UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Daniel Franzoni

Migração para nuvem de ferramenta de auxílio à tomada de decisão em operações de compra e venda de ativos utilizando Fast API e Airflow

Florianópolis
2023

Daniel Franzoni

Migração para nuvem de ferramenta de auxílio à tomada de decisão em operações de compra e venda de ativos utilizando Fast API e Airflow

Relatório final da disciplina DAS5511 (Projeto de Fim de Curso) como Trabalho de Conclusão do Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Santa Catarina em Florianópolis.
Orientador: Prof. Rodrigo Castelan Carlson, Dr.
Supervisor: Marcelo Nogueira Araujo

Florianópolis
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Franzoni, Daniel
Migração para nuvem de ferramenta de auxílio à tomada de decisão em operações de compra e venda de ativos utilizando Fast API e Airflow / Daniel Franzoni ; orientador, Rodrigo Castelan Carlson, 2023.
55 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia de Controle e Automação, Florianópolis, 2023.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Migração para nuvem. I. Carlson, Rodrigo Castelan. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Daniel Franzoni

Migração para nuvem de ferramenta de auxílio à tomada de decisão em operações de compra e venda de ativos utilizando Fast API e Airflow

Esta monografia foi julgada no contexto da disciplina DAS5511 (Projeto de Fim de Curso) e aprovada em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação

Florianópolis, 11 de Dezembro de 2023.

Prof. Marcelo de Lellis Costa de Oliveira, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Rodrigo Castelan Carlson, Dr.
Orientador
UFSC/CTC/DAS

Documento assinado digitalmente
Rodrigo Castelan Carlson
Data: 18/12/2023 18:39:53-0300
CPF: ***.743.969-**
Verifique as assinaturas em https://v.ufsc.br

Marcelo Nogueira Araújo
Supervisor
Radix Engenharia e Software

Documento assinado digitalmente
MARCELO NOGUEIRA ARAUJO
Data: 18/12/2023 14:22:10-0300
Verifique em https://validar.iti.gov.br

Publio Macedo Monteiro Lima, Dr.
Avaliador
UFSC/CTC/DAS

Prof. Eduardo Camponogara, Dr.
Presidente da Banca
UFSC/CTC/DAS

AGRADECIMENTOS

Agradeço à minha família, especialmente aos meus pais Eliane e Lorenzo, por todo amor e apoio que me deram ao longo dessa jornada acadêmica. E a minha namorada, Maria Fernanda, que foi muito importante nesta última etapa da graduação.

Agradeço também ao meu orientador, professor Rodrigo Carlson, que me direcionou para conclusão deste trabalho.

E por último, um agradecimento ao meu supervisor na empresa, Marcelo Araújo, pelas oportunidades no processo de formação profissional e zelo pelo bem estar.

DECLARAÇÃO DE PUBLICIDADE

Rio de Janeiro, 11 de Dezembro de 2023.

Na condição de representante da Radix Engenharia e Software na qual o presente trabalho foi realizado, declaro não haver ressalvas quanto ao aspecto de sigilo ou propriedade intelectual sobre as informações contidas neste documento, que impeçam a sua publicação por parte da Universidade Federal de Santa Catarina (UFSC) para acesso pelo público em geral, incluindo a sua disponibilização *online* no Repositório Institucional da Biblioteca Universitária da UFSC. Além disso, declaro ciência de que o autor, na condição de estudante da UFSC, é obrigado a depositar este documento, por se tratar de um Trabalho de Conclusão de Curso, no referido Repositório Institucional, em atendimento à Resolução Normativa n° 126/2019/CUn.

Por estar de acordo com esses termos, subscrevo-me abaixo.

gov.br
Documento assinado digitalmente
MARCELO NOGUEIRA ARAUJO
Data: 30/11/2023 11:30:49-0300
Verifique em https://validar.iti.gov.br

Marcelo Nogueira Araújo
Radix Engenharia e Software

RESUMO

Este trabalho trata da migração de uma ferramenta de auxílio à tomada de decisão em operações de compra e venda de ativos, que fornece informações baseadas em dados concretos e simulações personalizadas. A migração para nuvem da ferramenta é vista como uma ação necessária para o crescimento sustentável e escalável da ferramenta. Isto por que a usabilidade atrapalha o dia a dia dos usuários finais, além de não apresentar mais performance desejável e possuir alta complexidade no desenvolvimento de melhorias. Este movimento vem sendo uma prática muito adotada por empresas com o crescimento do uso de dados, desse modo ganhando fácil escalabilidade quando o volume de dados utilizado aumenta também. Com isto, novas ferramentas e técnicas para gerenciar os processos em nuvem foram surgindo e deixando cada vez mais vantajoso o uso da nuvem que proporciona mais performance e menor custo para executar as mesmas tarefas. Neste trabalho, serão abordadas algumas destas técnicas e estruturas que foram utilizadas a fim de alcançar um melhor desempenho da ferramenta e centralizar os processamentos dos dados proporcionando uma melhor governança. Desta forma, a migração garante que a ferramenta não caia em desuso e proporcione mais agilidade no trabalho do usuário. Além de proporcionar novas funcionalidades que outrora não seriam possíveis.

Palavras-chave: Migração. Nuvem. Ferramenta.

ABSTRACT

This work deals with the migration of a decision support tool in buying and selling operations of assets, which provides information based on concrete data and customized simulations. The migration of the tool to the cloud is seen as a necessary action for the sustainable and scalable growth of the tool. This is because usability hinders the day-to-day activities of end users, in addition to not presenting the desired performance and having high complexity in the development of improvements. This move has become a widely adopted practice by companies with the growth of data usage, thus gaining easy scalability when the volume of data used also increases. With this, new tools and techniques to manage processes in the cloud have been emerging, making the use of the cloud increasingly advantageous, providing more performance and lower costs to perform the same tasks. In this work, some of these techniques and structures that were used to achieve better performance of the tool and centralize data processing, providing better governance, will be addressed. Thus, migration ensures that the tool does not become obsolete and provides more agility in the user's work. In addition, it offers new functionalities that were not possible before.

Keywords: Migration. Cloud. Tool.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Atualmente vivemos na era dos dados, onde informação e entendimento vindo dos dados são formas de ganhar dinheiro. Para que seja possível utilizá-los é necessário montar uma infraestrutura que possibilite explorar este potencial dos dados e analisá-los. Esta infraestrutura pode ser construída transferindo aplicativos, cargas de trabalho e dados para um ambiente de computação em nuvem.

A migração de ferramentas e processamentos para a nuvem vem sendo uma prática muito adotada por empresas com o crescimento do uso de dados, desse modo ganhando fácil escalabilidade quando o volume de dados utilizado aumenta também. Com este movimento novas ferramentas e técnicas, para gerenciar os processos em nuvem, foram surgindo e deixando cada vez mais vantajoso o uso da nuvem que proporciona mais performance e menor custo para executar as mesmas tarefas. Além disso, a nuvem possibilita automatizar, otimizar, escalabilizar e paralelizar os processos de forma mais dinâmica e sem se preocupar com a infraestrutura física.

Desta forma, o projeto de migração para nuvem da ferramenta de auxílio à tomada de decisão em operações de compra e venda de ativos apresentado neste documento visa melhorar a ferramenta tanto para o usuário final como para a governança do cliente realizando a migração dos processamentos para a nuvem. Esta ferramenta ampara os usuários de forma a permitir uma análise melhor das oportunidades do mercado, baseando-se em dados concretos e simulações personalizadas para cenários de contratos de compra e venda.

