MÓDULO 01: PYTHON MODERNO PARA MLOps

El Puente de Scripter a Ingeniero de Software

Guía MLOps v5.0: Senior Edition | DuqueOM | Noviembre 2025

MÓDULO 01: Python Moderno para MLOps

De Scripts que "Funcionan" a Código que Pasa Code Review

"La diferencia entre un Junior y un Senior no es la complejidad del algoritmo, es la calidad del código que lo rodea."

Duración	Teoría	Práctica
6-8 horas	30%	70%

Por Qué Este Módulo Existe

El Problema

Lo Que Lograrás en Este Módulo

Al terminar, tu código: 1. **Pasará mypy** sin errores (tipado estático) 2. **Validará inputs** automáticamente (Pydantic) 3. **Será instalable** como paquete (`pip instal -e .`) 4. **Seguirá SOLID** (especialmente Single Responsibility y Dependency Injection) 5. **Tendrá estructura profesional** (src layout)

1.1 Type Hints: Tu Contrato con el Futuro

¿Por Qué Tipar?

```
        # × ANTES: ¿Qué recibe? ¿Qué retorna?
        def process_data(data, config):
            # ... 200 líneas después ...
            return result

        #  DESPUÉS: Contrato claro, IDE autocompleta, errores detectados ANTES de ejecutar
        def process_data(data: pd.DataFrame, config: TrainingConfig) -> ProcessedData:
            # ... 200 líneas después ...
            return result
```

Los Tipos Esenciales para ML

```python
from typing import (
    List, Dict, Tuple, Optional, Union,
    Any, Callable, TypeVar, Generic,
    Literal, TypedDict
)
from pathlib import Path
import pandas as pd
import numpy as np
from numpy.typing import NDArray

# ═══════════════════════════════════════════════════════════
# TIPOS BÁSICOS
# ═══════════════════════════════════════════════════════════

# Primitivos
name: str = "BankChurn"
n_estimators: int = 100
learning_rate: float = 0.01
is_trained: bool = False

# Colecciones
features: List[str] = ["age", "balance", "tenure"]
params: Dict[str, Any] = {"n_estimators": 100, "max_depth": 5}
shape: Tuple[int, int] = (1000, 10)

# Optional = puede ser None
model_path: Optional[Path] = None  # Path | None en Python 3.10+

# Union = múltiples tipos posibles
target: Union[str, List[str]] = "Exited"  # str | List[str] en 3.10+

# ═══════════════════════════════════════════════════════════
# TIPOS PARA ML
# ═══════════════════════════════════════════════════════════

# DataFrames (pandas-stubs requerido para mypy)
def load_data(path: Path) -> pd.DataFrame:
    return pd.read_csv(path)

# Arrays NumPy
def compute_predictions(X: NDArray[np.float64]) -> NDArray[np.float64]:
    return model.predict_proba(X)[:, 1]

# Literal = valores específicos permitidos
ModelType = Literal["random_forest", "logistic", "xgboost"]

def train_model(model_type: ModelType) -> BaseEstimator:
    if model_type == "random_forest":
        return RandomForestClassifier()
    # mypy SABE que model_type solo puede ser estos 3 valores

# ═══════════════════════════════════════════════════════════
# TIPOS AVANZADOS
# ═══════════════════════════════════════════════════════════

# TypedDict = diccionarios con estructura conocida
class MetricsDict(TypedDict):
    auc_roc: float
    precision: float
    recall: float
    f1: float

def evaluate(y_true: NDArray, y_pred: NDArray) -> MetricsDict:
    return {
        "auc_roc": roc_auc_score(y_true, y_pred),
        "precision": precision_score(y_true, y_pred),
        "recall": recall_score(y_true, y_pred),
        "f1": f1_score(y_true, y_pred),
    }

# Generic + TypeVar = funciones que preservan tipos
T = TypeVar("T", bound=BaseEstimator)

def clone_and_fit(model: T, X: NDArray, y: NDArray) -> T:
    """Clona el modelo, lo entrena, y retorna el mismo tipo."""
    cloned = clone(model)
    cloned.fit(X, y)
    return cloned  # mypy sabe que retorna el mismo tipo que entró
```

