**Documentação da API de Previsão de Score de Crédito Quantum Finance**

**1. Introdução**

Este documento detalha a API REST desenvolvida para servir o modelo de Machine Learning de previsão de score de crédito do projeto Quantum Finance. A API foi projetada com foco em escalabilidade, observabilidade e manutenibilidade, seguindo as melhores práticas de MLOps e arquitetura serverless na AWS. O objetivo principal é fornecer uma interface robusta e eficiente para a inferência do modelo, permitindo que aplicações frontend, como o aplicativo Streamlit, consumam as previsões de score de crédito de forma segura e confiável.

**1.1. Visão Geral do Projeto Quantum Finance Credit Score**

O projeto Quantum Finance Credit Score é uma iniciativa abrangente de Machine Learning que visa prever o score de crédito de indivíduos. Ele engloba desde a experimentação e modelagem (utilizando DagsHub com S3 e MLflow para versionamento de dados e modelos) até a entrega final no frontend. A API aqui documentada representa a camada de inferência, desacoplando o processo de predição do ciclo de treinamento do modelo.

**1.2. Objetivo da API**

O principal objetivo desta API é disponibilizar o modelo de classificação de score de crédito (treinado e versionado via MLflow/DagsHub) por meio de um endpoint RESTful. Isso permite que outras aplicações consumam as previsões do modelo de forma programática, garantindo:

* **Isolamento**: Separação clara entre o código de inferência e o código de treinamento/modelagem.
* **Leveza**: Pacote de deploy otimizado para ambientes serverless, com dependências mínimas.
* **Reprodutibilidade**: Garantia de que o modelo em produção é o mesmo que foi treinado e validado, com rastreabilidade completa.
* **Escalabilidade**: Capacidade de lidar com um grande volume de requisições de forma elástica, inerente à arquitetura serverless.
* **Observabilidade**: Monitoramento contínuo do desempenho da API e do modelo em produção.

**1.3. Tecnologias Utilizadas**

A arquitetura da API é construída sobre um stack tecnológico moderno e robusto, predominantemente na AWS, com foco em soluções serverless e ferramentas de MLOps:

* **AWS Lambda**: Serviço de computação serverless que executa o código da API em resposta a eventos.
* **Amazon API Gateway**: Serviço que atua como

um "front door" para a API, gerenciando requisições, roteamento e segurança.

* **Amazon ECR (Elastic Container Registry)**: Repositório de imagens Docker para armazenar a imagem containerizada da função Lambda.
* **Amazon S3 (Simple Storage Service)**: Utilizado para armazenar os dados de inferência em tempo real, permitindo auditoria e análise de drift.
* **Amazon CloudWatch**: Serviço de monitoramento e observabilidade para coletar logs e métricas da API e do modelo.
* **MLflow/DagsHub**: Plataforma de MLOps utilizada para rastreamento de experimentos, versionamento de modelos e artefatos, e registro de modelos. O DagsHub atua como o servidor de tracking do MLflow.
* **Python**: Linguagem de programação principal para o desenvolvimento da API e do modelo.
* **Docker**: Utilizado para containerizar a aplicação, garantindo portabilidade e consistência entre ambientes.
* **GitHub Actions**: Ferramenta de CI/CD para automatizar o processo de build, teste e deploy da API.

**1.4. Benefícios e Boas Práticas**

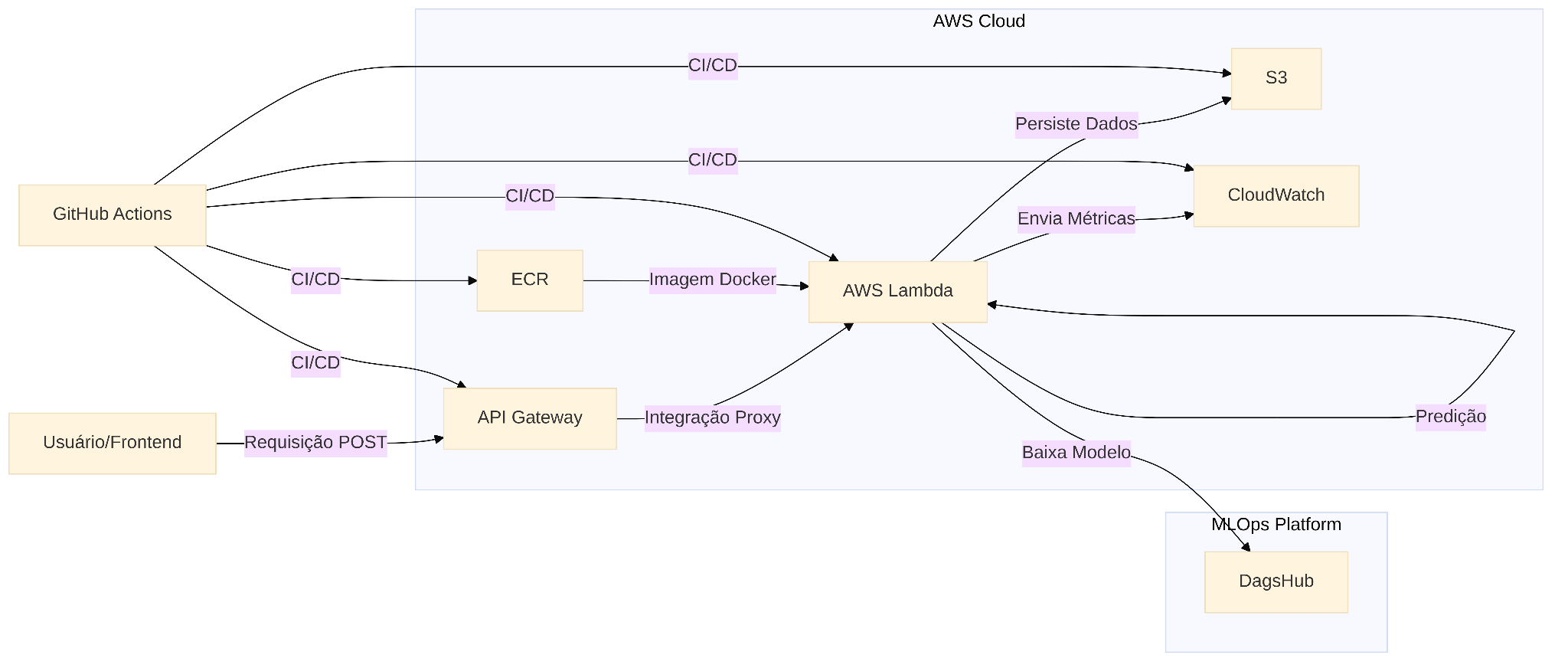
A implementação desta API incorpora diversas boas práticas de MLOps e desenvolvimento de software, resultando em benefícios significativos:

* **MLOps Integrado**: Desde o versionamento de modelos com MLflow/DagsHub até o monitoramento em produção com CloudWatch e a esteira de CI/CD com GitHub Actions, o ciclo de vida do modelo é gerenciado de ponta a ponta.
* **Arquitetura Serverless**: Reduz a sobrecarga operacional, permitindo que a equipe se concentre no desenvolvimento do modelo e da lógica de negócios, além de proporcionar escalabilidade automática e pagamento por uso.
* **Isolamento de Código**: A API é um repositório separado do projeto de modelagem, garantindo que o código de inferência seja leve, com poucas dependências e fácil de implantar.
* **Rastreabilidade e Governança**: O model\_downloader.py garante que a API sempre utilize a versão mais recente do modelo registrado no MLflow, e os metadados do modelo são armazenados para auditoria. A persistência dos dados de inferência no S3 também contribui para a rastreabilidade e detecção de drift.
* **Observabilidade Robusta**: Métricas detalhadas no CloudWatch e logs estruturados permitem um monitoramento eficaz do desempenho da API e do modelo, facilitando a identificação e resolução de problemas.
* **Segurança Aprimorada**: A utilização de API Keys e Usage Plans no API Gateway, juntamente com a configuração de OIDC para o CI/CD, garante que o acesso à API seja controlado e seguro.

**2. Arquitetura da Solução**

A arquitetura da API de previsão de score de crédito é baseada em princípios serverless e de MLOps, utilizando serviços da AWS para garantir escalabilidade, resiliência e observabilidade. A seguir, apresentamos um diagrama que ilustra a interação entre os principais componentes.

