Perfecto. Te armaré una propuesta integral de proyecto con un enfoque en arquitectura moderna de datos y MLOps, que aproveche tu dataset de e-commerce pero lo adapte inteligentemente a un contexto financiero/fintech, incluyendo datos sintéticos si es necesario. El proyecto cubrirá desde ingesta hasta despliegue, aplicando conceptos como Lambda Architecture, Spark optimizations, CI/CD, y feature store con AWS y Kafka.

Te avisaré cuando esté lista la propuesta detallada con estructura de carpetas, herramientas, objetivos y cómo documentarla para que se vea profesional en tu GitHub.

Diseño de un Proyecto de Portafolio para ML Engineer en FinTech

Introducción

En este proyecto de portafolio se construirá una **solución de machine learning** simulada para el sector **fintech**, abarcando de extremo a extremo las competencias clave de un **Machine Learning Engineer**. Se implementará una **arquitectura Lambda** para combinar procesamiento batch (histórico) y streaming (tiempo real), evitando problemas como *data leakage* en los pipelines de entrenamiento. Además, se incorporarán prácticas de **Big Data** con Apache Spark (incluyendo *broadcast joins* y optimizaciones de Spark SQL), flujos de **datos en streaming** con Kafka/Kinesis, un paradigma de **feature store** (offline/online), y un pipeline de **MLOps** con CI/CD (GitHub Actions + Docker) y despliegue del modelo (AWS SageMaker o FastAPI en AWS Lambda). Finalmente, se orquestará todo con Airflow o AWS Glue, simulando un entorno productivo con monitoreo básico y proporcionando documentación clara. A continuación, se detallan los componentes técnicos y decisiones de diseño para cumplir con todos los requisitos.

Arquitectura Lambda: capa Batch, Speed y Serving

(image) *Arquitectura Lambda con capas Batch, Speed y Serving.* La solución adoptará una **arquitectura Lambda** clásica, que separa el procesamiento de datos en tres capas: **Batch** (lote), **Speed** (tiempo real) y **Serving** (consulta). Esta arquitectura permitirá manejar datos históricos a gran escala y flujos de eventos en tiempo real simultáneamente. En la capa *Batch* se mantiene un **dataset maestro** append-only y se calculan periódicamente vistas globales con todos los datos disponibles (por ejemplo, re-entrenamiento nocturno de un modelo con *batch data* acumulada). En paralelo, la capa *Speed* ingiere eventos entrantes (por ejemplo transacciones financieras o eventos de usuarios) a través de un *stream* (Kafka/Kinesis) y genera vistas de baja latencia con datos recientes aún no incorporados en batch. Finalmente, la capa *Serving* combina las salidas de batch

(precisas pero con mayor latencia) y speed (inmediatas pero potencialmente menos precisas) en un almacén accesible para consultas o APIs de inferencia.

Esta arquitectura híbrida garantiza tanto **procesamiento en tiempo real** como **análisis histórico profundo**, lo cual es crítico en fintech. Por ejemplo, se puede detectar fraude en transacciones al vuelo mediante la capa speed, mientras que la capa batch asegura un registro completo para informes regulatorios o re-cálculo de modelos con todos los datos. En resumen, Lambda Architecture facilita entrenar modelos con datos históricos y aplicarlos sobre nuevos eventos en streaming, equilibrando precisión y latencia en escenarios fintech exigentes.

Prevención de *Data Leakage* en los pipelines de ML

Un aspecto fundamental del diseño es **evitar el *data leakage*** (fuga de información) durante la preparación de datos y entrenamiento del modelo. *Data leakage* ocurre cuando información del conjunto de prueba (o del futuro) se filtra inadvertidamente al conjunto de entrenamiento, llevando a evaluaciones demasiado optimistas y modelos que fallan en producción. Para prevenirlo, se seguirán estas buenas prácticas:

- **Separación estricta de conjuntos**: Realizar la división de datos en *train/test* al inicio, *antes de cualquier* etapa de preprocesamiento (10. Common pitfalls and recommended practices — scikit-learn 1.6.1 documentation). Ninguna transformación (normalización, imputación, codificación, etc.) debe calcularse tomando datos del conjunto de prueba. Por ejemplo, si se escala una variable por su media, esta media se calculará **solo con el subset de entrenamiento**, nunca con datos de prueba. Esto evita que información estadística del test influya en el entrenamiento.
- **Pipelines de preprocesamiento**: Utilizar los pipelines de scikit-learn (Pipeline o make_pipeline) para encadenar transformaciones y modelos garantiza que las llamadas a fit se apliquen únicamente sobre los datos de entrenamiento, mientras que las transformaciones aprendidas se aplican a los datos de prueba mediante transform. Este enfoque reduce el riesgo de olvidar separar los datos manualmente y asegura que no se *fitee* el preprocesamiento con datos fuera del entrenamiento.
- **Validación adecuada**: Emplear validación cruzada y procedimientos como *K-fold* o *TimeSeriesSplit* (para series temporales) dentro de pipelines, de modo que cada pliegue entrene transformaciones solo con sus propios datos de entrenamiento y jamás vea datos del fold de validación. Asimismo, evitar crear *features* utilizando directamente variables que no

estarían disponibles en inferencia (ejemplo: usar el futuro para predecir el pasado es una fuga de datos de tipo *look-ahead*).