Configurar mypy para ML

```toml
# pyproject.toml
[tool.mypy]
python_version = "3.11"
warn_return_any = true
warn_unused_ignores = true
disallow_untyped_defs = true
ignore_missing_imports = true  # Para librerías sin stubs (sklearn, etc.)

[[tool.mypy.overrides]]
module = [
    "sklearn.*",
    "pandas.*",
    "numpy.*",
    "mlflow.*",
]
ignore_missing_imports = true
```

Ejercicio 1.1: Tipar una Función Real

```python
# ✗ Código sin tipos (típico de notebooks)
def prepare_features(df, num_cols, cat_cols, target):
    X = df.drop(columns=[target])
    y = df[target]

    preprocessor = ColumnTransformer([
        ('num', StandardScaler(), num_cols),
        ('cat', OneHotEncoder(), cat_cols)
    ])

    X_transformed = preprocessor.fit_transform(X)
    return X_transformed, y, preprocessor

#  TU TAREA: Añade type hints completos
# Pista: usa NDArray, pd.DataFrame, List[str], ColumnTransformer, etc.
```

▶ Ver Solución

1.2 Pydantic: Validación de Datos que No Perdona

El Problema de Configs sin Validar

```
# × ANTES: Diccionario sin validación
config = {
    "n_estimators": "100",  # Oops, es string
    "max_depth": -5,        # Valor inválido
    "random_state": None,   # ¿Es intencional?
    # "learning_rate" falta
}

model = RandomForestClassifier(**config)  # BOOM en runtime
```

Pydantic al Rescate

```
from pydantic import BaseModel, Field, field_validator, model_validator
from typing import Literal, Optional
from pathlib import Path

class ModelConfig(BaseModel):
    """Configuración validada para modelos de clasificación."""

    # Campos con valores por defecto y restricciones
    n_estimators: int = Field(
        default=100,
        ge=10,      # >= 10
        le=1000,    # <= 1000
        description="Número de árboles en el ensemble"
    )

    max_depth: Optional[int] = Field(
        default=None,
        ge=1,
        le=50,
        description="Profundidad máxima. None = sin límite"
    )

    learning_rate: float = Field(
        default=0.1,
        gt=0,       # > 0
        le=1,       # <= 1
    )

    model_type: Literal["random_forest", "xgboost", "lightgbm"] = "random_forest"

    random_state: int = Field(default=42, description="Seed para reproducibilidad")

    # Validador personalizado
    @field_validator("n_estimators")
    @classmethod
    def validate_n_estimators(cls, v: int) -> int:
        if v < 10:
            raise ValueError("n_estimators debe ser >= 10 para resultados estables")
        return v

    # Validador que usa múltiples campos
    @model_validator(mode="after")
    def validate_model_params(self) -> "ModelConfig":
        if self.model_type == "xgboost" and self.learning_rate > 0.3:
            raise ValueError("XGBoost funciona mejor con learning_rate <= 0.3")
        return self


class DataConfig(BaseModel):
    """Configuración para rutas de datos."""

    raw_data_path: Path
    processed_data_path: Path
    model_output_path: Path = Path("models/")

    target_column: str = "Exited"
    test_size: float = Field(default=0.2, ge=0.1, le=0.5)

    @field_validator("raw_data_path")
    @classmethod
    def validate_raw_path_exists(cls, v: Path) -> Path:
        if not v.exists():
            raise ValueError(f"El archivo de datos no existe: {v}")
        return v


class TrainingConfig(BaseModel):
    """Configuración completa de entrenamiento."""

    model: ModelConfig = Field(default_factory=ModelConfig)
    data: DataConfig

    experiment_name: str = "bankchurn"
    run_name: Optional[str] = None

    class Config:
        # Permite crear desde archivo YAML/JSON
        extra = "forbid"  # Error si hay campos desconocidos
```