**2.1. Diagrama de Arquitetura (Visão Geral)**

**

*Figura 1: Diagrama de Arquitetura da API de Previsão de Score de Crédito.*

**2.2. Componentes da API e suas Interações**

**2.2.1. AWS Lambda (Função score-credit-api)**

O coração da API é a função AWS Lambda score-credit-api. Esta função é responsável por:

* **Carregar o Modelo**: Ao ser inicializada, a função baixa o modelo de Machine Learning (model.pkl) e seus metadados (model\_metadata.json) do MLflow/DagsHub, garantindo que a versão mais recente e correta do modelo seja utilizada.
* **Processar Requisições**: Recebe o payload de entrada via API Gateway, realiza o pré-processamento dos dados (prepare\_payload) e executa a inferência utilizando o modelo carregado.
* **Persistir Dados de Inferência**: Após a predição, os dados de entrada, a predição e os metadados do modelo são persistidos no Amazon S3 para fins de auditoria e análise de drift (write\_real\_data).
* **Emitir Métricas**: Envia métricas detalhadas para o Amazon CloudWatch, permitindo o monitoramento do desempenho do modelo e da API (input\_metrics).
* **Retornar Resposta**: Formata a predição e os metadados do modelo em uma resposta JSON padronizada para o cliente.

**2.2.2. Amazon API Gateway**

O API Gateway atua como o ponto de entrada para a API, expondo o endpoint RESTful para consumo externo. Suas principais funções incluem:

* **Exposição da API**: Fornece um endpoint HTTP para a função Lambda, permitindo que clientes externos a invoquem.
* **Integração Proxy**: Configurado para integrar diretamente com a função Lambda, passando as requisições e respostas de forma transparente.
* **Autenticação**: Utiliza API Keys para controlar o acesso à API, garantindo que apenas clientes autorizados possam fazer requisições.
* **Usage Plan**: Permite definir limites de taxa (rate limits) e cotas de uso (quotas) para as API Keys, protegendo a API contra abusos e garantindo um uso justo.

**2.2.3. Amazon ECR (Elastic Container Registry)**

O ECR é um serviço de registro de imagens Docker totalmente gerenciado. Ele é utilizado para armazenar a imagem containerizada da função Lambda score-credit-api. A containerização garante que o ambiente de execução da API seja consistente e reproduzível, empacotando todas as dependências necessárias junto com o código da aplicação.

**2.2.4. Amazon S3 (Simple Storage Service)**

O S3 é utilizado para armazenar os dados de inferência em tempo real. Cada requisição à API gera um arquivo CSV único no S3, contendo os dados de entrada, a predição, o timestamp da requisição e a versão do modelo utilizada. Este repositório de dados brutos é crucial para:

* **Auditoria**: Permite rastrear todas as inferências realizadas pela API.
* **Análise de Drift**: Facilita a detecção de mudanças na distribuição dos dados de entrada (drift de dados) ou no desempenho do modelo (drift de conceito) ao longo do tempo.
* **Re-treinamento**: Os dados coletados podem ser utilizados para re-treinar o modelo periodicamente, garantindo sua relevância e precisão.

**2.2.5. Amazon CloudWatch**

O CloudWatch é o serviço central para monitoramento e observabilidade da API. Ele coleta e armazena logs da função Lambda e métricas personalizadas, permitindo:

* **Monitoramento de Desempenho**: Acompanhamento de métricas como latência, erros e invocações da função Lambda.
* **Métricas de Negócio/Modelo**: Coleta de métricas específicas do modelo, como a distribuição das previsões de score de crédito e a distribuição das features de entrada, que são cruciais para identificar problemas no modelo em produção.
* **Alertas**: Configuração de alarmes para notificar a equipe sobre anomalias ou problemas críticos.
* **Logs Estruturados**: Facilita a depuração e a análise de causa raiz em caso de falhas.

**2.2.6. MLflow/DagsHub (Versionamento e Download do Modelo)**

O MLflow, com o DagsHub atuando como servidor de tracking, é fundamental para o gerenciamento do ciclo de vida do modelo. Ele permite:

* **Rastreamento de Experimentos**: Registro de parâmetros, métricas e artefatos de cada experimento de treinamento do modelo.
* **Versionamento de Modelos**: Armazenamento de diferentes versões do modelo, facilitando a comparação e a seleção da melhor versão para produção.
* **Download de Artefatos**: A API utiliza o model\_downloader.py para se conectar ao MLflow/DagsHub e baixar o artefato do modelo (model.pkl) e seus metadados, garantindo que a versão correta do modelo seja implantada.

**3. Estrutura do Projeto da API**

A estrutura do projeto da API foi cuidadosamente desenhada para garantir isolamento, modularidade e facilidade de manutenção. O repositório quantum-finance-api-credit-score é independente do repositório de modelagem, contendo apenas o código e as configurações necessárias para a inferência do modelo.

**3.1. Organização de Pastas (quantum-finance-api-credit-score)**

A estrutura de diretórios do projeto é a seguinte:

quantum-finance-api-credit-score/

├── src/

│ └── app.py

├── model/

│ ├── model.pkl

│ ├── model\_metadata.json

│ └── model\_downloader.py

├── tests/

│ ├── conftest.py

│ └── test\_app.py

├── .github/

│ └── workflows/

│ └── ci-cd.yml

├── requirements.txt

├── Dockerfile

├── .gitignore

└── README.md

* **src/**: Contém o código-fonte principal da aplicação, incluindo a lógica do handler da Lambda (app.py).
* **model/**: Armazena o modelo serializado (model.pkl), seus metadados (model\_metadata.json) e o script responsável por baixá-los do MLflow (model\_downloader.py). Este diretório é recriado dinamicamente via script, sempre que a API é inicializada ou atualizada com um novo modelo.
* **tests/**: Contém os testes unitários e de integração para a API.
* **.github/workflows/**: Armazena os arquivos de configuração do GitHub Actions para a esteira de CI/CD.
* **requirements.txt**: Lista as dependências Python necessárias para a execução da API.
* **Dockerfile**: Define a imagem Docker para a função Lambda.
* **.gitignore**: Especifica os arquivos e diretórios a serem ignorados pelo controle de versão do Git.
* **README.md**: Documentação geral do projeto (este documento).

**3.2. Ambiente Virtual e Dependências (requirements.txt)**

Um ambiente virtual Python dedicado é criado dentro da pasta do projeto para garantir o isolamento das dependências. A versão do Python é escolhida para compatibilidade total com o modelo e as bibliotecas utilizadas no treinamento, evitando problemas de serialização.

O arquivo requirements.txt é mantido o mais enxuto possível, contendo apenas as bibliotecas estritamente necessárias para rodar o modelo e comunicar com a infraestrutura AWS. Pacotes de desenvolvimento, testes ou notebooks são propositalmente excluídos para reduzir o peso do deploy e evitar inicialização lenta no AWS Lambda.

**Pacotes incluídos:**

* boto3==1.33.13: Para comunicação com serviços AWS (S3 para armazenar dados reais, CloudWatch para métricas, etc.).
* mlflow==2.22.0: Para download e versionamento do modelo registrado no MLflow (DagsHub).
* scikit-learn==1.6.1: Necessário para carregar e executar o modelo.
* lightgbm==4.0.0: Algoritmo usado no stack final do modelo.
* catboost==1.2.8: Outro algoritmo usado no stack final do modelo.
* numpy==1.26.4: Base para operações numéricas.

**3.3. Configuração do .gitignore**

O arquivo .gitignore é configurado para excluir arquivos e diretórios que não precisam ser versionados no repositório, como:

* .venv/: O ambiente virtual Python.
* model/: O diretório que contém o modelo serializado e seus metadados. Estes arquivos são gerados dinamicamente pelo model\_downloader.py e não devem ser versionados diretamente.

**4. Gerenciamento do Modelo (MLOps)**

O gerenciamento do ciclo de vida do modelo é uma parte crítica desta arquitetura, garantindo que a API sempre utilize a versão correta e mais recente do modelo, com total rastreabilidade. Isso é alcançado através da integração com MLflow/DagsHub e um script dedicado para download do modelo.

**4.1. model\_downloader.py**

O script model\_downloader.py, localizado na pasta model/, é responsável por automatizar o processo de obtenção do modelo. Suas principais funções são:

1. **Conectar ao MLflow Tracking Server**: Estabelece conexão com o servidor MLflow hospedado no DagsHub.
2. **Buscar a Versão Mais Recente**: Identifica a versão mais recente do modelo registrado sob o nome credit-score-model.
3. **Baixar o Artefato**: Realiza o download do arquivo serializado do modelo (model.pkl).
4. **Gerar Metadados**: Cria um arquivo JSON (model\_metadata.json) com informações cruciais sobre o modelo baixado, como nome, versão, run\_id, caminho de origem e timestamp de download.

Este processo garante que a API sempre utilize o modelo mais atualizado e que a versão em produção esteja devidamente documentada e rastreável.