Adicionalmente, se documentarán ejemplos de qué **no se debe hacer**, para contrastar: por ejemplo, demostrar que si uno normaliza usando toda la data antes de *split*, los resultados en test parecen mejores de lo que realmente serían, evidenciando el leakage. Con estos lineamientos, el pipeline de datos mantiene una separación correcta entre entrenamiento y prueba, asegurando estimaciones de desempeño realistas.

División de datos de entrenamiento y prueba (*train/test split*)

Es crucial dividir correctamente los datos en entrenamiento, validación y prueba para evaluar el modelo de forma confiable. En este proyecto se manejarán dos escenarios: **datos IID (mezclados)** y **datos temporales**. Para datos convencionales (no secuenciales), se usará train_test_split de scikit-learn con el parámetro stratify cuando corresponda. El uso de stratify=y asegura que la proporción de clases en los subconjuntos de train y test refleje la del conjunto original, lo cual es importante en casos de **clases desequilibradas** (por ejemplo, transacciones fraudulentas vs legítimas). De esta forma, evitamos sesgos debidos a una distribución distinta en el conjunto de prueba y garantizamos que el modelo se evalúe en condiciones similares a las de entrenamiento.

Por otro lado, cuando trabajemos con **series de tiempo** (p. ej., cotizaciones financieras o secuencia cronológica de eventos), **no se barajarán los datos aleatoriamente**. En su lugar, se respetará el orden temporal: típicamente entrenando el modelo en un intervalo de tiempo anterior y probándolo en datos futuros (por ejemplo, entrenamiento con datos de 2023 para probar en datos simulados de 2024). Para esto, podríamos usar TimeSeriesSplit de scikit-learn, que genera cortes consecutivos donde los índices de prueba son siempre posteriores a los de entrenamiento. Esto evita introducir información futura en el entrenamiento (otro tipo de *leakage* temporal). Si se requiere validación cruzada temporal, cada pliegue seguirá respetando la causalidad temporal.

En resumen, se tendrá especial cuidado en: (1) **Estratificar** por etiqueta en problemas de clasificación no temporales para mantener balance de clases, (2) **No mezclar orden temporal** en problemas secuenciales, y (3) documentar claramente el procedimiento de split en el repositorio para reproducibilidad (incluyendo la semilla random_state cuando aplique, para resultados consistentes).

Procesamiento de Big Data con Apache Spark

Dado que el dataset base (eventos de e-commerce adaptados al caso fintech) puede ser grande, y además integra múltiples fuentes, se utilizará **Apache Spark**

para el procesamiento distribuido. En particular, se implementará al menos un **join optimizado con broadcast** y varias técnicas de optimización de consultas en Spark SQL.

- **Broadcast Join**: En Spark, los *joins* entre un conjunto de datos muy grande y otro pequeño pueden optimizarse transmitiendo (broadcasting) la tabla pequeña a todos los nodos, en lugar de barajar la grande. En el proyecto, supongamos que tenemos un dataset principal de transacciones y una tabla pequeña de referencia (por ejemplo, una lista de clientes VIP o un catálogo de riesgos por categoría). Se usará spark.sql.broadcast o el hint /*+ BROADCAST(*)*/ para forzar un **broadcast hash join** de ese dataset pequeño. Esto evitará costosos *shuffles* de la tabla grande a través de la red, mejorando significativamente la velocidad de la unión. Un broadcast join envía la tabla pequeña a cada nodo del cluster, de modo que la unión se realiza localmente en cada partición de la tabla grande. Spark de hecho puede decidir automáticamente usar *broadcast* si la tabla menor está por debajo de cierto umbral (por defecto ~10 MB), pero explicitaremos este caso para demostrar dominio de la técnica. Se incluirá en la documentación el *explain plan* de Spark antes y después de aplicar el broadcast join, mostrando la diferencia en el plan de ejecución.
- **Optimizaciones de Spark SQL**: Para maximizar el rendimiento, aplicaremos varias técnicas soportadas por Spark 3.x:
    - **Predicate Pushdown**: Aseguraremos que los filtros (WHERE clauses) se apliquen lo más temprano posible, incluso empujándolos hacia la capa de lectura de datos (por ejemplo, Parquet). Esto significa que Spark limitará la lectura de datos desde el origen según las condiciones de filtro, reduciendo I/O. En la práctica, al leer de un archivo Parquet o base columnar, Spark solo leerá las columnas y bloques necesarios cumpliendo la condición, *saltando* datos innecesarios.
    - **Partition Pruning**: Organizaremos los datos de historial en particiones (por ejemplo, por fecha o por tipo de evento). Si una consulta o job de Spark aplica un filtro en la columna de partición (ej: fecha = 2025-01-01), Spark **prunará** (omitirá) todas las particiones que no satisfacen ese filtro, leyendo solo la partición relevante. Esto es vital en big data: si nuestros datos de transacciones están particionados por mes, una consulta de un mes específico no disparará un escaneo de todos los archivos, sino únicamente del mes objetivo.