Com o crescimento contínuo da ferramenta, surgiram alguns obstáculos. O volume de cálculos e dados aumentou e tornou o processamento local oneroso, assim dificultando a sua utilização e atrapalhando o dia a dia dos usuários finais. Além disso, a execução dos processamentos locais implica na dificuldade em prover suporte ao usuário pois gera erros distintos para cada usuário, necessidade de realizar uma nova instalação quando se tem uma nova versão da ferramenta, performance distinta para cada computador de usuário e alta complexidade no desenvolvimento de novas funcionalidades e melhorias.

A migração para nuvem da ferramenta é vista como uma ação necessária para o crescimento sustentável e escalável da ferramenta, desta forma garantindo que ela não caia em desuso e proporcionando mais agilidade no trabalho do usuário.

Para alcançar os resultados esperados de forma eficaz e com melhoria contínua dos processos de trabalho da equipe, realizou-se o projeto tendo como base uma metodologia ágil, que será abordada posteriormente. A gestão da evolução do projeto e a sequência de atividades a serem realizadas eram sempre discutidas entre equipe e cliente, para encontrar as melhores soluções e evoluir a ferramenta. Assim todos os resultados e decisões eram compartilhados entre todos.

Neste documento, será realizada uma breve apresentação da empresa no qual foi realizado o projeto de fim de curso e serão apresentadas as pricipais metodologias e tecnologias utilizadas para o desenvolvimento deste projeto. Após estas etapas, serão expostos em um capítulo o estudo realizado sobre o funcionamento da ferramenta antes da migração e o planejamento do projeto de migração. Então por último, serão apresentadas as 3 etapas da migração e os seus resultados. As etapas de migração são a automatização de processos utilizando o Airflow, a migração de processos que executam em tempo real para uma API e a integração da nova arquitetura com o *Front-end* e a coleta de métricas de uso da ferramenta.

1.1 OBJETIVOS

O objetivo geral do projeto é com a migração para a nuvem dos processamentos da ferramenta tornar o crescimento da ferramenta sustentável e escalável, diminuir o tempo de processamento e melhorar a governança.

Os objetivos específicos são:

- Centralizar os processamentos de dados;
- Melhorar governança de dados e da ferramenta;
- Melhorar o suporte ao usuário;
- Eliminar a necessidade de instalação nos computadores dos usuários;
- Equalizar as performances de execução dos processamentos para todos os usuários;
- Desenvolver coleta de métricas de utilização e erros;
- Transformar processos manuais em automatizados quando possível;
- Tornar os recursos dos processamentos escaláveis.

1.2 A EMPRESA

A Radix é uma empresa multi-nacional de engenharia e software que presta consultoria e incentiva a transformação digital dos seus clientes.

Para alcançar o desenvolvimento com qualidade das mais diferentes soluções, a empresa conta com um amplo leque de parceiros tecnológicos e uma equipe multidisciplinar que garantem o atendimento aos clientes nas mais variadas frentes.

Desde 2010, quando foi fundada, é reconhecida como uma das melhores empresas para se trabalhar no Brasil e na América Latina pelo Great Place to Work.

2 METODOLOGIA E TECNOLOGIAS

2.1 METODOLOGIA

O gerenciamento do projeto que executou a migração foi realizado utilizando-se a metodologia *Scrum*. O *Scrum* é muito utilizado em desenvolvimento de *software* e proporciona uma gestão dinâmica de projetos de forma ágil, iterativa e incremental.

O termo Scrum vem do rúgbi e se refere à maneira como um time se une para avançar com a bola pelo campo. Tudo se alinha: posicionamento cuidadoso, unidade de propósito e clareza de objetivo. Trata-se de uma analogia perfeita para o que se espera das equipes (SUTHERLAND; SUTHERLAND, 2014).

Este método possui 3 pilares centrais. O primeiro é a transparência dos processos, dos requisitos de entrega e *status* de desenvolvimento. Tudo que acontece no projeto deve estar transparente e alinhado com todos os responsáveis pelos resultados. O segundo é a inspeção em todas as etapas de tudo o que está sendo desenvolvido. E o terceiro é a adaptação do processo e do produto, que podem necessitar ajustes durante o projeto para buscar melhores resultados.

No *Scrum* existem 3 papéis fundamentais o *Scrum Master* (SM), *Product Owner* (PO) e o Time. O SM é responsável por auxiliar o time a entender e manter os princípios e as práticas do *Scrum* no dia-a-dia, além de ter o papel de facilitador para o trabalho removendo impedimentos e refinando itens da próxima *Sprint* junto ao PO. O *Product Owner* é responsável no desenvolvimento do produto pelas decisões de quais recursos serão desenvolvidos e qual a ordem de prioridade de desenvolvimento. O Time é a equipe que desenvolve o projeto ou produto. A equipe é quem define como as coisas serão feitas e quantas tarefas são possíveis de realizar em uma *Sprint* para atingir as metas estabelecidas pelo PO.

O *Scrum* é executado em blocos temporários curtos e periódicos chamados de *Sprints*, que normalmente variam de 2 a 4 semanas e ao final são realizados *feedbacks* do usuário ou cliente final para fazer ajustes caso necessário. Para cada *Sprint* são priorizados entregáveis para serem desenvolvidos e entregues pelo time naquele período.

Para o bom andamento do projeto e do método *Scrum* são realizadas cerimônias ao decorrer das *Sprints* para alinhamentos e tomada de decisões, são elas:

- *Sprint Planning*: reunião que antecede o início de cada *Sprint* para decidir o que será desenvolvido durante a *Sprint* e quais entreguas serão feitas.
- *Daily*: reunião diária, com duração de 15 min, onde são transparecidos a todos o que foi feito no dia anterior; o que será feito no dia; se existe algum bloqueio para o andamento das atividades.

- *Sprint Review*: reunião realizada após o encerramento de cada *Sprint* para validar e adaptar o desenvolvimento do produto. Nessa reunião é onde o PO propõe alterações para serem realizadas na *Sprint* seguinte ou no futuro.
- *Retrospectiva*: reunião realizada ao final de cada *Sprint* apenas com o Time, na qual todos podem expor sua opinião de como foi o trabalho realizado durante a *Sprint*. Este é o momento onde é verificada a necessidade de adaptação do processo a partir do que aconteceu de positivo e negativo.

2.2 TECNOLOGIAS

2.2.1 Microsoft Azure

O Microsoft Azure é uma plataforma de núvem pública administrada pela Microsoft. Ele oferece uma gama de serviços como plataforma como serviço, infraestrutura como serviço, banco de dados como serviço e entre outros.

O Azure, como outras plataformas de nuvem, se baseia em uma tecnologia conhecida como virtualização. A maioria dos hardwares de computador pode ser emulada no software. O hardware do computador é simplesmente um conjunto de instruções codificadas no silício de forma permanente ou semipermanente. As camadas de emulação são usadas para mapear instruções de software para instruções de hardware. As camadas de emulação permitem que o hardware virtualizado seja executado em software como o próprio hardware (EKUAN; ZIMMERGREN; MOORE; PARKER; BUCK; COULTER, 2023)

2.2.1.1 Microsoft Azure Data Lake Storage Gen2

Data lake é um repositório centralizado que permite armazenar e proteger grandes volumes de dados em sua forma original. Por possuir uma arquitetura aberta e escalonável, um data lake permite qualquer tipo de dado de qualquer fonte, desde dados estruturados até semiestruturados e não estruturados.