**4.2. Artefatos do Modelo (model.pkl, model\_metadata.json)**

Após a execução do model\_downloader.py, dois artefatos principais são gerados e armazenados na pasta model/:

* **model.pkl**: O arquivo serializado do modelo de Machine Learning treinado. Este arquivo é carregado pela API para realizar as predições.
* **model\_metadata.json**: Um arquivo JSON contendo os metadados do modelo. Exemplo de conteúdo:

{

"model\_name": "credit-score-model",

"version": "2",

"run\_id": "c85566115777495aa40d6814d0bb7102",

"source": "mlflow-artifacts:/.../artifacts/model",

"downloaded\_at": "2025-08-07 18:55:15"

}

**4.3. Rastreabilidade e Governança do Modelo**

A combinação do MLflow/DagsHub com o model\_downloader.py e o model\_metadata.json oferece uma rastreabilidade completa e robusta do modelo em uso. Isso é fundamental para a governança de modelos, permitindo:

* Saber exatamente qual versão do modelo está em produção a qualquer momento.
* Auditar o histórico de versões e as mudanças no desempenho do modelo.
* Facilitar a reversão para versões anteriores, se necessário.
* Garantir a conformidade com regulamentações e políticas internas.

**5. Implementação da Lógica da API (app.py)**

O arquivo app.py contém a lógica central da API, incluindo o carregamento do modelo, o pré-processamento dos dados, a execução da predição, a persistência dos dados de inferência e o envio de métricas. Ele é o ponto de entrada para a função AWS Lambda.

**5.1. Carregamento do Modelo e Metadados**

No início do app.py, o modelo serializado (model.pkl) e seus metadados (model\_metadata.json) são carregados na memória. Isso garante que o modelo esteja pronto para uso assim que a função Lambda for inicializada (warm start).

import joblib

import json

model = joblib.load('model/model.pkl')

with open('model/model\_metadata.json', 'r', encoding="utf-8") as f:

model\_info = json.load(f)

**5.2. Função write\_real\_data (Persistência de Dados de Inferência no S3)**

A função write\_real\_data é responsável por persistir os dados de entrada recebidos pela API, juntamente com a predição e a versão do modelo, no Amazon S3. Esta coleta de dados é vital para auditoria, investigação de drift de dados e drift de conceito no futuro.

* **Estratégia de Nomenclatura**: Cada requisição gera um arquivo CSV no formato {YYYYMMDD\_HHMMSS}\_credit\_score\_data.csv. Isso evita sobrescrita e mantém a granularidade de análise. Em larga escala, um particionamento diário pode ser uma evolução para gerenciar o volume de arquivos.
* **Conteúdo**: O arquivo CSV inclui os dados de entrada, a predição, um timestamp e a versão do modelo.

from datetime import datetime

import boto3

def write\_real\_data(data, prediction):

now = datetime.now()

now\_formatted = now.strftime("%Y-%m-%d %H:%M:%S")

file\_name = f"{now.strftime('%Y%m%d\_%H%M%S')}\_credit\_score\_data.csv"

data["credit\_score"] = prediction

data["timestamp"] = now\_formatted

data["model\_version"] = model\_info["version"]

s3 = boto3.client('s3')

bucket\_name = 'quantum-finance-credit-score'

s3\_path = 'credit-score-real-data'

try:

# Tenta ler o objeto existente para append (se a estratégia fosse de append no mesmo arquivo)

# No entanto, a estratégia atual é um novo arquivo por requisição, então este bloco é mais para referência

existing\_object = s3.get\_object(Bucket=bucket\_name, Key=f"{s3\_path}/{file\_name}")

existing\_data = existing\_object['Body'].read().decode('utf-8').strip().split('\n')

existing\_data.append(','.join(map(str, data.values())))

update\_content = '\n'.join(existing\_data)

except s3.exceptions.NoSuchKey:

update\_content = ','.join(data.keys()) + '\n' + ','.join(map(str, data.values()))

s3.put\_object(Body=update\_content, Bucket=bucket\_name, Key=f"{s3\_path}/{file\_name}")

**5.3. Função input\_metrics (Envio de Métricas para o CloudWatch)**

A função input\_metrics é responsável por enviar métricas personalizadas para o Amazon CloudWatch, permitindo o monitoramento do comportamento do modelo e das features de entrada em produção.

* **Métrica Principal**: Credit Score Prediction com dimensão RiskCategory (mapeada a partir das classes 0/1/2 para Poor/Standard/Good).
* **Métricas de Features**: Métricas de contagem para cada feature de entrada, com suas dimensões (FeatureName, FeatureValue). Isso permite monitorar a distribuição das features de entrada e identificar possíveis drifts.

import boto3

cloudwatch = boto3.client('cloudwatch')

def input\_metrics(data, prediction):

# Métrica principal: Predição do Score de Crédito

risk\_category\_map = {0: 'Poor', 1: 'Standard', 2: 'Good'}

risk\_category = risk\_category\_map.get(prediction, 'Unknown')

cloudwatch.put\_metric\_data(

Namespace='Credit Score Model',

MetricData=[

{

'MetricName': 'Credit Score Prediction',

'Dimensions': [

{

'Name': 'RiskCategory',

'Value': risk\_category

},

{

'Name': 'ModelVersion',

'Value': model\_info['version']

}

],

'Value': 1, # Contagem de previsões por categoria

'Unit': 'Count'

},

]

)

# Métricas de contagem para cada feature de entrada

metric\_data\_features = []

for feature\_name, feature\_value in data.items():

if feature\_name not in ['credit\_score', 'timestamp', 'model\_version']:

metric\_data\_features.append(

{

'MetricName': 'Client Feature',

'Dimensions': [

{

'Name': 'FeatureName',

'Value': feature\_name

},

{

'Name': 'FeatureValue',

'Value': str(feature\_value) # Converter para string para dimensão

}

],

'Value': 1,

'Unit': 'Count'

}

)

if metric\_data\_features:

cloudwatch.put\_metric\_data(

Namespace='Credit Score Features',

MetricData=metric\_data\_features

)

**5.4. Função prepare\_payload (Pré-processamento e Ordem das Features)**

A função prepare\_payload é crucial para garantir que os dados de entrada da API sejam transformados e ordenados exatamente da mesma forma que foram utilizados durante o treinamento do modelo. Isso evita inconsistências e garante a correta execução da inferência.

* **Numéricas**: Inseridas na lista data\_processed na exata ordem usada no treino.
* **Categóricas**: Convertidas para flags booleanas conforme get\_dummies aplicado no pré-processamento (com drop\_first=True).
* **Ordem Crítica**: A ordem das features categóricas (dummies) é mantida fixa utilizando listas ordenadas, e não set, que não preserva a ordem. Isso é vital para que o vetor de entrada seja consistente com o dataset de treino.

import pandas as pd

def prepare\_payload(data):

# Exemplo simplificado de como o payload seria preparado

# A lógica real dependeria do pré-processamento exato do modelo

# Aqui, assumimos que 'data' já vem com as features esperadas

# Definir a ordem esperada das features numéricas e categóricas (dummies)

# Esta lista deve ser idêntica à ordem das colunas no DataFrame de treino

expected\_numeric\_features = [

"idade", "renda\_anual", "salario\_liquido\_mensal", "qtd\_contas\_bancarias",

"qtd\_cartoes\_credito", "taxa\_juros", "qtd\_emprestimos", "dias\_atraso\_pagamento",

"qtd\_pagamentos\_atrasados", "variacao\_limite\_credito", "qtd\_consultas\_credito",

"divida\_pendente", "percentual\_utilizacao\_credito", "total\_emprestimos\_mensal",

"valor\_investido\_mensal", "saldo\_mensal", "tempo\_historico\_credito\_meses"

]

# Ordem fixa das categorias para as dummies (sem as categorias-base)

# ocupacao\_Accountant é a base

expected\_occupations = [

"Architect", "Developer", "Doctor", "Engineer", "Entrepreneur", "Journalist",

"Lawyer", "Manager", "Mechanic", "Media\_Manager", "Musician", "Not Informed",

"Scientist", "Teacher", "Writer"

]

# pagamento\_valor\_minimo\_No é a base

expected\_min\_payment = [

"Not Informed", "Yes"

]

# comportamento\_pagamento\_High\_spent\_Large\_value\_payments é a base

expected\_payment\_behavior = [

"High\_spent\_Medium\_value\_payments", "High\_spent\_Small\_value\_payments",

"Low\_spent\_Large\_value\_payments", "Low\_spent\_Medium\_value\_payments",

"Low\_spent\_Small\_value\_payments"

]

# tipos\_emprestimos\_Auto Loan é a base

expected\_loan\_types = [

"Credit-Builder Loan", "Debt Consolidation Loan", "Home Equity Loan",

"Mortgage Loan", "Not Specified", "Payday Loan", "Personal Loan",

"Student Loan", "Two or More Types of Loan"

]

processed\_data = []

# Adicionar features numéricas na ordem correta

for feature in expected\_numeric\_features:

processed\_data.append(data.get(feature))

# Adicionar features categóricas (dummies) na ordem correta

# Para cada categoria esperada, verifica se o valor da feature original corresponde

# e adiciona 1 ou 0. A categoria base (dropada no treino) resultará em 0 para todas as suas dummies.

# Ocupação

current\_occupation = data.get("ocupacao")

for occ in expected\_occupations:

processed\_data.append(1 if current\_occupation == occ else 0)

# Pagamento Valor Mínimo

current\_min\_payment = data.get("pagamento\_valor\_minimo")

for pvm in expected\_min\_payment:

processed\_data.append(1 if current\_min\_payment == pvm else 0)

# Comportamento Pagamento

current\_payment\_behavior = data.get("comportamento\_pagamento")

for cpg in expected\_payment\_behavior:

processed\_data.append(1 if current\_payment\_behavior == cpg else 0)

# Tipos Empréstimos

current\_loan\_type = data.get("tipos\_emprestimos")

for tpe in expected\_loan\_types:

processed\_data.append(1 if current\_loan\_type == tpe else 0)

# Sanity check: garantir que o vetor de entrada tem o tamanho esperado (17 numéricas + 31 dummies = 48)

assert len(processed\_data) == 48, f"Tamanho do payload processado incorreto: {len(processed\_data)}"

return processed\_data

**5.5. Função handler (Ponto de Entrada do Lambda, Fluxo de Execução)**

A função handler é o ponto de entrada principal para a função AWS Lambda. Ela orquestra todo o fluxo de execução da API, desde o recebimento da requisição até o retorno da resposta.