Uso en la Práctica

```python
# ═══════════════════════════════════════════════════════════
# CARGAR DESDE YAML
# ═══════════════════════════════════════════════════════════
import yaml

# configs/config.yaml
"""
model:
  n_estimators: 200
  max_depth: 10
  model_type: random_forest

data:
  raw_data_path: data/raw/churn.csv
  processed_data_path: data/processed/
  target_column: Exited
  test_size: 0.2

experiment_name: bankchurn-v2
"""

def load_config(path: Path) -> TrainingConfig:
    """Carga y valida configuración desde YAML."""
    with open(path) as f:
        raw_config = yaml.safe_load(f)

    # Pydantic valida automáticamente
    return TrainingConfig(**raw_config)

# Si hay CUALQUIER error de validación, Pydantic lo detecta
config = load_config(Path("configs/config.yaml"))

# ═══════════════════════════════════════════════════════════
# INTEGRACIÓN CON CLI (typer + pydantic)
# ═══════════════════════════════════════════════════════════
import typer
from typing import Annotated

app = typer.Typer()

@app.command()
def train(
    config_path: Annotated[Path, typer.Argument(help="Ruta al config YAML")],
    override_n_estimators: Annotated[Optional[int], typer.Option("--n-estimators")] = None,
):
    """Entrena el modelo con configuración validada."""
    config = load_config(config_path)

    # Override desde CLI si se proporciona
    if override_n_estimators:
        config.model.n_estimators = override_n_estimators

    trainer = ChurnTrainer(config)
    trainer.run()
```

Pydantic para Validar Requests de API

```python
from pydantic import BaseModel, Field
from typing import Optional
from fastapi import FastAPI

class PredictionRequest(BaseModel):
    """Request para predicción de churn."""

    credit_score: int = Field(..., ge=300, le=850, description="Score crediticio")
    age: int = Field(..., ge=18, le=100)
    tenure: int = Field(..., ge=0, le=50, description="Años como cliente")
    balance: float = Field(..., ge=0)
    num_of_products: int = Field(..., ge=1, le=4)
    has_cr_card: bool
    is_active_member: bool
    estimated_salary: float = Field(..., ge=0)
    geography: Literal["France", "Germany", "Spain"]
    gender: Literal["Male", "Female"]

    class Config:
        json_schema_extra = {
            "example": {
                "credit_score": 650,
                "age": 35,
                "tenure": 5,
                "balance": 50000.0,
                "num_of_products": 2,
                "has_cr_card": True,
                "is_active_member": True,
                "estimated_salary": 75000.0,
                "geography": "France",
                "gender": "Female"
            }
        }


class PredictionResponse(BaseModel):
    """Response de predicción."""

    churn_probability: float = Field(..., ge=0, le=1)
    prediction: Literal["churn", "no_churn"]
    confidence: float = Field(..., ge=0, le=1)
    model_version: str


app = FastAPI(title="BankChurn API")

@app.post("/predict", response_model=PredictionResponse)
async def predict(request: PredictionRequest) -> PredictionResponse:
    # FastAPI + Pydantic validan automáticamente el input
    # Si el JSON no cumple el schema, retorna 422 con detalles del error
    ...
```

1.3 Estructura de Paquetes: El `src/` Layout

El Problema del “Script Spaghetti”

```
× ESTRUCTURA TÍPICA DE DATA SCIENTIST:

project/
├── train_model_v2_final.py
├── train_model_v2_final_REAL.py
├── utils.py
├── utils_new.py
├── notebook_exploration.ipynb
├── notebook_final.ipynb
├── data/
└── requirements.txt

PROBLEMAS:
- ¿Cuál es el archivo correcto?
- ¿Cómo importo `utils.py` desde otro directorio?
- No es instalable con pip
- Tests no encuentran los módulos
```

La Estructura Profesional: `src/` Layout

```
 ESTRUCTURA PROFESIONAL:

bankchurn-predictor/
├── src/
│   └── bankchurn/                #  El paquete instalable
│       ├── __init__.py           # Expone API pública
│       ├── config.py             # Configuración Pydantic
│       ├── data/
│       │   ├── __init__.py
│       │   ├── loader.py         # Carga de datos
│       │   └── preprocessing.py  # Transformaciones
│       ├── models/
│       │   ├── __init__.py
│       │   ├── trainer.py        # Clase ChurnTrainer
│       │   └── inference.py      # Predicción
│       └── utils/
│           ├── __init__.py
│           ├── logger.py
│           └── metrics.py
│
├── app/
│   └── fastapi_app.py            #  API (no es parte del paquete)
│
├── tests/
│   ├── conftest.py               # Fixtures de pytest
│   ├── unit/
│   │   ├── test_config.py
│   │   └── test_preprocessing.py
│   └── integration/
│       └── test_pipeline.py
│
├── notebooks/                    #  Solo para exploración
│   └── 01_eda.ipynb
│
├── configs/
│   └── config.yaml
│
├── data/
│   ├── raw/                      # DVC-tracked
│   └── processed/
│
├── pyproject.toml                #  Configuración moderna
├── Makefile
└── README.md
```