- **Dynamic Partition Pruning (DPP)**: Este proyecto habilitará *Spark 3 Adaptive Query Execution*, que incluye *pruning* dinámico. En un join entre una tabla grande particionada (ej: transacciones por fecha) y otra dimensión filtrada (ej: fechas de interés), Spark puede tomar el resultado del filtro en la dimensión pequeña y aplicarlo a la tabla grande durante la ejecución. Por ejemplo, si la consulta pide datos de *dimensión.fecha = '2025-01'*, Spark derivará que solo debe leer la partición enero 2025 de la tabla de transacciones, incluso si la condición originalmente venía de la tabla de dimensiones. Esto ocurre **en tiempo de ejecución**: Spark obtiene las llaves filtradas de la tabla pequeña y pruna las particiones de la grande antes de el join, ahorrando tiempo.
- **Adaptive Query Execution (AQE)**: Mantendremos spark.sql.adaptive.enabled=true para que Spark ajuste planes sobre la marcha. Por ejemplo, Spark podría decidir cambiar un join de sort-merge a un broadcast join si al momento de ejecutarlo detecta que la tabla es lo suficientemente pequeña. AQE también coalesce partitions (combina particiones pequeñas) y optimiza la paralelización para evitar tareas diminutas o desbalanceadas. Estas optimizaciones automáticas se reflejarán en los *logs* de Spark, que documentaremos.
- **Bloom Filter Join**: Con Spark 3.3+, se introdujo la posibilidad de usar **Bloom filters** en joins para filtrar filas no coincidentes en la etapa de lectura. Implementaremos una demostración donde, para un join con una tabla grande, se activa un filtro Bloom basado en las llaves de la tabla pequeña (por ejemplo, IDs de usuario presentes) para evitar leer de la tabla grande registros cuyo ID no esté en la pequeña. Esto actúa como un filtro de pertenencia probabilístico muy eficiente en espacio, reduciendo aún más el volumen de datos procesado en el join.

Todas estas técnicas combinadas harán que nuestros **pipelines Spark** sean escalables y rápidos. Por ejemplo, podríamos combinar broadcast join + DPP: primero Spark broadcastea la tabla pequeña de fechas filtradas, luego inserta ese filtro en la tabla grande de transacciones para solo leer las particiones necesarias. En la documentación, proporcionaremos fragmentos de código PySpark para cada técnica (por ejemplo, demostrando el uso de spark.conf.set('spark.sql.optimizer.dynamicPartitionPruning.enabled', True) y analizando planes físicos con df.explain()).

CI/CD con GitHub Actions y Docker

Para acercar el proyecto a prácticas de ingeniería reales, se establecerá un pipeline de **Integración Continua/Despliegue Continuo (CI/CD)** utilizando **GitHub Actions** y **Docker**. El repositorio GitHub del proyecto incluirá un workflow de Actions que automatice tareas clave en cada cambio importante de código (por ejemplo, *push* o *pull request* a la rama main):

1. **Construcción del entorno**: El pipeline disparará la construcción de una imagen Docker que encapsula todas las dependencias del proyecto (Python, librerías de ML, Spark, etc.). Esto garantiza que el entorno es consistente y reproducible. En el archivo YAML de Actions se definirá un job para *build* que ejecute docker build usando el Dockerfile incluido. Si la imagen resulta pesada, se podría optar por usar cachés o separar en dos imágenes (una para entrenamiento y otra para despliegue).

2. **Ejecución de pruebas automáticas**: Tras construir la imagen (o en un entorno virtual preconfigurado), se ejecutarán tests unitarios y de integración. Por ejemplo, pruebas para verificar que las funciones de preprocesamiento no filtran datos del futuro, que el modelo entrena y predice sin errores en un subconjunto de datos, o que los scripts Spark producen las salidas esperadas. GitHub Actions provee logs de estas ejecuciones; cualquier falla hará que el pipeline marque estatus rojo, evitando incorporar código defectuoso.

3. **Linting/Formato**: Se puede integrar pasos de comprobación de estilo de código (con *flake8*, *black*, etc.) para asegurar un código limpio y estándar. No es un requisito técnico explícito, pero agrega profesionalismo al repositorio.

4. **Container Registry**: Opcionalmente, el workflow podría publicar la imagen Docker resultante en un registro (por ejemplo, GitHub Container Registry o AWS ECR) con una etiqueta versionada. Esto facilita luego desplegar esa imagen en producción o entornos de prueba. Por ejemplo, tras pasar los tests, la imagen se etiqueta como ml-fintech-project:latest y se sube a ECR, preparando la fase de despliegue.