O Azure Data Lake Storage Gen2 é um repositório de dados baseado em nuvem construído sobre o Blob Storage. O Blob Storage oferece recursos como alta disponibilidade e gerenciamento do ciclo de vida com baixo custo. E o Data Lake Storage oferece armazenamento hierárquico, segurança robusta, recuperação após desastres, semântica de sistema de arquivos e compatibilidade com Hadoop.

2.2.1.2 Azure Synapse Analytics

O Azure Synapse é um serviço de disponibilização e análise de dados empresarial e ilimitado. Ele fornece uma plataforma que reúne o melhor da tecnologia SQL

usada em *Data Warehouse* corporativo moderno, além de possibilitar análise de grandes volumes de dados com tecnologia Spark. O Synapse proporciona a liberdade de consultar dados de acordo com os termos desejados, usando recursos *server-less* ou provisionados em escala. O Azure Synapse oferece uma experiência unificada para ingerir, preparar, gerenciar e servir dados para necessidades imediatas da área de negócio.

2.2.2 Kubernetes

O Kubernetes é um sistema de código aberto de orquestração de aplicativos em contêineres que facilita automatizar, dimensionar e gerenciar a implantação destes aplicativos em grande escala.

Os contêineres são pacotes de software que contêm todas as dependências necessárias para que os aplicativos sejam abstraídos pelos ambientes que os executam. Desse modo, os contêineres permitem que aplicativos sejam implementados em qualquer lugar de forma rápida e consistente.

Os Pods são as menores unidades de computação possíveis de gerenciar e criar dentro do Kubernetes. Eles podem ser um ou mais contêineres agrupados que compartilham recursos de armazenamento e rede, além de uma especificação de como executar os contêineres. O conteúdo de um pod é alocado e agendado conjuntamente e executado em um contexto compartilhado.

2.2.3 Docker e Docker Compose

Docker é uma plataforma aberta para desenvolver e executar aplicações, que permite isolar a aplicação da infraestrutura e encurtar o tempo de desenvolvimento até a etapa de produtização. Ele tem a capacidade de empacotar e executar um aplicativo em um contêiner a partir das imagens. Uma imagem é um modelo com instruções para criar um contêiner, geralmente é baseada em outra imagem já existente com customizações para atender as instruções do modelo.

Docker Compose é uma ferramenta para configurar e executar aplicações Docker com múltiplos contêineres. A partir de um arquivo YAML são realizadas as configurações dos serviços da aplicação, e então com um comando o Compose gerencia a criação e inicialização dos serviços.

2.2.4 Apache Spark e Apache Airflow

Esta estrutura de código aberto é amplamente utilizada para processamento e análise de dados. O Apache Spark é conhecido por ser eficiente em trabalhar com grandes volumes de dados. Por ser altamente escalável, ele possibilita a distribuição do processamento em diversos computadores. Dentre suas vantagens notáveis, está

a existência de um conjunto variado de tarefas de processamento de dados, acessibilidade para diversas linguagens de programação, fácil usabilidade e disponibilidade de diversas bibliotecas de alto nível.

O Apache Airflow é uma plataforma de código aberto criada para desenvolvimento e monitoramento de fluxos de trabalho que tem sido amplamente utilizada no universo da engenharia de dados.

Esta ferramenta tem a propriedade de definir, programar e monitorar fluxos de trabalhos complexos chamados de DAGs. DAG é uma sigla para *Directed Acyclic Graphs* (Grafos Acíclicos Dirigidos). Isto significa que ao criar um DAG é gerada uma coleção de tarefas organizadas que se deseja programar e executar em uma ordem escolhida.

O Airflow foi desenvolvido para lidar com dependências entre tarefas e para gerenciar falhas através de mecanismos de retentativa e notificações. Estas tarefas são definidas em Python, sendo flexíveis e acessíveis para desenvolvimento. Também existe uma interface na web que permite a visualização e gestão dos fluxos de trabalho, o que facilita o monitoramento e resolução de problemas.

2.2.5 Memcached

A Memcached é um sistema de armazenamento de objetos de memória distribuída de código aberto de alto desempenho e de natureza genérica. É um armazamento de valores-chave na memória para pequenos pedaços de dados como strings e objetos. Este sistema é destinado a aplicação de aplicativos da web dinâmicos. Sua estrutura permite implantação rápida e facilidade de desenvolvimento.

2.2.6 WSL

O Subsistema do Windows para Linux (WSL) é um recurso do sistema operacional Windows que permite executar um sistema de arquivos linux, juntamente com ferramentas de linha de comando do Linux e aplicativos gui, diretamente no Windows, juntamente com sua área de trabalho e aplicativos tradicionais do Windows (LOEWEN; BUCK; WOJCIAKOWSKI; JUNKER; LEMOINE, 2023).

2.2.7 Fast API

Fast API é uma estrutura web moderna e de alta performance para construção de APIs utilizando a linguagem Python. Ela permite processar requisições em paralelo de forma nativa e o modo de programar utilizando esta ferramenta possibilita maior agilidade no desenvolvimento.

API é uma sigla para *Application Programming Interface* que é traduzida para a língua portuguesa como interface de programação de aplicação. A API conecta

diversos programas e plataformas permitindo que estes se comuniquem trocando informações, através de normas, padrões e protocolos.

3 ESTUDO SOBRE A FERRAMENTA E PLANEJAMENTO DA MIGRAÇÃO

O projeto de migração para nuvem da ferramenta de auxílio à tomada de decisão em operações de compra e venda de ativos foi realizado a fim de atender uma demanda prevista inicialmente pelo time de desenvolvedores desta ferramenta, que foram responsáveis por desenvolver toda a lógica e as regras de negócio traduzidas em cálculos e no código da primeira versão desta. Esta demanda fazia parte do plano de ação da construção da ferramenta e tinha o intuito de manter a evolução dela sustentável e escalável, já que a primeira versão concebida era executada localmente no computador de cada usuário sendo limitada por este ambiente.

Com o amadurecimento da ferramenta, no qual ela provou seu valor e teve crescimento de suas funcionalidades, o time de negócio e os usuários indentificaram a necessidade de realizar o próximo passo previsto, que é a migração para a nuvem. No momento em que foi iniciada a migração, já era notado que a usabilidade e a performance estavam prejudicadas pela execução descentralizada nas máquinas locais. Assim como também ocorria com o suporte ao usuário que era demorado e desgastante, pois não se tinha acesso físico às máquinas e cada uma apresentava erros e performances distintas.

Este projeto visou obter como resultado migrar toda a parte de processamento de dados da ferramenta, que é realizada por código em Python, para um ambiente de computação em nuvem utilizando a plataforma Microsoft Azure - já difundida dentro da empresa. Mas como a ferramenta estava em uso e constante crescimento, foi necessário realizar a migração de forma a possibilitar a evolução da versão local sem que ela fosse comprometida pelas modificações para a execução em nuvem.

O início do projeto se deu pelas reuniões realizadas com o time de desenvolvedores e o estudo da documentação da versão local da ferramenta. Neste momento, obteve-se acesso aos códigos Python dos processos e ao Excel utilizado como *Front-end*, com isto foi possível entender como eram chamados e iniciados os processos através de comandos *Shell Script* no Excel. Foi feito todo o processo de instalação da ferramenta com acompanhamento dos desenvolvedores explicando cada parte e o que necessitava no ambiente local do usuário para o funcionamento da mesma. Além disso, foram realizadas reuniões para esclarecimento de dúvidas sobre a documentação técnica, o modo de uso dos usuários e sobre a evolução da ferramenta.