**Fluxo Geral:**

1. **Recebe o Evento**: O handler recebe o evento (que pode vir do API Gateway ou de uma chamada direta no Lambda).
2. **Extrai e Valida Dados**: Extrai o payload de dados do evento. Valida a presença do campo body para suportar chamadas via API Gateway.
3. **Processa Payload**: Chama prepare\_payload para transformar e ordenar os dados de entrada.
4. **Faz a Predição**: Utiliza o modelo carregado (model.predict) para gerar a previsão de score de crédito.
5. **Registra Métricas**: Invoca input\_metrics para enviar métricas para o CloudWatch.
6. **Salva Requisição e Resposta**: Chama write\_real\_data para persistir os dados de entrada, predição e metadados no S3.
7. **Retorna Resposta**: Constrói e retorna uma resposta JSON padronizada com a predição e a versão do modelo, juntamente com um statusCode e Content-Type apropriados.

**Boas Práticas Aplicadas:**

* **Validação de Entrada**: Verifica a estrutura do evento para garantir compatibilidade com diferentes fontes de invocação.
* **Logs Estruturados**: Registra logs para facilitar o debugging em produção.
* **Retorno Padronizado**: Garante que a resposta da API seja consistente e fácil de consumir.

import json

def handler(event, context):

try:

# Extrai os dados do evento

if 'body' in event: # Chamada via API Gateway

body = json.loads(event['body'])

data = body.get('data')

else: # Chamada direta no Lambda

data = event.get('data')

if not data:

return {

'statusCode': 400,

'headers': { 'Content-Type': 'application/json' },

'body': json.dumps({ 'message': 'Payload de entrada ausente ou inválido.' })

}

# Processa o payload

processed\_input = prepare\_payload(data)

# Faz a predição

prediction = int(model.predict([processed\_input])[0]) # Garante que a predição é um int

# Registra métricas no CloudWatch

input\_metrics(data, prediction)

# Salva a requisição e resposta no S3

write\_real\_data(data, prediction)

# Retorna a resposta

return {

'statusCode': 200,

'headers': { 'Content-Type': 'application/json' },

'body': json.dumps({

'prediction': prediction,

'version': model\_info['version']

})

}

except Exception as e:

print(f"Erro na execução do handler: {e}")

return {

'statusCode': 500,

'headers': { 'Content-Type': 'application/json' },

'body': json.dumps({ 'message': f'Erro interno do servidor: {str(e)}' })

}

**6. Contrato da API**

Esta seção descreve o contrato da API, incluindo o endpoint, método HTTP, estrutura do payload de entrada e formato da resposta. É essencial para que os consumidores da API possam integrá-la corretamente.

**6.1. Endpoint**

O endpoint da API é fornecido pelo Amazon API Gateway após a implantação. Ele terá um formato similar a:

https://<api-gateway-id>.execute-api.<region>.amazonaws.com/<stage>/predict

Onde:

* <api-gateway-id>: ID único gerado pelo API Gateway.
* <region>: Região da AWS onde a API está implantada (ex: us-east-1).
* <stage>: Estágio de implantação da API (ex: production, test).

**6.2. Método HTTP**

A API aceita requisições via método **POST**.

**6.3. Estrutura do Payload de Entrada**

O payload de entrada deve ser um objeto JSON contendo um campo data, que por sua vez é um objeto JSON com as features do cliente. As features devem ser fornecidas com os tipos de dados corretos (numéricos como float/int, categóricos como string).

**Features esperadas:**

* **Numéricas (17)**: idade, renda\_anual, salario\_liquido\_mensal, qtd\_contas\_bancarias, qtd\_cartoes\_credito, taxa\_juros, qtd\_emprestimos, dias\_atraso\_pagamento, qtd\_pagamentos\_atrasados, variacao\_limite\_credito, qtd\_consultas\_credito, divida\_pendente, percentual\_utilizacao\_credito, total\_emprestimos\_mensal, valor\_investido\_mensal, saldo\_mensal, tempo\_historico\_credito\_meses.
* **Categóricas (4)**: ocupacao, pagamento\_valor\_minimo, comportamento\_pagamento, tipos\_emprestimos.

**Exemplo de Payload de Entrada:**

{

"data": {

"idade": 35,

"renda\_anual": 85000.0,

"salario\_liquido\_mensal": 5500.0,

"qtd\_contas\_bancarias": 3,

"qtd\_cartoes\_credito": 2,

"taxa\_juros": 2.5,

"qtd\_emprestimos": 1,

"dias\_atraso\_pagamento": 0,

"qtd\_pagamentos\_atrasados": 0,

"variacao\_limite\_credito": 500.0,

"qtd\_consultas\_credito": 2,

"divida\_pendente": 1500.0,

"percentual\_utilizacao\_credito": 35.5,

"total\_emprestimos\_mensal": 800.0,

"valor\_investido\_mensal": 1200.0,

"saldo\_mensal": 3000.0,

"tempo\_historico\_credito\_meses": 72,

"ocupacao": "Engineer",

"pagamento\_valor\_minimo": "Yes",

"comportamento\_pagamento": "Low\_spent\_Medium\_value\_payments",

"tipos\_emprestimos": "Personal Loan"

}

}

**6.4. Estrutura da Resposta**

A API retornará um objeto JSON com a predição do score de crédito (um inteiro representando a classe de risco) e a versão do modelo utilizada para a inferência.

**Exemplo de Resposta de Sucesso (HTTP 200 OK):**

{

"prediction": 1,

"version": "2"

}

Onde prediction pode ser:

* 0: Poor
* 1: Standard
* 2: Good

**Exemplo de Resposta de Erro (HTTP 400 Bad Request ou HTTP 500 Internal Server Error):**

{

"message": "Payload de entrada ausente ou inválido."

}

**7. Segurança e Governança de Uso**

A segurança e a governança de uso são aspectos fundamentais para proteger a API e garantir seu uso adequado. Esta API implementa mecanismos de autenticação e controle de consumo através do Amazon API Gateway.

**7.1. API Key (Obrigatoriedade e Configuração)**

O acesso à API é protegido por uma API Key, que é configurada como obrigatória no método POST do API Gateway. Isso significa que toda requisição à API deve incluir a chave no cabeçalho x-api-key.

**Exemplo de Requisição com API Key:**

POST /predict HTTP/1.1

Host: <api-gateway-id>.execute-api.<region>.amazonaws.com

x-api-key: sua-api-key-aqui

Content-Type: application/json

{

"data": { ... }

}

Requisições sem a API Key ou com uma chave inválida resultarão em um erro de autorização.

**7.2. Usage Plan (Controle de Consumo)**

Um Plano de Utilização (Usage Plan) chamado Usage Plan Credit Score é criado e associado ao estágio de production da API Credit Score Prediction. Este plano permite:

* **Rate Limits**: Definir o número máximo de requisições por segundo que um cliente pode fazer.
* **Quotas**: Definir o número total de requisições que um cliente pode fazer em um determinado período (ex: por mês).

Isso protege a API contra sobrecarga e garante que os recursos sejam distribuídos de forma justa entre os consumidores.

**7.3. Considerações de Segurança (OIDC, IAM Roles, Secrets)**

Além da API Key para acesso externo, outras medidas de segurança são implementadas ou consideradas:

* **OIDC (OpenID Connect)**: Utilizado para autenticação segura do GitHub Actions com a AWS, eliminando a necessidade de credenciais estáticas (access/secret keys) no pipeline de CI/CD. Uma role githubactions-role é criada no IAM com uma relação de confiança configurada para o provedor OIDC do GitHub.
* **IAM Roles**: A função Lambda e a role de CI/CD (githubactions-role) recebem apenas as permissões mínimas necessárias para executar suas tarefas (princípio do menor privilégio), acessando S3, CloudWatch, ECR, etc.
* **Secrets Management**: Segredos como chaves de API, endpoints e outras informações sensíveis devem ser mantidos fora do código-fonte e gerenciados através de variáveis de ambiente ou serviços de gerenciamento de segredos da AWS (ex: AWS Secrets Manager).