5. **Despliegue Automatizado**: Si el proyecto se integra con la nube, un paso final del pipeline podría, por ejemplo, llamar a AWS (a través de Terraform, AWS CLI, o SDK) para actualizar un endpoint SageMaker o desplegar un contenedor nuevo. En este caso de portafolio, podríamos simular el despliegue notificando en los logs o creando un release en GitHub, dado que no siempre se tiene una infraestructura real activa. No obstante, dejaremos listo un script (y documentación) para realizar el *deployment* manualmente con facilidad.

Todo este proceso de CI/CD demuestra capacidad de **automatizar la ingeniería de ML**: cada cambio de código dispara validaciones y empaquetados consistentes, reduciendo errores humanos. En el README se incluirá un badge de CI para evidenciar el estado del build, y se explicará cómo está compuesto el workflow de GitHub Actions (archivos .github/workflows/…yml con comentarios sobre cada job). El uso de **Docker** asegura también que cualquier evaluador pueda ejecutar el proyecto localmente con un simple docker run, sin incurrir en problemas de dependencias.

Despliegue del Modelo y MLOps

La etapa de **despliegue** del modelo es clave para demostrar habilidades de MLOps. Se contemplarán dos enfoques, ambos relevantes en fintech, y se podría implementar uno u otro (o documentar ambos como alternativa):

- **Opción 1: AWS SageMaker** – Aprovechando la experiencia del usuario con AWS, se puede usar SageMaker para el ciclo completo: entrenar el modelo en un *notebook instance* o job de entrenamiento, y luego desplegarlo como un **endpoint HTTPS escalable**. SageMaker manejaría la infraestructura de inferencia y permitiría fácilmente habilitar monitoreo de modelos. En el proyecto de portafolio, por simplificar costos, quizás no se use un endpoint continuo 24/7, pero sí se podría demostrar el proceso: tras entrenar el modelo, emplear el SDK de SageMaker (boto3 o sagemaker Python SDK) para serializar el modelo (por ej. a S3) y lanzar un Predictor Endpoint. Este endpoint serviría un modelo de ejemplo (por ej., detección de fraude) vía REST. Se documentará cómo probar el endpoint con la AWS CLI o una pequeña aplicación cliente.
- **Opción 2: FastAPI + AWS Lambda (Serverless)** – Para mostrar versatilidad, otra ruta es crear un **microservicio REST** con FastAPI que encapsule la inferencia del modelo, y luego desplegarlo en AWS Lambda usando contenedores. FastAPI permitiría definir un endpoint /predict que reciba las características de una transacción y devuelva la predicción (fraude sí/no, puntaje de riesgo, etc.). Este servicio se puede empacar en la imagen Docker ya construida y luego subirla a AWS Lambda (que soporta despliegue de contenedores de hasta 10 GB). Con esto se logra un endpoint serverless que escala automáticamente y tiene costo eficiente, exponiendo la inferencia del modelo. AWS API Gateway se usaría para proveer la URL HTTP pública. Este patrón demuestra conocimiento de despliegue **serverless** de ML, muy útil para aplicaciones fintech de baja latencia sin mantener servidores.

Independientemente de la opción elegida, el proyecto incluirá **infraestructura como código** (IaC) para reproducir el despliegue. Por ejemplo, podría haber un

script de CloudFormation/Terraform o simplemente instrucciones paso a paso para:

- Crear roles de IAM seguros (con permisos mínimos necesarios para el modelo, S3, etc.).
- Configurar variables de entorno (claves, endpoints) de forma segura.
- Realizar *testing* post-deployment: se proveerán ejemplos de invocación del modelo, verificando que se obtiene la respuesta esperada.

Esta sección del portafolio resalta habilidades de **MLOps**: no basta con desarrollar el modelo, sino que se demuestra cómo integrarlo en un servicio listo para consumo. También se tocará el tema de **versionado de modelos** (por ejemplo, conservar la versión anterior del modelo desplegado hasta validar la nueva), y de **escalabilidad** (en SageMaker, elegir instance types apropiados; en Lambda, considerar limitaciones de memoria/timeout para modelos grandes). Todo estará claramente explicado en la documentación, de modo que un reclutador pueda ver que se comprendió el *end-to-end* de entregar un modelo a producción.

Feature Store para gestión de *features*

En entornos de ML productivos, especialmente en fintech, es común separar un **Feature Store** para gestionar las características usadas por los modelos. En este proyecto se implementará el paradigma de feature store con un enfoque simplificado: distinguiremos entre un **feature store offline** (para datos históricos de entrenamiento) y un **feature store online** (para inferencia en tiempo real), aunque sea en forma simulada.