Para que um usuário pudesse utilizar a ferramenta em sua versão local, era necessário que sua máquina possuísse o Python e o Virtualenv instalados pelo time de TI, dessa forma era possível realizar a instalação da ferramenta. Com estes dois requisitos cumpridos o usuário realizava o *download* de um arquivo comprimido, que era disponibilizado no SharePoint do cliente e atualizado a cada versão nova. Neste arquivo eram encontrados um arquivo de instalação, o arquivo do Excel utilizado como

Front-end e os códigos Python com dependências que constituíam a ferramenta. Após a descompactação do arquivo comprimido era possível editar as informações necessárias no arquivo de instalação e realizar a instalação ao executá-lo. Por fim, era preciso abrir o arquivo do *Front-end* e editar uma célula do Excel que indicava à ferramenta onde estavam os dados locais sobre informações de mercado espelhados do SharePoint do cliente. Estes dados de mercado são utilizados para gerar uma base de dados que é a base de todos os cálculos realizados nos processos da ferramenta.

A ferramenta permite ao usuário modificar parâmetros e utilizá-los para calcular cenários junto às informações da base de dados ao apertar o botão da calculadora que deseja executar no *Front-end*. Para isto acontecer, os botões que compõe as visualizações em diferentes abas do Excel executam Macros que transformam os parâmetros presentes em cada aba em arquivos utilizados de entrada para os processos em Python. Após a transformação, através de comandos *Shell Script* no VBA os processos em Python são inicializados e são gerados os dados da calculadora desejada que são salvos em pastas ao lado do arquivo do *Front-end*. Com os dados da calculadora gerados, estes dados são consumidos por tabelas dentro do Excel que são atualizadas por instruções na execução da Macro logo após o término da execução dos comandos *Shell Script*. Por fim, as visualizações das abas do Excel consomem os dados gerados pelas calculadores através das tabelas e mostram ao usuário os resultados solicitados sob demanda.

A base de dados utilizada em todos os cálculos realizados nos processos da ferramenta precisava ser atualizada diariamente pelo usuário. Esta atualização era possível através de uma Macro executada por um botão que, por meio de comandos *Shell Script* no VBA, também inicializa os processos em Python e disponibiliza a base de dados em uma pasta ao lado do arquivo do Excel.

3.1 ARQUITETURA PROPOSTA PELA MIGRAÇÃO

Após o entendimento do funcionamento, desenvolvimento e utilização da versão local da ferramenta, foi estruturado um plano de migração a partir do levantamento de requisitos e objetivos que desejavam ser atingidos.

No levantamento de requisitos entendeu-se que o *Front-end* da ferramenta continuaria sendo o Excel após o término da migração. Então ela ainda deveria suportar o Excel, mas também permitir migrar o *Front-end* para outra ferramenta de análise e visualização de dados. As calculadoras precisariam ser executadas e disponibilizar os resultados em tempo real, desse modo mantendo a dinâmica do ambiente local. E os processos que não são executados com entradas de usuário e podem executar sob agendamento podem ser automatizados.

A partir dos requisitos citados anteriormente, pode-se determinar outros requisitos mais focados em tecnologias e desenvolvimento da migração como todos os dados

gerados pela ferramenta executada em nuvem devem ser armazenados no Azure Data Lake Storage Gen2 do cliente. Os dados armazenados no Data Lake que precisam ser consumidos pelo Excel são lidos através de *views* do Azure Synapse Analytics utilizando-se o Power Query dentro do Excel. Os processos executados por agendamento devem ser migrados para o Airflow e os processos executados em tempo real devem ser migrados para uma API que precisa ser desenvolvida em Python.

Com o intuido de atender aos requisitos de utilizar o Data Lake para centralizar o armazenamento de todos os dados gerados pela ferramenta e utilizar o Synapse para consumir eles, foi proposta uma estruturação com partições e transformação dos dados gerados para salvá-los no formato Parquet. Esta proposta está ilustrada na Figura 1. Com os dados em Parquet, que é um formato colunar e mais eficiente de se armazenar dados, pode-se aproveitar da capacidade de organizar os dados em partições. As partições são colunas do dado, geralmente com baixa variedade de dados e alta repetição de valores, que pode-se transformar em uma forma de dividir o dado criando-se pastas no padrão `<nome_coluna>=<valor_registro>` e armazenando as linhas que tem o mesmo valor de registro para aquela coluna dentro desta pasta. As partições criadas na migração são no padrão mostrado pelas pastas vermelhas apresentadas na Figura 1.

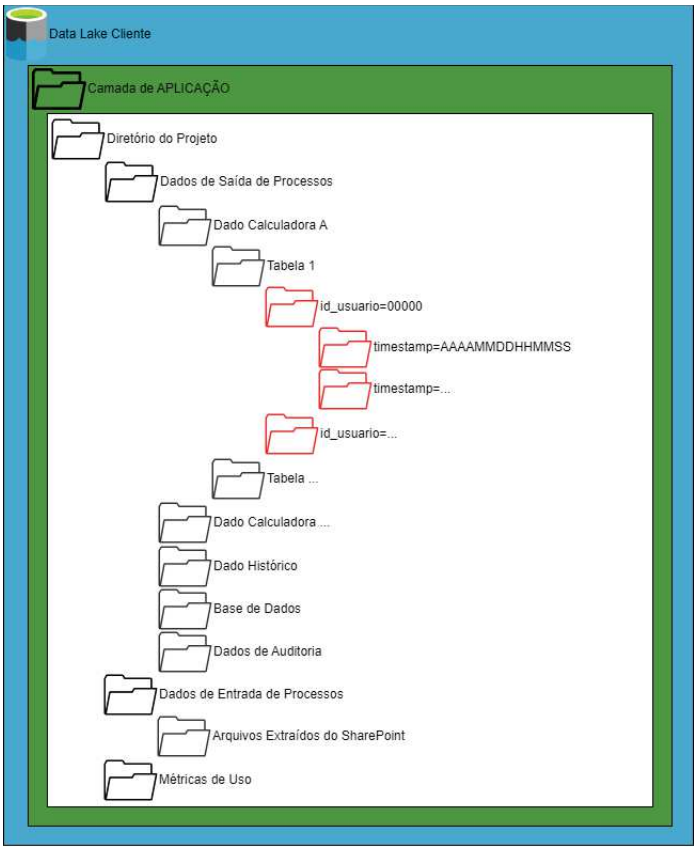
A ferramenta não utilizava o Parquet como padrão de arquivo e nem organizava em partições, mas armazenava todos os dados de saída dos processos no formato CSV ou JSON no local de execução. Para que fosse possível ter os dados em Parquet, foi utilizado o Apache Spark que é capaz de ler os dados em CSV e JSON e salvá-los em Parquet particionado no Data Lake de forma nativa.

A estrutura hierárquica de pastas implementada no Azure Data Lake Storage Gen2 do cliente é ilustrada pela Figura 1. Todos os dados gerados pela ferramenta após a migração são armazenados dentro desta estrutura e organizados em dados de Saída e dados de Entrada. Os dados de saída são subdivididos em diretórios com o nome da respectiva calculadora que gerou os dados e dentro deles tem-se subdiretórios, nomeados na ilustração da Figura 1 no padrão `Tabela <número>`, que correspondem ao arquivo gerado pelo processo transformado em Parquet.