**8. Observabilidade e Auditoria**

A observabilidade e a capacidade de auditoria são cruciais para manter a API e o modelo em produção saudáveis e confiáveis. Esta arquitetura integra o Amazon CloudWatch para monitoramento e o Amazon S3 para auditoria de dados.

**8.1. Métricas no CloudWatch**

O Amazon CloudWatch é configurado para coletar métricas detalhadas da API e do modelo, permitindo um monitoramento proativo e a detecção de anomalias.

* **Métrica "Credit Score Prediction"**: Monitora a distribuição das previsões do modelo, com a dimensão RiskCategory (Poor, Standard, Good) e ModelVersion. Isso permite observar mudanças no perfil de risco dos clientes ao longo do tempo.
* **Métrica "Client Feature"**: Coleta métricas de contagem para cada feature de entrada, com as dimensões FeatureName e FeatureValue. Esta métrica é fundamental para identificar **drift de dados**, ou seja, mudanças na distribuição das features de entrada que podem indicar que o modelo precisa ser re-treinado.

**Alerta de Cardinalidade**: É importante ter cautela ao criar métricas por feature bruta para evitar uma "explosão" de séries temporais únicas no CloudWatch, o que pode gerar custos elevados e dificultar a análise. A estratégia adotada foca em contagens agregadas por valor de feature, o que é mais gerenciável.

**8.2. Logs Estruturados**

A função Lambda gera logs estruturados que são enviados automaticamente para o CloudWatch Logs. Estes logs são essenciais para:

* **Debugging**: Facilitar a identificação e resolução de erros em produção.
* **Análise de Causa Raiz**: Entender o fluxo de execução da API e os eventos que levaram a um determinado comportamento.
* **Auditoria de Eventos**: Registrar eventos importantes, como o início e fim de uma requisição, erros, etc.

**8.3. Auditoria de Dados (S3)**

Conforme detalhado na seção 5.2, a função write\_real\_data persiste cada requisição de inferência (dados de entrada, predição, timestamp e versão do modelo) em um arquivo CSV único no Amazon S3. Este repositório de dados brutos serve como um registro completo para:

* **Auditoria Legal e de Conformidade**: Atender a requisitos regulatórios que exigem o registro de todas as decisões tomadas por sistemas automatizados.
* **Revisão de Desempenho do Modelo**: Analisar o desempenho do modelo em dados reais e identificar casos específicos de falha.
* **Detecção de Drift**: Utilizar esses dados para análises offline de drift de dados e drift de conceito, informando quando o modelo precisa ser atualizado.

**9. CI/CD da API (GitHub Actions e AWS)**

A esteira de Continuous Integration/Continuous Deployment (CI/CD) é fundamental para automatizar o processo de build, teste e deploy da API, garantindo entregas rápidas, consistentes e confiáveis. Esta seção descreve a implementação do CI/CD utilizando GitHub Actions integrado com a AWS.

**9.1. Objetivo e Escopo**

**Objetivo**: Implementar uma esteira de CI/CD para a API que garanta que cada alteração aprovada no código seja automaticamente testada, construída e implantada nos serviços de nuvem da AWS.

**Escopo**: O pipeline de CI/CD abrange o repositório quantum-finance-api-credit-score e interage com os seguintes serviços AWS:

* **AWS Lambda**: Para deploy e atualização da função da API.
* **Amazon API Gateway**: Para atualização da configuração da API e estágios.
* **Amazon ECR**: Para push da imagem Docker containerizada do modelo.
* **Amazon S3**: Para armazenamento de artefatos e modelos (embora o modelo seja baixado do DagsHub em tempo de execução da Lambda, o S3 é usado para outros artefatos e dados de inferência).
* **Amazon CloudWatch**: Para monitoramento e logs.

**9.2. Pré-requisitos**

Para a configuração e execução da esteira de CI/CD, os seguintes pré-requisitos são necessários:

* **Conta AWS**: Com permissões administrativas para criação de roles e policies.
* **Repositório GitHub**: O código da API deve estar versionado no GitHub (quantum-finance-api-credit-score).
* **API Operacional**: A API já deve estar configurada e operacional em Lambda/API Gateway (pelo menos a primeira implantação manual).
* **GitHub Actions Configurado**: O repositório deve ter a estrutura .github/workflows/ para os arquivos de workflow.

**9.3. Configuração do OIDC (OpenID Connect) no IAM**

Para garantir uma autenticação segura e sem credenciais estáticas entre o GitHub Actions e a AWS, é utilizado o OpenID Connect (OIDC). Este método permite que o GitHub Actions assuma uma IAM Role na AWS de forma temporária e segura.

1. **Criar Provedor de Identidade no IAM**: No console do IAM, um novo provedor de identidade do tipo OpenID Connect é criado com as seguintes configurações:
   * **URL**: https://token.actions.githubusercontent.com
   * **Público (Audience)**: sts.amazonaws.com
2. Isso gera um ARN para o provedor, como arn:aws:iam::010928227743:oidc-provider/token.actions.githubusercontent.com.
3. **Associação a uma Role do GitHub Actions**: Uma IAM Role, por exemplo, githubactions-role, é criada e configurada com uma relação de confiança que permite ao provedor OIDC assumi-la. As condições de segurança restringem ainda mais o acesso:
   * aud: sts.amazonaws.com
   * sub: repo:DanielllCouto/\* (restrito ao escopo dos repositórios do usuário DanielllCouto).

**9.4. Relação de Confiança e Permissões da Role**

A relação de confiança configurada para a githubactions-role é crucial para a segurança do pipeline:

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Principal": {

"Federated": "arn:aws:iam::010928227743:oidc-provider/token.actions.githubusercontent.com"

},

"Action": "sts:AssumeRoleWithWebIdentity",

"Condition": {

"StringEquals": {

"token.actions.githubusercontent.com:aud": "sts.amazonaws.com"

},

"StringLike": {

"token.actions.githubusercontent.com:sub": "repo:DanielllCouto/\*"

}

}

}

]

}

Esta role recebe permissões específicas para permitir que o pipeline execute as operações necessárias na AWS, seguindo o princípio do menor privilégio:

* **S3**: s3:PutObject, s3:GetObject, s3:ListBucket (para upload/download de artefatos e dados de inferência).
* **Lambda**: lambda:UpdateFunctionCode, lambda:UpdateFunctionConfiguration, lambda:GetFunction (para deploy e atualização da função da API).
* **API Gateway**: apigateway:PATCH, apigateway:GET, apigateway:PUT (para implantação e atualização da API).
* **CloudWatch**: cloudwatch:PutMetricData, logs:CreateLogGroup, logs:CreateLogStream, logs:PutLogEvents (para envio de métricas e logs).
* **ECR**: ecr:GetLoginPassword, ecr:BatchCheckLayerAvailability, ecr:InitiateLayerUpload, ecr:UploadLayerPart, ecr:CompleteLayerUpload, ecr:PutImage (para push de imagens Dockerizadas).

**9.5. Dockerfile e Containerização para Lambda**

A função Lambda é containerizada usando Docker, o que garante um ambiente de execução consistente e reproduzível. O Dockerfile define a imagem que será construída e enviada para o ECR.

FROM public.ecr.aws/lambda/python:3.10

# Instala dependências de compilação necessárias para alguns pacotes Python

RUN yum install -y gcc gcc-c++ && yum clean all

# Copia o arquivo de dependências e instala

COPY requirements.txt ${LAMBDA\_TASK\_ROOT}

RUN pip install -r requirements.txt

# Copia o código da aplicação e os artefatos do modelo

COPY src/app.py ${LAMBDA\_TASK\_ROOT}

COPY model/model.pkl ${LAMBDA\_TASK\_ROOT}/model/

COPY model/model\_metadata.json ${LAMBDA\_TASK\_ROOT}/model

# Define o handler da função Lambda

CMD [ "app.handler" ]

**9.6. Processo de Build, Tag e Push para ECR**

O pipeline de CI/CD executa os seguintes passos para construir e publicar a imagem Docker no ECR:

1. **Build da Imagem**: Constrói a imagem Docker localmente, especificando a plataforma linux/arm64 para compatibilidade com as arquiteturas de processador da AWS Lambda (Graviton).

docker build --platform linux/arm64 -t quantum-finance/credit-score .