- **Feature Store Offline (batch)**: Corresponde a las características calculadas sobre el histórico completo de datos, almacenadas en un repositorio central (a menudo un data lake o almacén columnar). En nuestro caso, las *features* offline podrían generarse en trabajos Spark batch (por ejemplo, agregar el gasto total de cada usuario por mes, número de transacciones sospechosas en su historial, etc.) y guardarse como archivos Parquet en S3 (simulado localmente). Estas features sirven para entrenar el modelo de forma consistente y reproducible. Cada versión de *features* offline estará identificada (por fecha de corte, etc.). Herramientas reales usarían algo como S3, Hive o BigQuery para esto. Por simplicidad, podemos usar archivos locales Parquet/CSV que simulen esa capa offline. Lo importante es que **la misma lógica** de cálculo se aplique luego al online.
- **Feature Store Online (tiempo real)**: Son las features disponibles en inferencia para cada evento o entidad, optimizadas para acceso de baja

latencia (milisegundos). Como ejemplo, podríamos utilizar **Redis** (almacén en memoria) para guardar, por cada usuario, las últimas características actualizadas necesarias para el modelo de predicción en tiempo real. Por ejemplo: "usuario X -> su puntaje crediticio actualizado, su saldo promedio de 7 días, etc.". Cuando llegue un nuevo evento de ese usuario a la API de modelo, el servicio consultaría a Redis para obtener estas features al instante y realizar la predicción. Redis es una elección común para feature stores online por su latencia sub-milisegundo y throughput alto. En AWS, SageMaker Feature Store internamente provee un store online con DynamoDB o Redis y un store offline con S3.

En el proyecto, se mostrará cómo sincronizar ambos stores: típicamente, las *features* offline se recalculan en batch (ej: diariamente) y se cargan al store online para disponibilidad inmediata. También, a medida que entran nuevos eventos, la capa speed puede actualizar en Redis ciertas features (ej: contador de transacciones últimas 10 min). Esta consistencia es esencial para evitar *training-serving skew* (que el modelo en producción reciba features con definiciones distintas a las usadas en entrenamiento).

Si bien implementar un feature store completo excede un portafolio, simularemos la idea: posiblemente con un pequeño módulo que lea las features precomputadas (offline) y las inserte en un *store* en memoria antes de ejecutar la inferencia en streaming. Documentaremos las estructuras de datos usadas (por ejemplo, claves de Redis formadas como feature:{userid}) y cómo se haría en un entorno real. El objetivo es demostrar entendimiento de que un **feature store centralizado** permite reutilizar features entre entrenamiento e inferencia de manera consistente y a escalas de producción.

Integración de múltiples fuentes de datos

Para enriquecer el caso de uso y exhibir habilidades de *data engineering*, el proyecto integrará **múltiples fuentes de datos**. Partimos del dataset base de *e-commerce* proporcionado (campos: event_time, event_type, product_id, category_id, category_code, brand, price, user_id, user_session). Este dataset, originalmente de navegación/compra en e-commerce, se adaptará para un contexto fintech imaginado. Por ejemplo, podríamos reinterpretar cada registro como un **evento transaccional**:

- user_id como identificador del cliente,
- price como monto de una transacción,
- event_type indicando tipo de evento financiero (purchase = compra con tarjeta, view = consulta de saldo, etc.),

- categorías y marcas podrían mapearse a categorías de gasto o tipos de comercio.

A ese núcleo de datos, añadiremos **datos sintéticos** adicionales para asemejar un entorno fintech:

- *Datos de usuario (perfil)*: Generar un dataset sintético de clientes con atributos como puntaje crediticio, ingreso mensual, tipo de cuenta, etc. Esto podría crearse con librerías como Faker o simplemente sampling de distribuciones realistas. Serviría para aportar features estáticas por usuario.
- *Eventos financieros simulados*: Por ejemplo, un stream de eventos de transacciones bancarias (depósitos, retiros) o alertas de geolocalización (posiciones de cajero automático). Estos datos se pueden sintetizar correlacionados con las sesiones de e-commerce para crear un panorama unificado.
- *Datos macro o de mercado*: Quizá una serie temporal diaria de cotización de una criptomoneda o tasa de interés, para ver cómo podría incorporarse como feature global en un modelo (esto demostraría ingestión de fuentes públicas vía API si se desea).

La integración de estas fuentes ocurrirá en el pipeline de ingestión batch y streaming. En la capa batch, Spark puede hacer joins entre la tabla principal de eventos de e-commerce (transacciones) y, por ejemplo, la tabla de perfil de usuarios, para crear un dataset enriquecido de entrenamiento. Aquí destacaremos el uso de **joins correctamente particionados y quizás el broadcast join** (la tabla de usuarios es más pequeña, por lo que puede ser broadcast). De este modo cada registro de transacción incorporará las características demográficas/financieras del usuario correspondiente, aumentando el poder predictivo del modelo.

En la capa streaming, se demostrará la ingesta de al menos dos flujos: por ejemplo, un **Kafka topic** para eventos de compra online y otro para **eventos de retiro en cajero** (sintéticos). Se implementará un pequeño consumidor Spark Streaming o Kafka Streams que una estos flujos en una vista en ventana (window) para generar alertas o alimentar al modelo en tiempo real. La orquestación de múltiples fuentes mostrará la capacidad de manejar **datos heterogéneos** (diferentes formatos o velocidades).