Com os dados salvos em Parquet no Data Lake pode-se então criar as *views* do Synapse. Estas consomem os dados das Tabelas abstraindo o formato do dado para que seja possível consultar e filtrar os registros utilizando a linguagem SQL de qualquer ferramenta que se comunique com o Synapse. Quando uma consulta é executada através do Synapse, os dados armazenados em Parquet particionado permitem que o Synapse leia somente as partições necessárias do Data Lake para atender aos filtros da consulta e levar ao cliente somente o conjunto de dados necessário.

Ao fim da migração, o que se esperava atingir é a proposta de arquitetura e caminho do dado apresentado no diagrama da Figura 2. Com esta proposta é possível

Figura 1 – Representação da estrutura de arquivos no Data Lake do armazenamento dos dados gerados pela ferramenta proposta pela migração.



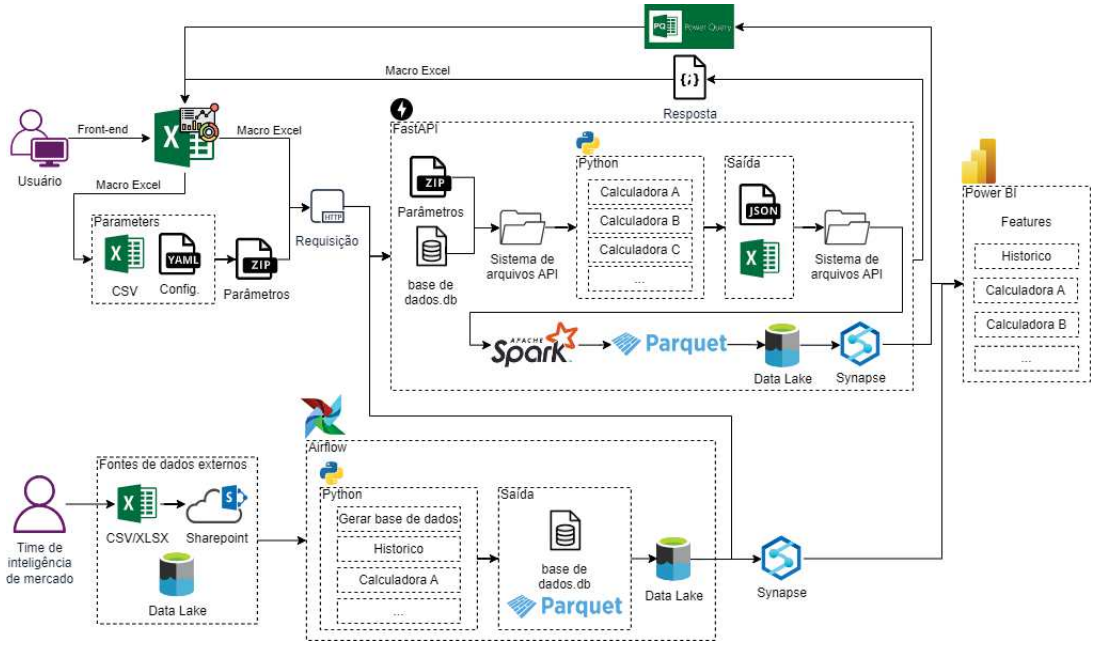
acomodar todos os processamentos na nuvem e modificar parte da jornada de uso dos usuários. Desse modo, o usuário precisa somente baixar o arquivo do *Front-end* do SharePoint, preencher os dados de identificação e da pasta do SharePoint espelhada no computador em suas respectivas células, estar conectado na VPN da empresa e então já estaria tudo pronto para utilizar o *Front-end* conectado aos processos migrados para a nuvem.

A escolha de se utilizar a FastAPI se deu pelos seguintes fatores:

- O desenvolvimento com esta ferramenta é mais rápido do que a maioria das outras disponíveis;
- Pela linguagem utilizada no desenvolvimento ser Python, assim casar com o desenvolvimento já feito da ferramenta;
- Pelos desenvolvedores do cliente dominarem a linguagem e poderem dar suporte posteriormente;
- Possibilitar recebimento e processamento de Requisições nativo.

A decisão de se utilizar o Airflow para os processos que seriam executados automaticamente, foi pelos fatores dele já ser utilizado para orquestrar outros pipelines de dados do cliente; possibilitar agendamento de tarefas; possibilitar executar as calculadoras exatamente como as Macros do Excel chamam com poucas alterações no código; possibilidade de conectar com qualquer recurso público ou no domínio da nuvem do cliente na Azure.

Figura 2 – Diagrama da arquitetura proposta pela migração.



3.2 AMBIENTE DE DESENVOLVIMENTO LOCAL PARA A API

No início da etapa de desenvolvimento da API, para suportar executar os processos em nuvem, foi solicitado ao time de infraestrutura do cliente que fossem disponibilizados dois ambientes utilizando uma imagem Docker criada para a API. Um seria o ambiente de QAS que permite a realização dos testes e validações durante a etapa de desenvolvimento da migração. Ao aprovar as modificações, estas são passadas para o ambiente de produção. No ambiente de produção estaria a versão final e estável da ferramenta migrada para os usuários utilizarem.

No entanto, desenvolver e testar no ambiente de QAS era demorado, trabalhoso e não fluido atrasando o desenvolvimento. Isto era causado pelo fato de que o ambiente de QAS já é na nuvem e todas as alterações que necessitam ser testadas precisam passar por um processo de implantação automatizado que demorava de 5 a 10 minutos. Além disso, quando ocorria um problema durante a execução em QAS só era possível ver os *logs* de erro acessando o Kibana, que é uma ferramenta que consegue capturar

esses *logs* já que a API é hospedada dentro de um container em um POD gerenciado pelo Kubernetes no Azure. No Kibana era necessário realizar uma série de filtros até encontrar a parte do log que mostra o erro desejado e isto era demorado.

Para contornar o desenvolvimento utilizando o ambiente de QAS e melhorar a dinâmica dos testes e identificação de erros, foi criado um ambiente de desenvolvimento local utilizando o WSL. O ambiente de desenvolvimento utiliza a mesma imagem Docker criada para os ambientes de QAS e produção, mas o processo de execução e implantação dentro do WSL era feito manualmente e utilizando-se o Docker Compose. Na Figura 3 pode-se observar o arquivo YAML criado para configurar os serviços da aplicação que posteriormente são definidos e executados pelo Docker Compose que gerencia os contêineres.