1. **Login no ECR**: Autentica o cliente Docker com o ECR, utilizando um token de login temporário obtido via AWS CLI.

aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 010928227743.dkr.ecr.us-east-1.amazonaws.com

1. **Tag da Imagem**: Marca a imagem local com a tag do repositório ECR e a tag latest (ou uma tag de versão específica).

docker tag quantum-finance/credit-score:latest 010928227743.dkr.ecr.us-east-1.amazonaws.com/quantum-finance/credit-score:latest

1. **Push da Imagem**: Envia a imagem taggeada para o repositório ECR.

docker push 010928227743.dkr.ecr.us-east-1.amazonaws.com/quantum-finance/credit-score:latest

**9.7. Criação e Configuração da Função Lambda**

A função Lambda score-credit-api é criada ou atualizada a partir da imagem Docker hospedada no ECR. As configurações incluem:

* **Nome da Função**: score-credit-api.
* **Imagem de Container**: Referencia a imagem no ECR (quantum-finance/credit-score:latest).
* **Permissões**: Associada a uma IAM Role com as permissões necessárias para acessar S3 e CloudWatch.
* **Variáveis de Ambiente**: Podem ser configuradas para passar informações como o endpoint do MLflow ou o nome do bucket S3.

**9.8. Configuração do API Gateway**

O API Gateway é configurado para expor a função Lambda. Isso envolve:

* **Criação da API REST**: Com o nome Credit Score Prediction.
* **Adição de Método POST**: Integrado à função Lambda score-credit-api usando integração proxy.
* **Implantação em Estágio**: A API é implantada em um estágio (ex: test, production).
* **Configuração de API Key e Usage Plan**: Conforme detalhado na seção 7.

**9.9. Workflow de CI/CD (GitHub Actions)**

O arquivo de workflow do GitHub Actions (.github/workflows/ci-cd.yml) orquestra todo o processo. Um exemplo simplificado de workflow pode incluir:

1. **Trigger**: Em push para a branch main ou em Pull Request.
2. **Checkout do Código**.
3. **Configuração da AWS Credentials**: Utilizando a IAM Role e OIDC.
4. **Build e Push da Imagem Docker para ECR**.
5. **Atualização da Função Lambda**: Apontando para a nova imagem no ECR.
6. **Implantação do API Gateway**: Atualizando o estágio da API.
7. **Testes de Integração**: Após o deploy, executar testes para validar a funcionalidade da API.

**10. Testes**

Uma suíte de testes abrangente é essencial para garantir a qualidade e a confiabilidade da API. Os testes são divididos em testes locais, testes unitários e testes de integração.

**10.1. Testes Locais**

Testes locais são utilizados para validar a lógica da API em um ambiente de desenvolvimento, sem a necessidade de implantar na AWS. Isso agiliza o ciclo de desenvolvimento e depuração.

* **data.json**: Um arquivo JSON contendo um payload de entrada de teste é criado.
* **test.py**: Um script Python que lê o data.json, carrega o app.py e chama o handler localmente, simulando uma invocação da Lambda. Isso permite verificar a resposta da API e a lógica interna sem interagir com os serviços AWS reais.

**10.2. Testes Unitários (pytest)**

Testes unitários são implementados utilizando a framework pytest para validar componentes individuais da API de forma isolada, sem dependências externas (como AWS).

**Estrutura de Pastas:**

tests/

├── conftest.py

└── test\_app.py

* **tests/conftest.py**: Garante que o projeto seja importável nos testes, adicionando o diretório raiz ao sys.path.

# tests/conftest.py

import sys

import os

# Obtém o caminho do diretório pai (projeto/)

projeto\_path = os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..'))

# Adiciona o caminho do diretório pai ao sys.path

sys.path.insert(0, projeto\_path)

* **tests/test\_app.py**: Contém os casos de teste para o app.py.
  + **test\_model\_exists()**: Verifica se o arquivo model.pkl existe.
  + **test\_model\_version\_exists()**: Verifica se o arquivo model\_metadata.json existe.
  + **test\_handler\_call\_local\_without\_aws(monkeypatch)**: Testa o handler localmente, "mockando" as chamadas para S3 e CloudWatch para evitar interações reais com a AWS. Simula uma chamada direta à função Lambda.
  + **test\_handler\_call\_via\_apigw\_body(monkeypatch)**: Testa o handler simulando um evento vindo do API Gateway (com o payload dentro do campo body como string JSON). Também "mocka" as chamadas AWS.

**Execução dos Testes:**

pytest -v

Os testes verificam se a resposta da API tem o statusCode esperado (200), se a prediction é um inteiro e se a version do modelo está presente na resposta.

**10.3. Testes de Integração (API Gateway)**

Após o deploy da API, testes de integração são realizados para validar a funcionalidade completa da API, incluindo a interação com o API Gateway, a função Lambda e os serviços AWS (S3, CloudWatch).

* **Ferramentas**: Podem ser utilizados clientes HTTP como curl, Postman, ou extensões de navegador como Talend API Tester para enviar requisições POST ao endpoint do API Gateway.
* **Verificações**: Confirmar se a resposta JSON está correta, se as métricas são registradas no CloudWatch e se os arquivos de dados de inferência são armazenados no S3.

**11. Melhorias e Próximos Passos**

Esta seção aborda possíveis melhorias e evoluções para a arquitetura da API, visando otimização contínua e adaptação a novas necessidades.

**11.1. Otimizações de Performance**

* **Warm Starts da Lambda**: Para reduzir a latência de "cold starts", pode-se explorar provisioned concurrency para a função Lambda, mantendo instâncias pré-aquecidas.
* **Otimização do Carregamento do Modelo**: Para modelos muito grandes, considerar técnicas de otimização de carregamento ou formatos de modelo mais eficientes.

**11.2. Evolução da Coleta de Dados (Particionamento S3)**

A estratégia atual de um arquivo CSV por requisição no S3 pode gerar um grande número de arquivos em cenários de alto volume. Uma evolução seria implementar um particionamento diário ou horário dos dados no S3, consolidando as inferências em arquivos maiores e mais gerenciáveis para análises posteriores (ex: s3://bucket/credit-score-real-data/year=YYYY/month=MM/day=DD/data.csv).

**11.3. Autenticação (IAM/Cognito)**

Atualmente, a API utiliza API Keys para autenticação. Para cenários que exigem maior granularidade de controle de acesso ou integração com sistemas de identidade corporativos, pode-se evoluir para:

* **AWS IAM**: Utilizar IAM Roles e políticas para autenticar serviços AWS ou aplicações que consomem a API.
* **Amazon Cognito**: Para autenticação de usuários finais em aplicações web ou mobile, integrando com o API Gateway para autorização.

**12. Getting Started / Guia Rápido**

Esta seção oferece um guia rápido para realizar sua primeira requisição à API em menos de 5 minutos. O objetivo é demonstrar a facilidade de uso e a prontidão da API para consumo imediato.

**Pré-requisitos**

Antes de começar, certifique-se de ter:

* Uma **API Key** válida. Esta chave é fornecida após a configuração do API Gateway e é essencial para autenticar suas requisições.
* Um endpoint da API. Este será o URL gerado pelo API Gateway após a implantação da função Lambda (ex: https://<api-gateway-id>.execute-api.<region>.amazonaws.com/<stage>/predict).
* Um cliente HTTP (cURL, Postman, Insomnia, ou um ambiente Python com a biblioteca requests).

**Passo a Passo: Sua Primeira Requisição**

Vamos utilizar um payload de exemplo para prever o score de crédito de um cliente. Substitua YOUR\_API\_KEY pela sua chave real e YOUR\_API\_ENDPOINT pelo URL do seu endpoint.

**Payload de Exemplo:**

{

"data": {

"idade": 30,

"renda\_anual": 75000.0,

"salario\_liquido\_mensal": 4800.0,

"qtd\_contas\_bancarias": 2,

"qtd\_cartoes\_credito": 1,

"taxa\_juros": 3.0,

"qtd\_emprestimos": 0,

"dias\_atraso\_pagamento": 0,

"qtd\_pagamentos\_atrasados": 0,

"variacao\_limite\_credito": 200.0,

"qtd\_consultas\_credito": 1,

"divida\_pendente": 500.0,

"percentual\_utilizacao\_credito": 20.0,

"total\_emprestimos\_mensal": 0.0,

"valor\_investido\_mensal": 1000.0,

"saldo\_mensal": 2500.0,

"tempo\_historico\_credito\_meses": 60,

"ocupacao": "Developer",

"pagamento\_valor\_minimo": "No",

"comportamento\_pagamento": "High\_spent\_Large\_value\_payments",

"tipos\_emprestimos": "Auto Loan"

}

}

**Opção 1: Usando cURL (Terminal)**

Abra seu terminal e execute o seguinte comando, substituindo os placeholders:

curl -X POST \

'YOUR\_API\_ENDPOINT' \

-H 'Content-Type: application/json' \

-H 'x-api-key: YOUR\_API\_KEY' \

-d '{

"data": {

"idade": 30,

"renda\_anual": 75000.0,

"salario\_liquido\_mensal": 4800.0,

"qtd\_contas\_bancarias": 2,

"qtd\_cartoes\_credito": 1,

"taxa\_juros": 3.0,

"qtd\_emprestimos": 0,

"dias\_atraso\_pagamento": 0,

"qtd\_pagamentos\_atrasados": 0,

"variacao\_limite\_credito": 200.0,

"qtd\_consultas\_credito": 1,

"divida\_pendente": 500.0,

"percentual\_utilizacao\_credito": 20.0,

"total\_emprestimos\_mensal": 0.0,

"valor\_investido\_mensal": 1000.0,

"saldo\_mensal": 2500.0,

"tempo\_historico\_credito\_meses": 60,

"ocupacao": "Developer",

"pagamento\_valor\_minimo": "No",

"comportamento\_pagamento": "High\_spent\_Large\_value\_payments",

"tipos\_emprestimos": "Auto Loan"