Toda la generación de datos sintéticos estará documentada en notebooks de exploración, indicando supuestos y distribuciones utilizadas (p.ej., generar montos de transacción con distribución log-normal para simular gastos, generar timestamps de eventos siguiendo patrones diurnos, etc.). Con esta combinación de datos reales y sintéticos, simularemos un escenario fintech convincente donde

el modelo pueda, por ejemplo, predecir probabilidad de fraude combinando el comportamiento de compra en línea con recientes retiros en cajero y el perfil crediticio del usuario.

Procesamiento de streaming en tiempo real (Kafka/Kinesis)

Para cubrir el componente de **datos en streaming**, el proyecto integrará **Apache Kafka** como sistema de mensajería. Kafka actuará como la fuente de la capa speed de la arquitectura Lambda, entregando eventos en tiempo real al sistema de ML. La configuración incluirá uno o varios **topics** de Kafka, por ejemplo transactions_stream, donde se publicarán eventos de transacciones financieras simuladas continuamente (con campos como user_id, monto, tipo de transacción, timestamp, etc.).

Se desarrollará un **productor Kafka** (una pequeña aplicación Python usando kafka-python o confluent-kafka) que lee los eventos del dataset de e-commerce/financiero adaptado y los envía al topic con una cierta cadencia (simulando un flujo constante de actividad). Asimismo, un **consumidor de streaming** en Spark Structured Streaming leerá del topic en tiempo real. Spark Structured Streaming permite tratar el input de Kafka como un DataFrame incremental continuo, que podemos unir con datos estáticos o aplicar transformaciones en ventanas. Por ejemplo, en nuestro caso el consumidor podría:

- Agregar en una ventana de 5 minutos el número de transacciones y monto total por usuario,
- Unir esa información con la tabla de umbrales de riesgo (estática, quizá broadcast) para detectar anomalías,
- Pasar los datos enriquecidos al modelo de inferencia entrenado.

La inferencia en tiempo real puede manejarse de dos formas: (a) integrando directamente el modelo dentro del job de Spark (por ejemplo, usando Spark MLlib o cargando un modelo scikit-learn en cada micro-batch para puntuar los eventos), o (b) enviando los eventos a la API de predicción desplegada (FastAPI/Lambda) y recopilando la respuesta. Para demostrar la arquitectura completa, implementaremos la segunda opción: el stream de Spark formateará cada evento como un JSON y hará peticiones HTTP al endpoint de FastAPI/Lambda para obtener la predicción de fraude/riesgo en tiempo real. Esto cierra el ciclo de la Lambda Architecture: los nuevos datos son procesados inmediatamente por la capa speed con un modelo entrenado en batch.

Es importante destacar que este **pipeline de streaming** operará de forma continua. Usaremos características de **tolerancia a fallos** de Spark streaming

(checkpointing) para asegurar que si se reinicia el proceso, no se re-procesen eventos previos (o se procesen exactamente una vez, si se configura apropiadamente). También se considerará la latencia: dado que nuestro volumen es moderado, Spark Streaming en micro-batches de, digamos, 1 segundo será suficiente. Si se quisiera menor latencia, se podría explorar Kafka Streams o Flink, pero eso excede el alcance.

La documentación incluirá instrucciones para correr Kafka en local (por ejemplo mediante Docker Compose con ZooKeeper y Broker de Kafka), crear los topics necesarios, y luego arrancar el script de Spark Streaming (ya sea usando spark-submit o un notebook). Se mostrará en logs cómo los eventos fluyen y generan predicciones inmediatas. Esto evidenciará la habilidad de **construir pipelines de datos en tiempo real** integrados con ML, competencia muy relevante en fintech (ej. detección de fraude al vuelo, recomendaciones inmediatas, etc.).

Orquestación de pipelines con Airflow o AWS Glue

Para manejar la complejidad de múltiples procesos (batch ETL, entrenamiento, actualización de feature store, streaming, etc.), implementaremos la **orquestación** mediante flujos de trabajo administrados. Dos alternativas viables son **Apache Airflow** (auto-alojado o MWAA en AWS) o **AWS Glue Workflows**. Cualquiera de las dos exhibirá la capacidad de coordinar tareas dependientes de forma robusta.

Enfoque Airflow: Se creará un *DAG (Directed Acyclic Graph)* de Airflow que represente el pipeline completo:

- Una tarea para ingerir los datos brutos (por ejemplo, ejecutar un script PySpark que carga los CSV de e-commerce y genera la tabla raw en el data lake).
- Una tarea dependiente que ejecuta el procesamiento de *features* batch (Spark job para computar agregados, limpieza, etc., guardando resultados en feature store offline).
- Una tarea para **entrenar el modelo** (ejecutar un script Python que lee las features offline y entrena el modelo de ML, guardándolo en S3 o local).
- Una tarea de **evaluación** que, tras entrenamiento, valida el modelo con métricas y podría condicionar el despliegue.
- Si todo es satisfactorio, una tarea final de **despliegue** (que llame al script/endpoint de SageMaker o actualice el servicio FastAPI).