Figura 3 – Arquivo YAML do Docker Compose que configura os serviços do ambiente de desenvolvimento.

docker-compose.yml

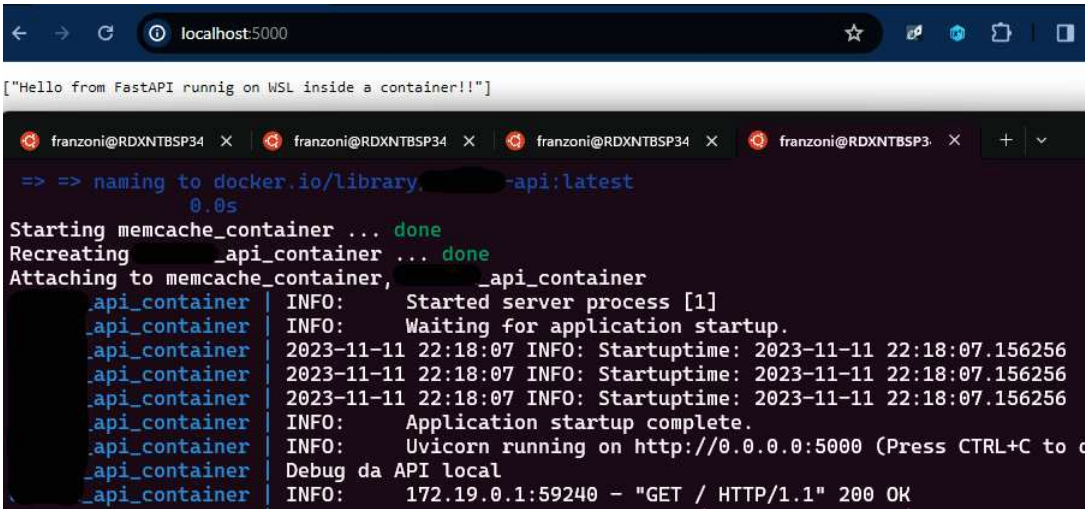
Daniel Franzoni, 4 months ago | 1 author (Daniel Franzoni)

```
1  version: '3'
2  services:
3    dev_memcache:
4      image: memcached
5      container_name: memcache_container
6      expose:
7        - "11211"
8      ports:
9        - "11211:11211"
10     networks:
11       - my-network
12     command: ["memcached", "-m", "100"]
13         _api:
14     build: .
15     image:       -api:latest
16     container_name:       _api_container
17     ports:
18       - "8000:5000"
19     networks:
20       - my-network
21     environment:
22       - PYTHONUNBUFFERED=1
23       - ENVIRONMENT=DEV
24       - MEMCACHE_HOST=memcache_container
25       - MEMCACHE_PORT=11211
26     depends_on:
27       - dev_memcache
28
29 networks:
30   my-network:
31     driver: bridge      Daniel Franzoni, 4 months ag
```

No arquivo apresentado na Figura 3 pode-se ver as configurações dos serviços para o ambiente de desenvolvimento. O primeiro configurado é o da Memcached que é utilizado para guardar algumas variáveis acessadas por processos executados em paralelo. O segundo que aparece no arquivo é o serviço da API que usa uma imagem personalizada para o projeto, onde na linha 14 é indicado pelo parâmetro `build: .` que

o Dockerfile a ser usado para construir a imagem está no mesmo diretório do arquivo YAML.

Figura 4 – Funcionamento da API executando no WSL localmente.



Na Figura 4 pode-se observar um teste da API sendo executada no ambiente de desenvolvimento. Na parte inferior da Figura aparece uma janela do terminal aberta dentro do WSL, que utiliza o Ubuntu, com logs de execução da API em tempo real da requisição realizada por um navegador. Na parte superior da Figura está aberta uma aba do navegador que realizou uma requisição com método GET para a API e pode-se ver a resposta em formato JSON que a API enviou para o navegador.

Para disponibilizar o ambiente de desenvolvimento para realizar os testes é preciso entrar na pasta do projeto onde está o arquivo YAML do Compose através do terminal aberto dentro do WSL, executar o comando `docker-compose up --build` e aguardar o Compose definir e executar os contêineres, então tudo está pronto para ser utilizado. Os processos executados no ambiente de desenvolvimento utilizam as credencias de usuário para realizar operações no Data Lake. As credenciais validadas são armazenadas na Memcached para não precisar realizar uma autenticação em cada operação no Data Lake.

O armazenamento de dados no Data Lake de cada ambiente é separado em camadas, então os dados que são gerados em um ambiente são isolados dos outros. Antes da camada de Aplicação são criadas as camadas DEV, QAS e PRD que correspondem aos ambientes de desenvolvimento, QAS e produção respectivamente. Nas camadas de QAS e PRD tem-se uma credencial de acesso específica para cada ambiente, então usuários só podem ler dados dessas camadas mas nunca editar, apagar ou escrever e em DEV é livre. A única forma de realizar as operações além de leitura nas camadas do Data Lake de QAS e PRD são com suas respectivas credenciais executando as operações dentro dos ambientes.

4 AUTOMATIZAÇÃO DE PROCESSOS

Inicialmente foi identificado que o processo mais oneroso era o que gerava o Histórico e que a migração seria iniciada por ele. Além de o Airflow já estar disponível para receber a migração, este processo não precisava de parâmetros de entrada de um usuário, o que facilitava a migração. Neste caso, era necessário realizar somente as adaptações no código Python para salvar os dados no Data Lake e no Excel para consumir o resultado pelo Synapse.

O primeiro passo para poder executar uma Task no Airflow é criar uma imagem para ser utilizada no executor que define o container em que será executada a Task. Para isto então, a partir de um Dockerfile, foi especificado o que se desejava na imagem. No Dockerfile foi configurada a instalação das dependências necessárias para o Airflow, o Spark, o Azure Data Lake Storage Gen2 e as dependências que haviam sido definidas pelos desenvolvedores da ferramenta. Com essas definições da imagem seria possível executar os processos em Python no Airflow, pois o ambiente da Task seria similar a máquina local de cada usuário.

Com a imagem pronta e disponibilizada pode-se criar a DAG do Airflow onde as Tasks são executadas. A ferramenta já possuía uma DAG criada no Airflow do cliente, nela haviam algumas Task que realizavam a operação de buscar os arquivos disponibilizados pelo time de inteligência de dados no SharePoint e armazenar no Data Lake. Estes arquivos já levados para o Data Lake eram os dados de entrada necessários para executar o processo do Histórico, desta forma era preciso apenas modificar a pasta que estavam sendo armazenados para entrar no padrão de estrutura para o Data Lake proposto.

As Tasks já existentes podem ser vistas na Figura 5 e são:

- `clear_outlier_data`;
- `import_instruments_metadata`;
- `import_input_reuters_prices`;
- `import_input_platts_prices`;
- `import_input_freight_prices`;
- `import_input_freight_flat_rate`;
- `process_instruments_metadata`.

A modificação das pastas em que os dados extraídos do SharePoint eram salvos foi feita pela criação das Tasks subsequentes as que levavam os arquivos para o Data Lake e que iniciam com o nome `change_folder`. A forma como estas Tasks são

definidas é apresentada na Figura 6. É utilizado um PythonOperator nesta Task que recebe o método do Python que realiza a operação de modificar a pasta de um arquivo no Data Lake. O método Python é apresentado na Figura 7 e recebe dois parâmetros, o primeiro é a pasta do Data Lake de onde se deseja pegar o arquivo e o segundo é a pasta de destino que se deseja salvar uma cópia do mesmo.

Figura 5 – DAG implementada no Airflow do cliente com as Tasks criadas na migração.