`pyproject.toml` Moderno (reemplaza setup.py)

```toml
[build-system]
requires = ["setuptools>=61.0", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "bankchurn"
version = "0.1.0"
description = "Predictor de churn bancario con MLOps completo"
readme = "README.md"
license = {text = "MIT"}
authors = [
    {name = "Tu Nombre", email = "tu@email.com"}
]
requires-python = ">=3.10"

dependencies = [
    "pandas>=2.0.0",
    "scikit-learn>=1.3.0",
    "pydantic>=2.0.0",
    "pyyaml>=6.0",
    "mlflow>=2.8.0",
    "fastapi>=0.104.0",
    "uvicorn>=0.24.0",
]

[project.optional-dependencies]
dev = [
    "pytest>=7.4.0",
    "pytest-cov>=4.1.0",
    "mypy>=1.6.0",
    "ruff>=0.1.0",
    "pre-commit>=3.5.0",
]
docs = [
    "mkdocs>=1.5.0",
    "mkdocs-material>=9.4.0",
]

[project.scripts]
# CLI commands
bankchurn-train = "bankchurn.cli:train"
bankchurn-predict = "bankchurn.cli:predict"

[tool.setuptools.packages.find]
where = ["src"]

[tool.ruff]
line-length = 100
select = ["E", "F", "I", "W", "B", "C4", "UP"]
ignore = ["E501"]  # Line too long (handled by formatter)

[tool.pytest.ini_options]
testpaths = ["tests"]
addopts = "-v --cov=src/bankchurn --cov-report=term-missing"

[tool.mypy]
python_version = "3.11"
warn_return_any = true
disallow_untyped_defs = true
ignore_missing_imports = true
```

`__init__.py` que Expone API Pública

```python
# src/bankchurn/__init__.py
"""
BankChurn Predictor
==================

Un sistema MLOps completo para predicción de churn bancario.

Uso básico:
    from bankchurn import ChurnTrainer, TrainingConfig

    config = TrainingConfig.from_yaml("configs/config.yaml")
    trainer = ChurnTrainer(config)
    trainer.run()
"""

from bankchurn.config import TrainingConfig, ModelConfig, DataConfig
from bankchurn.models.trainer import ChurnTrainer
from bankchurn.models.inference import ChurnPredictor

__version__ = "0.1.0"
__all__ = [
    "TrainingConfig",
    "ModelConfig",
    "DataConfig",
    "ChurnTrainer",
    "ChurnPredictor",
    "__version__",
]
```

Instalación en Modo Editable

```bash
# Desde la raíz del proyecto
pip install -e ".[dev]"

# Ahora puedes importar desde cualquier lugar:
from bankchurn import ChurnTrainer, TrainingConfig

# Y ejecutar CLI:
bankchurn-train configs/config.yaml
```

1.4 OOP para ML: Patrones que Funcionan

El Anti-Patrón: Funciones Sueltas

```python
# ✗ ANTI-PATRÓN: Todo son funciones sueltas que dependen de globals
RANDOM_STATE = 42
MODEL_PATH = "models/model.pkl"

def load_data():
    global data  # 😱
    data = pd.read_csv("data/raw/churn.csv")

def preprocess():
    global X, y  # 😱😱
    X = data.drop("Exited", axis=1)
    y = data["Exited"]

def train():
    global model  # 😱😱😱
    model = RandomForestClassifier(random_state=RANDOM_STATE)
    model.fit(X, y)
    joblib.dump(model, MODEL_PATH)

# PROBLEMAS:
# - Estado global = bugs difíciles de encontrar
# - Imposible de testear unitariamente
# - No puedes tener 2 configuraciones simultáneas
```