Este DAG puede programarse para correr diariamente a cierta hora, o ser desencadenado manualmente. Airflow permitirá monitorear cada paso, reintentar

en caso de fallas, y llevar un historial de runs. Usaremos las características de **dependencias y scheduling** de Airflow para asegurarnos de que, por ejemplo, el entrenamiento no inicie antes de que las features estén calculadas. Airflow es muy adecuado ya que es **Python nativo y extensible**, integrándose con nuestros scripts Python fácilmente. Se podrían utilizar *Operators* personalizados o sencillos BashOperator/PythonOperator para ejecutar cada sección. Dado que Airflow puede conectarse a AWS, podríamos usar SageMakerTrainingOperator para manejar entrenamiento en la nube, etc., pero como es un portafolio, quizás mantengamos los operadores genéricos con llamadas a scripts.

Enfoque AWS Glue Workflows: Alternativamente, para demostrar uso de servicios AWS administrados, podríamos implementar la orquestación con **Glue Workflows**. Glue Workflows permite encadenar **crawlers, jobs de ETL (Spark)** y triggers en AWS. En este caso, definiríamos un Workflow en AWS Glue que:

- Arranque un **AWS Glue Crawler** para detectar el esquema de nuevos datos en S3 (por si incorporamos datos nuevos incrementalmente).
- Luego ejecute un **Glue ETL Job** (escrito en PySpark) que haga la transformación batch de datos crudos a features.
- Después lance otro Job o un script Python (Glue Python Shell Job) que entrene el modelo usando, por ejemplo, data en S3.
- Finalmente, podría invocar una función Lambda que implemente el despliegue (o notificar completitud).

Glue Workflows provee un seguimiento visual del pipeline en la consola AWS, y es *serverless*, lo que simplifica no gestionar servidores para orquestación. Sin embargo, la elección entre Airflow y Glue puede comentarse: Airflow es más flexible y multi-nube, Glue es conveniente si todo el stack está en AWS. Para demostrar amplitud, podríamos diseñar el DAG en Airflow pero mencionar cómo sería el equivalente en Glue.

En la documentación, incluiremos ejemplos de código del DAG Airflow (archivo .py con la definición, usando quizás la API TaskFlow de Airflow 2 para mayor claridad). Se resaltará la gestión de dependencias (usando >> en Airflow) y algunas configuraciones útiles (como retries automáticos, o envío de email en caso de fallo para simular alertas). Esto mostrará que se entiende el rol de un **orquestador** en MLOps: asegurar que los distintos componentes corran en el orden correcto, de forma confiable y monitorizable.

Simulación de entorno productivo y monitoreo

Para acercar el proyecto a un **entorno productivo real**, se simularán componentes de infraestructura y se implementará monitoreo básico de datos/modelo. Algunos aspectos a destacar:

- **Infraestructura simulada**: Utilizaremos Docker Compose para levantar servicios necesarios (por ejemplo, contenedores de Kafka, Zookeeper, Redis, quizá un contenedor local de Airflow) de manera que cualquier persona pueda recrear el entorno en su máquina. Esto actúa como un mini-entorno productivo: un *cluster* Spark local, un *broker* de mensajería, un *feature store* en memoria, etc. También se puede simular diferentes entornos (dev/staging/prod) mediante archivos de configuración distintos, aunque en un solo proyecto de portafolio quizás baste con uno. El objetivo es demostrar familiaridad con conceptos de despliegue de infraestructura, aislando servicios en contenedores como se haría en Kubernetes u otras plataformas.
- **Monitoreo básico**: Una vez desplegado el modelo, necesitamos monitorear su desempeño y la salud del pipeline:
    - **Métricas del modelo**: Se implementará un registro de métricas de predicción en tiempo real, por ejemplo, contando cuántas predicciones de "fraude" positivas se están dando por hora, o calculando el tiempo de respuesta promedio de la API de inferencia. Esto podría simplemente loguearse en CloudWatch (si usamos AWS Lambda) o en logs de la aplicación (que Airflow o Docker podrían capturar). También se podría simular la monitorización de calidad del modelo: dado que quizás no tenemos *ground truth* inmediato en streaming, podemos monitorizar la distribución de las *features* y de las puntuaciones del modelo para detectar *data drift*. Herramientas especializadas como **EvidentlyAI** o **WhyLabs** existen, pero en el portafolio bastará con un script que compare estadísticas recientes vs del set de entrenamiento (por ejemplo, media y desviación de la probabilidad predicha, porcentaje de eventos fuera de rango esperado, etc.).
    - **Monitoreo de pipeline**: Airflow ya provee un UI con estado de tareas, pero además podríamos configurar alertas simples: por ejemplo, si el job de entrenamiento falla, emitir un mensaje (simulado por console log o email). Si usamos AWS, podríamos integrar CloudWatch Alarms para alertar si el endpoint SageMaker tiene errores o si la latencia excede cierto threshold. Documentaremos cómo se podría ampliar esto en un entorno real

(ex. integración con Datadog, Prometheus, etc. para monitoreo más robusto).