}

}'

**Opção 2: Usando Python requests**

Crie um arquivo Python (ex: quick\_start.py) e adicione o código abaixo:

import requests

import json

API\_ENDPOINT = 'YOUR\_API\_ENDPOINT'

API\_KEY = 'YOUR\_API\_KEY'

payload = {

"data": {

"idade": 30,

"renda\_anual": 75000.0,

"salario\_liquido\_mensal": 4800.0,

"qtd\_contas\_bancarias": 2,

"qtd\_cartoes\_credito": 1,

"taxa\_juros": 3.0,

"qtd\_emprestimos": 0,

"dias\_atraso\_pagamento": 0,

"qtd\_pagamentos\_atrasados": 0,

"variacao\_limite\_credito": 200.0,

"qtd\_consultas\_credito": 1,

"divida\_pendente": 500.0,

"percentual\_utilizacao\_credito": 20.0,

"total\_emprestimos\_mensal": 0.0,

"valor\_investido\_mensal": 1000.0,

"saldo\_mensal": 2500.0,

"tempo\_historico\_credito\_meses": 60,

"ocupacao": "Developer",

"pagamento\_valor\_minimo": "No",

"comportamento\_pagamento": "High\_spent\_Large\_value\_payments",

"tipos\_emprestimos": "Auto Loan"

}

}

headers = {

'Content-Type': 'application/json',

'x-api-key': API\_KEY

}

try:

response = requests.post(API\_ENDPOINT, headers=headers, data=json.dumps(payload))

response.raise\_for\_status() # Levanta um erro para códigos de status HTTP ruins (4xx ou 5xx)

print("Status Code:", response.status\_code)

print("Response Body:", response.json())

except requests.exceptions.RequestException as e:

print(f"Erro na requisição: {e}")

if response is not None:

print("Response Body (Error):", response.text)

Para executar o código Python, certifique-se de ter a biblioteca requests instalada (pip install requests). Salve o código e execute-o com python quick\_start.py.

**Opção 3: Usando Postman/Insomnia**

1. Crie uma nova requisição POST.
2. Defina o URL como YOUR\_API\_ENDPOINT.
3. Na aba Headers, adicione:
   * Content-Type: application/json
   * x-api-key: YOUR\_API\_KEY
4. Na aba Body, selecione raw e JSON, e cole o payload de exemplo.
5. Envie a requisição.

Em qualquer uma das opções, você deverá receber uma resposta JSON similar a:

{

"prediction": 1,

"version": "2"

}

Isso indica que sua primeira requisição foi bem-sucedida e você recebeu a previsão do score de crédito. A interpretação do valor da prediction pode ser encontrada na seção "Tabela de Interpretação de Resultados".

**13. Exemplos de Consumo em Código**

Esta seção fornece exemplos de como consumir a API utilizando diferentes linguagens e ferramentas, demonstrando a flexibilidade e a facilidade de integração.

**13.1. Python (usando a biblioteca requests)**

O Python é uma escolha comum para interagir com APIs devido à sua simplicidade e à poderosa biblioteca requests. Certifique-se de ter requests instalado (pip install requests).

import requests

import json

# Substitua pelo seu endpoint da API e sua API Key

API\_ENDPOINT = 'YOUR\_API\_ENDPOINT'

API\_KEY = 'YOUR\_API\_KEY'

# Payload de exemplo para a requisição

payload = {

"data": {

"idade": 30,

"renda\_anual": 75000.0,

"salario\_liquido\_mensal": 4800.0,

"qtd\_contas\_bancarias": 2,

"qtd\_cartoes\_credito": 1,

"taxa\_juros": 3.0,

"qtd\_emprestimos": 0,

"dias\_atraso\_pagamento": 0,

"qtd\_pagamentos\_atrasados": 0,

"variacao\_limite\_credito": 200.0,

"qtd\_consultas\_credito": 1,

"divida\_pendente": 500.0,

"percentual\_utilizacao\_credito": 20.0,

"total\_emprestimos\_mensal": 0.0,

"valor\_investido\_mensal": 1000.0,

"saldo\_mensal": 2500.0,

"tempo\_historico\_credito\_meses": 60,

"ocupacao": "Developer",

"pagamento\_valor\_minimo": "No",

"comportamento\_pagamento": "High\_spent\_Large\_value\_payments",

"tipos\_emprestimos": "Auto Loan"

}

}

# Cabeçalhos da requisição, incluindo a API Key

headers = {

'Content-Type': 'application/json',

'x-api-key': API\_KEY

}

try:

# Envia a requisição POST

response = requests.post(API\_ENDPOINT, headers=headers, data=json.dumps(payload))

response.raise\_for\_status() # Levanta um erro para códigos de status HTTP ruins (4xx ou 5xx)

# Imprime a resposta

print("Status Code:", response.status\_code)

print("Response Body:", response.json())

except requests.exceptions.RequestException as e:

print(f"Erro na requisição: {e}")

if response is not None:

print("Response Body (Error):", response.text)

**13.2. cURL (Ferramenta de Linha de Comando)**

cURL é uma ferramenta versátil de linha de comando para transferir dados com sintaxe de URL. É amplamente utilizada para testar APIs.

curl -X POST \

'YOUR\_API\_ENDPOINT' \

-H 'Content-Type: application/json' \

-H 'x-api-key: YOUR\_API\_KEY' \

-d '{

"data": {

"idade": 30,

"renda\_anual": 75000.0,

"salario\_liquido\_mensal": 4800.0,

"qtd\_contas\_bancarias": 2,

"qtd\_cartoes\_credito": 1,

"taxa\_juros": 3.0,

"qtd\_emprestimos": 0,

"dias\_atraso\_pagamento": 0,

"qtd\_pagamentos\_atrasados": 0,

"variacao\_limite\_credito": 200.0,

"qtd\_consultas\_credito": 1,

"divida\_pendente": 500.0,

"percentual\_utilizacao\_credito": 20.0,

"total\_emprestimos\_mensal": 0.0,

"valor\_investido\_mensal": 1000.0,

"saldo\_mensal": 2500.0,

"tempo\_historico\_credito\_meses": 60,

"ocupacao": "Developer",

"pagamento\_valor\_minimo": "No",

"comportamento\_pagamento": "High\_spent\_Large\_value\_payments",

"tipos\_emprestimos": "Auto Loan"

}

}'

**13.3. JavaScript (usando fetch API - Opcional)**

Para aplicações web ou Node.js, a fetch API é uma maneira moderna de fazer requisições HTTP. Este exemplo pode ser executado em um navegador ou ambiente Node.js.

const API\_ENDPOINT = 'YOUR\_API\_ENDPOINT';

const API\_KEY = 'YOUR\_API\_KEY';

const payload = {

data: {

idade: 30,

renda\_anual: 75000.0,

salario\_liquido\_mensal: 4800.0,

qtd\_contas\_bancarias: 2,

qtd\_cartoes\_credito: 1,

taxa\_juros: 3.0,

qtd\_emprestimos: 0,

dias\_atraso\_pagamento: 0,

qtd\_pagamentos\_atrasados: 0,

variacao\_limite\_credito: 200.0,

qtd\_consultas\_credito: 1,

divida\_pendente: 500.0,

percentual\_utilizacao\_credito: 20.0,

total\_emprestimos\_mensal: 0.0,

valor\_investido\_mensal: 1000.0,

saldo\_mensal: 2500.0,

tempo\_historico\_credito\_meses: 60,

ocupacao: "Developer",

pagamento\_valor\_minimo: "No",

comportamento\_pagamento: "High\_spent\_Large\_value\_payments",

tipos\_emprestimos: "Auto Loan"

}

};

fetch(API\_ENDPOINT, {

method: 'POST',

headers: {

'Content-Type': 'application/json',

'x-api-key': API\_KEY

},

body: JSON.stringify(payload)

})

.then(response => {

if (!response.ok) {

throw new Error(`HTTP error! status: ${response.status}`);

}

return response.json();

})

.then(data => {

console.log('Status Code:', response.status);

console.log('Response Body:', data);

})

.catch(error => {

console.error('Erro na requisição:', error);

});

**14. Tabela de Interpretação de Resultados**

A API retorna um valor numérico (prediction) que representa a categoria de risco de crédito. A tabela abaixo detalha a interpretação de cada valor:

| **Prediction** | **Categoria** | **Significado** |
| --- | --- | --- |
| 0 | Poor | Alto risco de inadimplência. |
| 1 | Standard | Risco moderado de inadimplência. |
| 2 | Good | Bom perfil de crédito, baixo risco de inadimplência. |

É importante notar que a prediction é um valor discreto que classifica o cliente em uma das categorias de risco predefinidas pelo modelo de Machine Learning.