El Patrón Correcto: Trainer Class

```python
#  PATRÓN PROFESIONAL: Clases con Dependency Injection
from dataclasses import dataclass, field
from pathlib import Path
from typing import Optional, Protocol
import pandas as pd
import numpy as np
from sklearn.base import BaseEstimator, clone
from sklearn.model_selection import train_test_split
import joblib
import mlflow

from bankchurn.config import TrainingConfig
from bankchurn.data.preprocessing import FeatureEngineer


class DataLoader(Protocol):
    """Protocolo para cargadores de datos (Dependency Injection)."""
    def load(self) -> pd.DataFrame: ...


class CSVDataLoader:
    """Implementación concreta para CSV."""
    def __init__(self, path: Path):
        self.path = path

    def load(self) -> pd.DataFrame:
        return pd.read_csv(self.path)


@dataclass
class ChurnTrainer:
    """
    Trainer para modelos de churn prediction.

    Sigue el principio de Single Responsibility:
    - Orquesta el entrenamiento
    - Delega preprocesamiento a FeatureEngineer
    - Delega tracking a MLflow

    Ejemplo:
        config = TrainingConfig.from_yaml("configs/config.yaml")
        trainer = ChurnTrainer(config)
        metrics = trainer.run()
    """

    config: TrainingConfig
    data_loader: Optional[DataLoader] = None
    _model: Optional[BaseEstimator] = field(default=None, init=False)
    _feature_engineer: Optional[FeatureEngineer] = field(default=None, init=False)

    def __post_init__(self):
        """Inicializa dependencias si no se inyectaron."""
        if self.data_loader is None:
            self.data_loader = CSVDataLoader(self.config.data.raw_data_path)

        self._feature_engineer = FeatureEngineer(
            num_features=self.config.data.numerical_features,
            cat_features=self.config.data.categorical_features,
        )

    def load_data(self) -> pd.DataFrame:
        """Carga datos usando el loader inyectado."""
        return self.data_loader.load()

    def prepare_data(
        self,
        df: pd.DataFrame
    ) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
        """Prepara datos para entrenamiento."""
        X = df.drop(columns=[self.config.data.target_column])
        y = df[self.config.data.target_column].values

        return train_test_split(
            X, y,
            test_size=self.config.data.test_size,
            random_state=self.config.model.random_state,
            stratify=y
        )

    def train(
        self,
        X_train: np.ndarray,
        y_train: np.ndarray
    ) -> BaseEstimator:
        """Entrena el pipeline completo."""
        from sklearn.pipeline import Pipeline

        # Pipeline = FeatureEngineer + Modelo
        pipeline = Pipeline([
            ('features', self._feature_engineer.get_transformer()),
            ('model', self._build_model())
        ])

        pipeline.fit(X_train, y_train)
        self._model = pipeline
        return pipeline
```

```
    def _build_model(self) -> BaseEstimator:
        """Construye el modelo según configuración."""
        if self.config.model.model_type == "random_forest":
            from sklearn.ensemble import RandomForestClassifier
            return RandomForestClassifier(
                n_estimators=self.config.model.n_estimators,
                max_depth=self.config.model.max_depth,
                random_state=self.config.model.random_state,
                class_weight='balanced',
                n_jobs=-1,
            )
        # ... otros modelos
        raise ValueError(f"Modelo no soportado: {self.config.model.model_type}")

    def evaluate(
        self,
        X_test: np.ndarray,
        y_test: np.ndarray
    ) -> dict[str, float]:
        """Evalúa el modelo y retorna métricas."""
        from sklearn.metrics import roc_auc_score, precision_score, recall_score, f1_score

        y_pred = self._model.predict(X_test)
        y_proba = self._model.predict_proba(X_test)[:, 1]

        return {
            "auc_roc": roc_auc_score(y_test, y_proba),
            "precision": precision_score(y_test, y_pred),
            "recall": recall_score(y_test, y_pred),
            "f1": f1_score(y_test, y_pred),
        }

    def save_model(self, path: Optional[Path] = None) -> Path:
        """Guarda el modelo entrenado."""
        path = path or self.config.data.model_output_path / "pipeline.pkl"
        path.parent.mkdir(parents=True, exist_ok=True)
        joblib.dump(self._model, path)
        return path

    def run(self) -> dict[str, float]:
        """
        Ejecuta el pipeline completo de entrenamiento.

        Returns:
            Diccionario con métricas de evaluación.
        """
        # MLflow tracking
        mlflow.set_experiment(self.config.experiment_name)

        with mlflow.start_run(run_name=self.config.run_name):
            # Log config
            mlflow.log_params(self.config.model.model_dump())

            # Pipeline
            df = self.load_data()
            X_train, X_test, y_train, y_test = self.prepare_data(df)
            self.train(X_train, y_train)
            metrics = self.evaluate(X_test, y_test)

            # Log metrics
            mlflow.log_metrics(metrics)

            # Save artifacts
            model_path = self.save_model()
            mlflow.log_artifact(str(model_path))

            return metrics
```