- **Logging estructurado**: Se fomentará en el código el uso de logs en formato JSON o claro, indicando timestamps y contexto, para facilitar debuggear problemas en producción. Por ejemplo, cada inferencia registrará un ID de correlación, el resultado, y tiempo tomado.

En resumen, aunque no desplegaremos un stack completo de monitoreo en nube por cuestiones de alcance, sí demostraremos consciencia de su importancia. Un **ML Engineer** debe asegurarse de que el modelo en producción se comporta bien y cuenta con visibilidad; por ello incluiremos en el README una sección de *MLOps Monitoring*, describiendo cómo se revisaría la performance periódicamente y cómo se detectaría si el modelo requiere re-entrenamiento (por ejemplo, si cierta métrica de drift excede un umbral, eso indicaría que la data nueva difiere mucho de la de entrenamiento, señal para lanzar el pipeline batch nuevamente).

Estructura del proyecto y documentación

El proyecto se publicará en GitHub con una **estructura de carpetas profesional**, facilitando la navegación y reproducibilidad. A grandes rasgos, la estructura podría ser:

```
MLFintechProject/
├── README.md
├── data/            # Datos de ejemplo (muestreo del e-commerce, datos sintéticos)
├── notebooks/       # Notebooks Jupyter para exploración y documentación de EDA
├── src/             # Código fuente organizado en módulos
│   ├── etl/         # Scripts de extracción y transformación de datos
│   ├── features/    # Cálculo de features (offline & online syncing)
│   ├── models/      # Entrenamiento y evaluación de modelos
│   ├── inference/   # Código de servicio de inferencia (FastAPI app, etc.)
│   └── utils/       # Utilidades comunes (config, logging, helpers)
├── airflow/         # DAGs de Airflow si aplica, o definiciones de Glue jobs
├── docker/          # Dockerfile y config para construir imágenes
```

```
├── .github/          # Workflows de GitHub Actions para CI/CD
└── requirements.txt  # Dependencias de Python
```

En la raíz, el **README.md** servirá como guía completa: incluirá una descripción del proyecto, instrucciones de configuración (por ejemplo "requiere Docker y Docker Compose, cómo iniciar Kafka, etc."), y guías de uso (**cómo reproducir** cada etapa: correr ETL, entrenar modelo, lanzar streaming, etc.). Importante, el README también explicará **cada decisión técnica** tomada, enlazando al código/notebooks correspondiente. Por ejemplo: "Se eligió Kafka para streaming por XYZ... (ver carpeta streaming/)". Esto mostrará a los evaluadores un razonamiento claro detrás de la implementación.

Aunque se proveerán Jupyter notebooks para la exploración inicial de datos y quizá un prototipo del modelo, el enfoque principal estará en **scripts y modularización** del código. Los notebooks serán tratados como reportes de análisis exploratorio, siguiendo la recomendación de usarlos solo para tareas ad-hoc y narrativas (How to Use Exploratory Notebooks [Best Practices]). Todo el código crítico para pipelines estará en módulos Python reutilizables (lo cual es más adecuado para producción). Para demostrar reproducibilidad, incluiremos ejemplos de ejecución por línea de comando o Makefile. Por ejemplo, make train podría ejecutar el script de entrenamiento end-to-end, make serve levantar el servicio FastAPI local, etc.

También se añadirá documentación en **docstrings** y comentarios en el código para facilitar la comprensión. Si el alcance lo permite, podríamos habilitar GitHub Pages o un wiki con la documentación, pero con un README bien estructurado probablemente baste. Diagramas de flujo (por ejemplo, un diagrama ilustrando cómo los datos fluyen de la capa batch y speed al feature store y al modelo) serán incluidos en la medida de lo posible, ya sea como imágenes en el README u hojas mermaid diagram.

En conclusión, la estructura y documentación asegurarán que el proyecto sea **fácil de revisar y ejecutar**. Un reclutador o ingeniero podrá clonar el repo, seguir las instrucciones para levantar los servicios necesarios (gracias a Docker Compose y scripts) y reproducir los resultados. La atención al detalle en la organización del proyecto y la claridad de la documentación reflejarán las *best practices* de un Machine Learning Engineer profesional. Cada componente implementado estará respaldado con explicaciones (y citas a fuentes cuando proceda) en la documentación, evidenciando tanto el saber hacer práctico como el fundamento teórico de las decisiones. Esto hará de este proyecto de portafolio una demostración integral de competencias para una posición de **ML Engineer en fintech**, cubriendo desde el manejo de big data y entrenamiento robusto, hasta el despliegue, orquestación y consideraciones operativas (MLOps).

Fuentes y Referencias:

- Documentación scikit-learn – prevención de data leakage y uso de pipelines
- Apache Spark optimizations – broadcast joins y técnicas de pruning/pushdown
- Arquitectura Lambda – procesamiento batch vs. streaming en ML y finanzas
- AWS Blog – Feature Store offline/online (Redis, S3) en sistemas de baja latencia
- Apache Airflow & MLOps – orquestación de flujos de ML en producción
- KDNuggets – pipelines de datos en tiempo real con Kafka y Spark, entre otros.