**15. Guia de Setup Local**

Este guia detalha os passos necessários para configurar e testar a API localmente. Isso permite que desenvolvedores e cientistas de dados possam inspecionar o código, executar testes e validar a lógica da API sem a necessidade de implantar na AWS, agilizando o ciclo de desenvolvimento e depuração.

**Pré-requisitos**

* **Git**: Para clonar o repositório.
* **Python 3.10+**: A versão do Python deve ser compatível com as bibliotecas utilizadas no projeto.
* **pip**: Gerenciador de pacotes Python.

**Passo a Passo para Configuração Local**

Siga os passos abaixo para ter o ambiente da API funcionando em sua máquina local:

1. **Clonar o Repositório**: Comece clonando o repositório da API do GitHub. Certifique-se de que o nome do repositório é quantum-finance-api-credit-score.

git clone https://github.com/DanielllCouto/quantum-finance-api-credit-score.git

cd quantum-finance-api-credit-score

1. **Criar e Ativar o Ambiente Virtual**: É uma boa prática criar um ambiente virtual para isolar as dependências do projeto.

python3.10 -m venv .venv

source .venv/bin/activate

1. **Instalar Dependências**: Instale todas as bibliotecas necessárias listadas no requirements.txt.

pip install -r requirements.txt

1. **Baixar o Modelo de Machine Learning**: O modelo e seus metadados são baixados do MLflow/DagsHub usando o script model\_downloader.py. Este script criará a pasta model/ e colocará os arquivos model.pkl e model\_metadata.json nela.

python model/model\_downloader.py

1. *Nota: Certifique-se de que suas credenciais ou configurações para acessar o DagsHub/MLflow estejam corretas, caso contrário, o download do modelo pode falhar.*
2. **Executar Testes Unitários**: Para verificar se tudo está configurado corretamente e se a lógica da API funciona como esperado, execute a suíte de testes unitários com pytest.

pytest -v

1. Você deverá ver uma saída indicando que todos os testes foram aprovados, confirmando que o ambiente local está funcional e que a lógica central da API está correta.

Após seguir esses passos, você terá um ambiente local pronto para desenvolvimento e testes da API, sem a necessidade de uma implantação completa na AWS. Isso é particularmente útil para iterar rapidamente sobre novas funcionalidades ou depurar problemas.

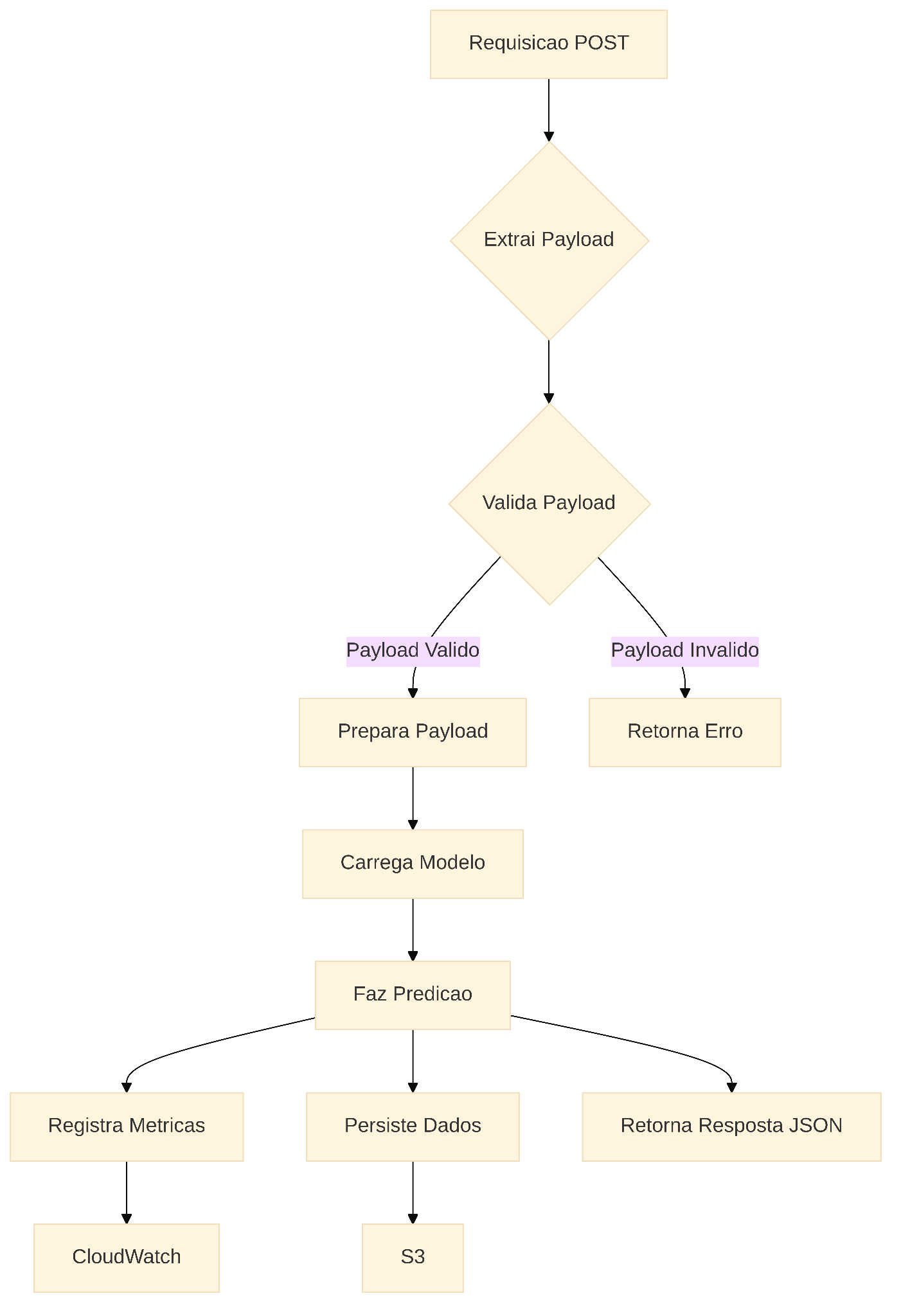
**16. Checklist de Boas Práticas**

Esta seção resume as principais boas práticas de MLOps e desenvolvimento de software aplicadas na construção e operação desta API, destacando os pilares de um projeto robusto e de alta qualidade.

* ✅ **Versionamento de Modelos (MLflow/DagsHub)**: Todos os modelos são versionados e rastreados, garantindo reprodutibilidade e facilitando a gestão do ciclo de vida do modelo.
* ✅ **Observabilidade (CloudWatch)**: Métricas detalhadas e logs estruturados são coletados para monitorar o desempenho da API e do modelo em tempo real, permitindo detecção proativa de problemas.
* ✅ **Auditoria de Dados (S3)**: Cada inferência é registrada no Amazon S3, criando um histórico completo para auditoria, análise de drift e re-treinamento futuro.
* ✅ **Testes (pytest)**: Uma suíte de testes unitários e de integração garante a correção da lógica da API e a robustez do sistema.
* ✅ **Segurança (API Key + Usage Plan)**: O acesso à API é controlado por API Keys e gerenciado por Usage Plans no API Gateway, protegendo contra acessos não autorizados e abusos.
* ✅ **CI/CD (GitHub Actions + OIDC)**: Uma esteira de integração e entrega contínua automatiza o deploy, garantindo consistência, velocidade e segurança nas atualizações, utilizando OIDC para autenticação sem credenciais estáticas.
* ✅ **Containerização (Docker/ECR)**: A API é empacotada em containers Docker e armazenada no ECR, assegurando um ambiente de execução consistente e portátil.
* ✅ **Arquitetura Serverless (AWS Lambda)**: Utilização de serviços serverless para escalabilidade automática, alta disponibilidade e otimização de custos.
* ✅ **Isolamento de Código**: O código da API é mantido em um repositório separado do projeto de modelagem, promovendo modularidade e reduzindo dependências.
* ✅ **Payload Consistente**: A função prepare\_payload garante que os dados de entrada sejam processados e ordenados de forma idêntica ao treinamento do modelo, evitando inconsistências na inferência.

**17. Exemplo de Fluxo Visual de Requisição e Resposta**

Para ilustrar o fluxo de uma requisição através da API, o diagrama abaixo representa a sequência de eventos desde o cliente até a resposta final, incluindo as interações com os serviços AWS.



*Figura 2: Fluxo de Requisição e Resposta da API de Previsão de Score de Crédito.*