Uso del Trainer

```
# ═══════════════════════════════════════════════════════════════
# USO BÁSICO
# ═══════════════════════════════════════════════════════════════
from bankchurn import ChurnTrainer, TrainingConfig

config = TrainingConfig.from_yaml("configs/config.yaml")
trainer = ChurnTrainer(config)
metrics = trainer.run()
print(f"AUC-ROC: {metrics['auc_roc']:.4f}")

# ═══════════════════════════════════════════════════════════════
# CON DEPENDENCY INJECTION (para tests)
# ═══════════════════════════════════════════════════════════════
class MockDataLoader:
    """Mock para tests unitarios."""
    def load(self) -> pd.DataFrame:
        return pd.DataFrame({
            "feature1": [1, 2, 3, 4, 5],
            "feature2": ["a", "b", "a", "b", "a"],
            "Exited": [0, 1, 0, 1, 0]
        })

# En test
def test_trainer_with_mock_data():
    config = TrainingConfig(...)
    mock_loader = MockDataLoader()

    trainer = ChurnTrainer(config, data_loader=mock_loader)
    metrics = trainer.run()

    assert "auc_roc" in metrics
    assert 0 <= metrics["auc_roc"] <= 1
```

1.5 Decoradores Útiles para ML

Decoradores Esenciales

```python
from functools import wraps
from time import perf_counter
from typing import Callable, TypeVar, ParamSpec
import logging

P = ParamSpec("P")
R = TypeVar("R")

# ═══════════════════════════════════════════════════════════════
# TIMING: Medir tiempo de ejecución
# ═══════════════════════════════════════════════════════════════
def timer(func: Callable[P, R]) -> Callable[P, R]:
    """Loguea el tiempo de ejecución de una función."""
    @wraps(func)
    def wrapper(*args: P.args, **kwargs: P.kwargs) -> R:
        start = perf_counter()
        result = func(*args, **kwargs)
        elapsed = perf_counter() - start
        logging.info(f"{func.__name__} took {elapsed:.2f}s")
        return result
    return wrapper


# ═══════════════════════════════════════════════════════════════
# RETRY: Reintentar en caso de fallo
# ═══════════════════════════════════════════════════════════════
def retry(max_attempts: int = 3, delay: float = 1.0):
    """Reintenta una función N veces antes de fallar."""
    def decorator(func: Callable[P, R]) -> Callable[P, R]:
        @wraps(func)
        def wrapper(*args: P.args, **kwargs: P.kwargs) -> R:
            last_exception = None
            for attempt in range(max_attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    last_exception = e
                    logging.warning(
                        f"{func.__name__} failed (attempt {attempt + 1}/{max_attempts}): {e}"
                    )
                    if attempt < max_attempts - 1:
                        import time
                        time.sleep(delay)
            raise last_exception
        return wrapper
    return decorator


# ═══════════════════════════════════════════════════════════════
# CACHE DE DATOS (útil para cargas pesadas)
# ═══════════════════════════════════════════════════════════════
from functools import lru_cache
import hashlib

def cache_dataframe(func: Callable[P, pd.DataFrame]) -> Callable[P, pd.DataFrame]:
    """
    Cachea DataFrames basado en los argumentos.
    Útil para evitar recargar datos en desarrollo.
    """
    cache: dict[str, pd.DataFrame] = {}

    @wraps(func)
    def wrapper(*args: P.args, **kwargs: P.kwargs) -> pd.DataFrame:
        # Crear key única basada en argumentos
        key = hashlib.md5(
            str((args, sorted(kwargs.items()))).encode()
        ).hexdigest()

        if key not in cache:
            cache[key] = func(*args, **kwargs)
            logging.info(f"Cached result for {func.__name__}")
        else:
            logging.info(f"Using cached result for {func.__name__}")

        return cache[key].copy()  # Retornar copia para evitar mutaciones

    wrapper.clear_cache = lambda: cache.clear()  # type: ignore
    return wrapper


# ═══════════════════════════════════════════════════════════════
# USO
# ═══════════════════════════════════════════════════════════════
@timer
@cache_dataframe
def load_training_data(path: Path) -> pd.DataFrame:
    """Carga datos de entrenamiento (cacheado)."""
    return pd.read_csv(path)

@retry(max_attempts=3, delay=2.0)
def download_from_s3(bucket: str, key: str) -> bytes:
    """Descarga archivo de S3 con reintentos."""
    import boto3
    s3 = boto3.client('s3')
    response = s3.get_object(Bucket=bucket, Key=key)
    return response['Body'].read()
```