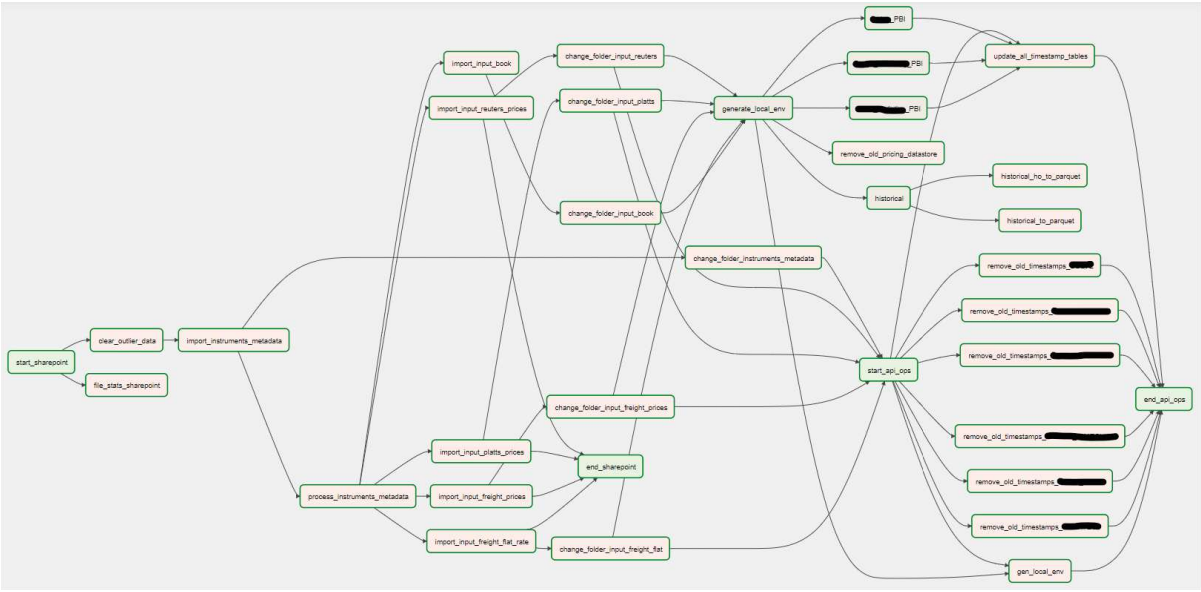


Figura 6 – Definição das configurações das Tasks que iniciam com o nome change _folder no arquivo de configuração da DAG do Airflow.

```
change_folder_instruments_metadata = PythonOperator(
    task_id="change_folder_instruments_metadata",
    python_callable=sharepoint.change_folder_USRZ_to_APPLICATION,
    op_kwargs={
        "uszr_read_path": lake_paths.USRZ_INSTRUMENTS_METADATA,
        "destination_path": BASE_PATH + lake_paths.APP_API_INPUT_INSTRUMENTS_METADATA
    },
    executor_config=EXECUTOR_CONFIG,
    dag=dag,
)
```

Antes de executar qualquer calculadora é necessário montar ou atualizar a base de dados utilizada em todos os cálculos realizados, esta base é montada a partir dos dados trazidos do SharePoint para o Data Lake. Então com esta etapa concluída pode-se realizar as alterações no processo de construção da base, chamado de `generate_local_env`, e montar a Task que iria executar o processo. Esta base de dados, como é utilizada em todos os cálculos, foi disponibilizada no Data Lake para que tanto as Tasks do Airflow que executam as calculadoras quanto a API desenvolvida

Figura 7 – Método Python que realiza a operação de modificar a pasta de um arquivo no Data Lake.

```python
def change_folder_USRZ_to_APPLICATION(uszr_read_path, destination_path):
    """
    Get a copy of the file from USRZ to APPLICATION folder

    :param uszr_read_path: Passar o caminho para salvamento no lake gen2
    :param destination_path:  Path to sharepoint path with files to parse
    """
    gen2 = Gen2Tools()

    # creating a local directory in the cloud to download sharepoint file
    home = expanduser("~")
    dir_path = home + "/Downloads/"
    if not os.path.exists(dir_path):
        os.makedirs(dir_path)
    file_name = uszr_read_path.split('/')[-1]
    download_local_file = dir_path + file_name
    gen2.download_file(uszr_read_path, download_local_file)
    gen2.upload_file(download_local_file, destination_path)
```

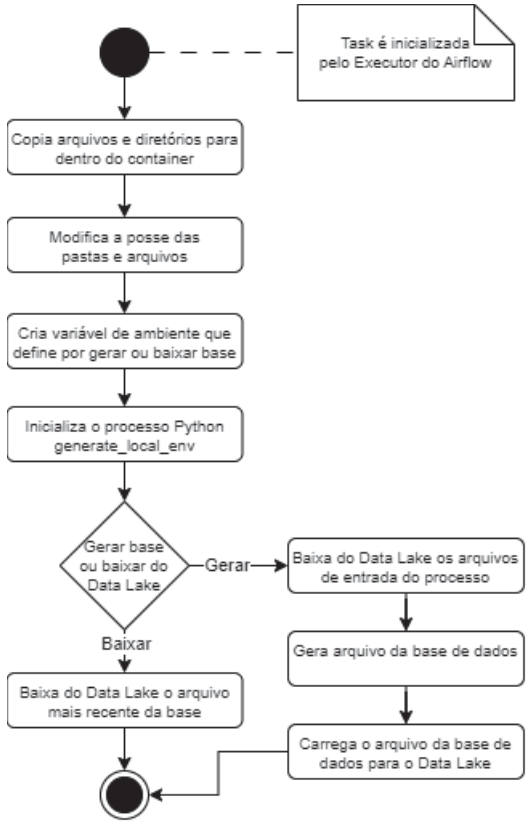
utilizem a mesma fonte de dados. Deste modo, ela precisa ser processada apenas uma vez. A Task `gen_local_env` executada após a Task `generate_local_env` faz uma requisição na API para atualizar a base de dados dentro dela. Um Diagrama de Atividades da Task `generate_local_env` é apresentado na Figura 8, ele ilustra o comportamento da execução desta Task que é descrito a seguir.

A execução de Tasks no Airflow é realizada em um container volátil, isto significa que após o término da Task tudo que está no disco e memória são excluídos para liberação dos recursos. Desta forma, toda vez que um processo que necessita da base de dados é executado em uma Task ele baixa a base já processada para o disco do container. Esta operação é feita na primeira parte do `generate_local_env` que foi alterada para identificar a presença de uma variável de ambiente e decidir se é baixado o arquivo da base de dados ou se é feito o processo normal de gerar a base. Esta alteração pode ser observada na Figura 9.

No caso de ser feito o processo normal do `generate_local_env` é realizada a operação de baixar os arquivos necessários do Data Lake, para então iniciar montagem do banco de dados dentro do container. Esta operação é apresentada na Figura 10. Ao término da montagem do banco é carregado o arquivo gerado para o Data Lake, este é versionado com o *timestamp* do momento em que foi gerado o arquivo. O carregamento para o Data Lake é apresentado na Figura 11.

A configuração da Task do `generate_local_env` no arquivo de configuração da DAG do Airflow é apresentada na Figura 12. Para executar o processo, da mesma forma como era feita no computador dos usuários, utilizou-se um BashOperator para a Task. Este operador, diferente do PythonOperator que recebe um método do Python que é executado automaticamente no início da Task, precisa receber as instruções

Figura 8 – Diagrama de Atividades da Task generate_local_env.



do que e como executar através do parâmetro `bash_command`. Neste parâmetro foram inseridas as instruções de como se desejava executar o `generate_local_env`.

No início do `bash_command` foi inserido o comando `cp -R <caminho fonte> <caminho destino>` para copiar todos os arquivos e diretórios necessários para executar o `generate_local_env` para dentro do container da Task. Em seguida é utilizado o comando `chmod 777 <caminho>`, este modifica a posse das pastas e arquivos copiados para o container para o usuário que executa dentro do container ter permissão de escrita e edição. Por último, cria-se a variável de ambiente que define se será feito o processamento normal ou será baixado o arquivo do Data Lake e na última linha é passado o comando para inicializar o processo `generate_local_env`.

