddv, framework_out, framework_rdv, framework_udv
   - Aquí se almacena el código junto con las **lógicas de los activos de datos**, estructurado por capa (Landing, Raw, Silver, Gold) y luego por concepto y subconcepto.
   - Ejemplo:
     - framework_rdv → Lógica para la capa Raw Data Vault.
     - framework_udv → Lógica para la capa Unified Data Vault (Silver).
     - framework_ddv → Lógica para la capa Data Driven Vault (Gold).
3. **Funciones generales y parámetros**
   - Carpetas: pacifico_params, pacifico_utils
   - Contienen funciones generales y reutilizables del (FMW) que interactúan con el **modelo de control de datos**.
   - Ejemplo:
     - pacifico_params → Parámetros centralizados para pipelines y workflows.
     - pacifico_utils → Funciones genéricas para validaciones, transformaciones y utilidades del framework.

Notebooks de trabajo

En Databricks separamos **orquestación** de **lógica** para mantener el código limpio, testeable y reutilizable:

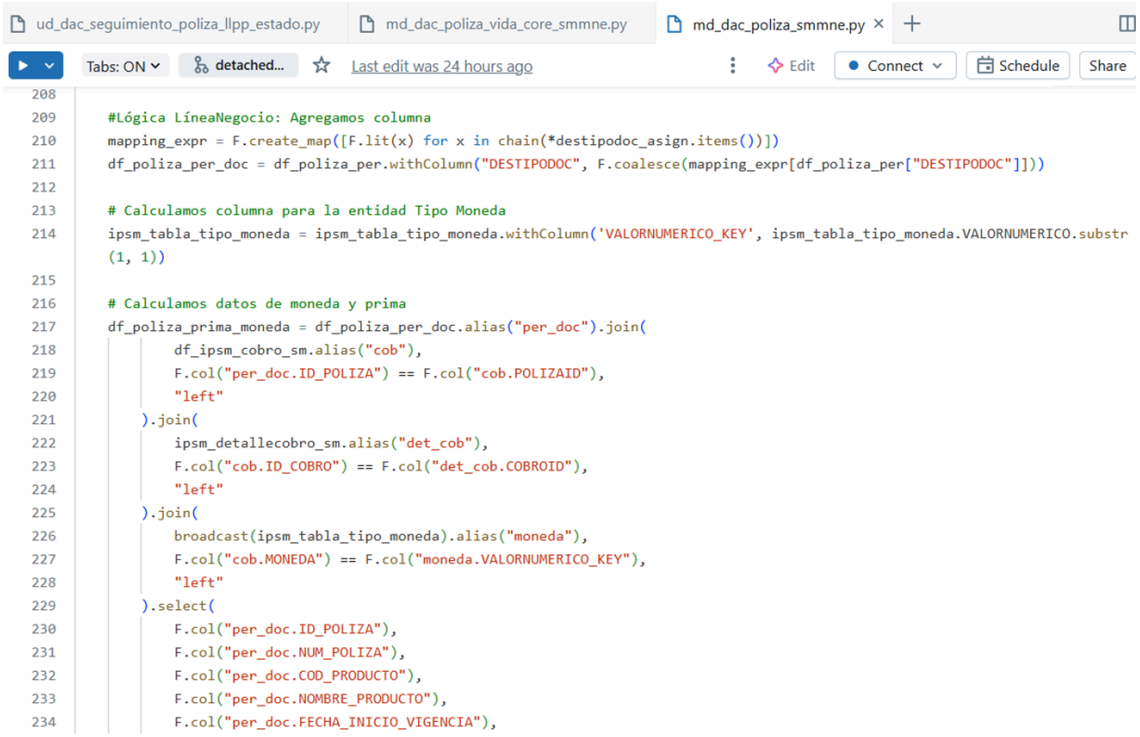
- **Notebooks (nb_*.py/.ipynb)** → Actúan como **tareas (tasks)** dentro de un workflow.
  Orquestan el paso a paso: leen parámetros, invocan funciones de negocio, manejan I/O y registran métricas/estados.
- **Scripts (*.py)** → Contienen la **lógica de negocio** (transformaciones PySpark puras).
  Están al **mismo nivel** que los notebooks para que estos los importen/ejecuten directamente.

Vista: notebooks y scripts al mismo nivel



En el panel de la izquierda verás notebooks nb_* y scripts *.py en la **misma carpeta**. El notebook orquesta; el .py implementa funciones reutilizables.

Los nb son como tasks en un workflow



Cada notebook se mapea a una **tarea del workflow** (Job de Databricks).
Recibe parámetros del Job, ejecuta la transformación y reporta estado/duración para observabilidad.

Relación nb ↔ .py (orquestación vs. lógica)

- **Notebook (nb):**
  - Recibe **parámetros de ejecución** (fecha de corte, core, flags de modo, etc.).
  - Llama funciones del **script .py** correspondiente.
  - Se encarga de **I/O** (lectura de fuentes, escritura en Silver/Gold), logging y control de errores.
- **Script (.py):**
  - Expone **funciones puras** (reciben DataFrames/parametros y devuelven DataFrames/artefactos).
  - No debe tener side-effects de I/O (para facilitar tests y reutilización).
  - Puede agrupar reglas por **dominio/entidad** (ej.: póliza, cliente, cobranza).