1.6 Ejercicio Integrador: Refactorizar un Script

Código Original (Típico de Notebook)

```python
# ✗ train_model.py - Código típico de Data Scientist
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.metrics import roc_auc_score
import joblib

# Configuración hardcodeada
DATA_PATH = "/home/usuario/data/churn.csv"
MODEL_PATH = "model.pkl"
TEST_SIZE = 0.2
N_ESTIMATORS = 100
RANDOM_STATE = 42

# Cargar datos
data = pd.read_csv(DATA_PATH)

# Features
num_features = ['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'EstimatedSalary']
cat_features = ['Geography', 'Gender']

X = data.drop('Exited', axis=1)
y = data['Exited']

# Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=TEST_SIZE, random_state=RANDOM_STATE)

# Preprocessing
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), num_features),
    ('cat', OneHotEncoder(), cat_features)
])

X_train_processed = preprocessor.fit_transform(X_train)
X_test_processed = preprocessor.transform(X_test)

# Train
model = RandomForestClassifier(n_estimators=N_ESTIMATORS, random_state=RANDOM_STATE)
model.fit(X_train_processed, y_train)

# Evaluate
y_pred_proba = model.predict_proba(X_test_processed)[:, 1]
auc = roc_auc_score(y_test, y_pred_proba)
print(f"AUC: {auc}")

# Save
joblib.dump(model, MODEL_PATH)
joblib.dump(preprocessor, "preprocessor.pkl")
```

Tu Tarea: Refactorizarlo

Convierte el script anterior en una estructura profesional con:

1. `src/bankchurn/config.py`: Configuración Pydantic
2. `src/bankchurn/data/preprocessing.py`: Clase FeatureEngineer
3. `src/bankchurn/models/trainer.py`: Clase ChurnTrainer
4. `pyproject.toml`: Configuración del paquete
5. **Tests unitarios** para cada componente

▶ Ver Estructura de Solución

1.7 Autoevaluación

Antes de pasar al siguiente módulo, verifica:

Checklist de Competencias

```
TYPE HINTS:
[ ] Puedo tipar funciones con tipos básicos (str, int, List, Dict)
[ ] Sé usar Optional, Union, Literal
[ ] Entiendo TypedDict y TypeVar
[ ] mypy pasa sin errores en mi código

PYDANTIC:
[ ] Puedo crear modelos Pydantic con validaciones
[ ] Sé usar Field con restricciones (ge, le, gt, etc.)
[ ] Puedo crear validadores personalizados (@field_validator)
[ ] Sé cargar configuración desde YAML

ESTRUCTURA:
[ ] Entiendo el src/ layout
[ ] Puedo crear un pyproject.toml funcional
[ ] Sé instalar mi paquete en modo editable
[ ] Entiendo qué exportar en __init__.py

OOP:
[ ] Puedo crear una clase Trainer con __post_init__
[ ] Entiendo Dependency Injection
[ ] Sé usar dataclasses con field()
[ ] Puedo escribir código testeable (sin globals)
```

Preguntas de Reflexión

1. ¿Por qué es importante tipar el código en proyectos MLOps?
2. ¿Cuál es la ventaja de Pydantic sobre validar con if/else?

3. ¿Por qué el `src/` layout es mejor que poner todo en la raíz?
4. ¿Cómo facilita Dependency Injection el testing?

Siguiente Paso

Con el conocimiento de Python moderno, estás listo para **diseñar sistemas ML** a nivel arquitectónico.

Ir a Módulo 02: Diseño de Sistemas ML →

Módulo 01 completado. Tu código ahora pasa code review de Senior.