A modificação da posse das pastas e arquivos com o comando `chmod 777 <caminho>` foi necessária pelo fato de não conseguir executar o `generate_local_env` sem realizar essa mudança. Isto acontece, pois as operações realizadas para alocar recurso, agendar as Task, construir a Task e subir o container para executar os processos é realizado pelo usuário do Airflow e dentro do container é outro usuário que realiza as operações, é um usuário definido na construção da imagem dentro do Dockerfile que não tem permissões de usuário raiz no container. Então ao realizar operações dentro do container com os arquivos e pastas copiados

Figura 9 – Operação de baixar o arquivo do banco de dados do Data Lake no processo generate_local_env.

```
    logger.info("Loading the data")
    from params.lake_paths import lake_paths
    from airflow.models import Variable

    gen2 = DataLakeHelper()
    db = DBInterface()

    ENV = Variable.get("AIRFLOW_ENV", default_var='DEV')
    logger.info(f"Get ENV from Variable: {ENV}")
    BASE_PATH = ''
    if ENV.lower() == 'dev':
        BASE_PATH = 'DEV/           _      /'
    elif ENV.lower() in ['qas', 'qa']:
        BASE_PATH = 'QAS/'

    download_pricing_datastore = os.environ.get('DOWNLOAD_PRICING_DATASTORE')
    logger.info(f'ENVIRONMENT VARIABLE DOWNLOAD_PRICING_DATASTORE: {download_pricing_datastore}')
    if (download_pricing_datastore is not None) and (download_pricing_datastore == 'True'):
        file_to_download = gen2.list_dir(
            max(gen2.list_dir(BASE_PATH + lake_paths.APP_PRICING_DATASTORE, file_dir='directory')),
            file_dir='file'
            )[0]
        logger.info(f'File to download from Data Lake PRICING_DATASTORE: {file_to_download}')
        download_from_data_lake(
            file_to_download,
            db._target_db_location
        )
        # finish function
        return

    # initialize the database
```

pelo usuário do Airflow não se tem permissão de escrita e edição, pois estes pertencem a outro usuário e assim se faz necessário o comando de mudar a posse na criação da Task.

Com o banco de dados pronto tem-se os dados de entrada para o processamento do Histórico. Então os próximos passos foram realizar as modificações no código Python, adicionar as configurações da Task no arquivo de configuração da DAG e por último criar a *view* no Synapse para consumir os dados armazenados no Data Lake.

As modificações no código Python do processamento do Histórico foram realizadas a partir da mesma premissa do `generate_local_env` de que ele seria executado como na máquina de um usuário, então os arquivos gerados pelo processo ficariam armazenados no sistema de arquivos do container que a Task executa. Dessa forma, a modificação realizada no código executa a operação de carregar os arquivos salvos no container para o Data Lake, como é mostrado pela Figura 13.

O Histórico era processado anteriormente no computador do usuário de forma a sempre reprocessar todos os anos, desde 2017, e gerava um arquivo no formato CSV com todas as informações. Esta abordagem além de demorar mais tempo re-

Figura 10 – Operação de baixar do Data Lake os arquivos de entrada no processo generate_local_env.

```
78      # initialize the database
79      db.initialize_db(force)
80
81      ingest_config = IngestionConfig.from_yaml(ingest_config_path)
82
83      logger.info("Downloading from Data Lake instrument metadata file")
84      download_instruments_metadata(lake_paths.APP_API_INPUT_INSTRUMENTS_METADATA)
85
86      logger.info("Downloading from Data Lake reuters_pricess file")
87      download_from_data_lake(
88          lake_paths.APP_API_INPUT_REUTERS,
89          reuters_wkb
90      )
91
92      logger.info("Downloading from Data Lake platts_prices file")
93      download_from_data_lake(
94          lake_paths.APP_API_INPUT_PLATTS,
95          platts_wkb
96      )
97
98      logger.info("Downloading from Data Lake freight_prices file")
99      download_from_data_lake(
100         lake_paths.APP_API_INPUT_FREIGHT_PRICES,
101         freight_csv
102     )
103
104     logger.info("Downloading from Data Lake freight_flat_rate file")
105     download_from_data_lake(
106         lake_paths.APP_API_INPUT_FLAT_RATE,
107         worldscale_wkb
108     )
```

Figura 11 – Operação de carregar o arquivo do banco de dados do container para o Data Lake no processo generate_local_env.

```
126     upload_pricing_datastore = os.environ.get('UPLOAD_PRICING_DATASTORE')
127     logger.info(f'ENVIRONMENT VARIABLE UPLOAD_PRICING_DATASTORE: {upload_pricing_datastore}')
128     if (upload_pricing_datastore is not None) and (upload_pricing_datastore == 'True'):
129         logger.info("UPLOADING pricing_datastore.db to Data Lake ...")
130         gen2.adl.upload_file(      Daniel Franzoni, last month • Airflow2 qas (#290) …
131             str(db._target_db_location),
132             os.path.join(BASE_PATH, lake_paths.APP_PRICING_DATASTORE, datetime.now().strftime("%Y%m%d%H%M%S"))
133         )
134         logger.info("UPLOADED to Data Lake ...")
```

processava a maior parte dos dados que se mantém estáticos, pois somente dados dentro do mês corrente que podem sofrer alteração. Desta forma, foi modificado o código para processar o Histórico de forma incremental, onde é gerado um arquivo no formato CSV para os dois últimos meses com a data limite do dia atual da execução do processo. Esta configuração do período de processamento é realizada pela função do Python `config-historical`, chamada na linha 498 do código mostrado na Figura 14, que editava o arquivo de configuração `run_config.yml` utilizado pelas calculadoras da ferramenta.

Posteriormente a Task do Histórico, faz-se a operação de transformar o arquivo

Figura 12 – Configuração da Task do generate_local_env no arquivo de configuração da DAG.

CSV gerado, que está no Data Lake com os dados novos, em Parquet e salvar os dados na partição passada para a operação de escrita do PySpark sobrescrevendo ou criando a partição necessária. A utilização de partições permite processar o histórico de forma incremental e atualizar somente a partição que tem dados novos. A operação de leitura do CSV e escrita do Parquet particionado é apresentada na Figura 15

Caso seja preciso reprocessar todo o Histórico, com as alterações implementadas no projeto, tem-se a opção de escolher o perído de um ano e paralelizar o processamento de cada ano em uma DAG dedicada a isso, como apresentado na Figura 16. No arquivo de configuração da DAG, que é mostrado na Figura 17, realiza-se as configurações da chamada da função de configuração `config-historical --proc-year {year}` e da variável de ambiente `export REPROCESS_HISTORICAL={year}` no container para o processo executar a opção de processar todo um ano específico e salvar no formato esperado respectivamente. Este formato é a organização dos arquivos CSV dentro de pastas que recebem o nome do ano a que o arquivo faz referência. Este padrão de organização é esperado pela operação de leitura do CSV e escrita do Parquet particionado apresentado na Figura 15.

Com a migração dos processos para o Airflow, a área de negócio do cliente identificou uma oportunidade de gerar os dados de três calculadoras de forma automática para alimentar um painel de visão executiva no PowerBI. Então foram criadas as Tasks com o nome no padrão `<nome_calculadora>_PBI` para executar as mesmas calculadoras que os usuários executam pelo Excel na API, mas no Airflow e com parâmetros padrões. As Tasks criadas podem ser observadas na Figura 5.

As três calculadoras são configuradas e chamadas de forma muito similar. As al-

Figura 13 – Operação de carregar o arquivo gerado pelo Histórico no Data Lake.

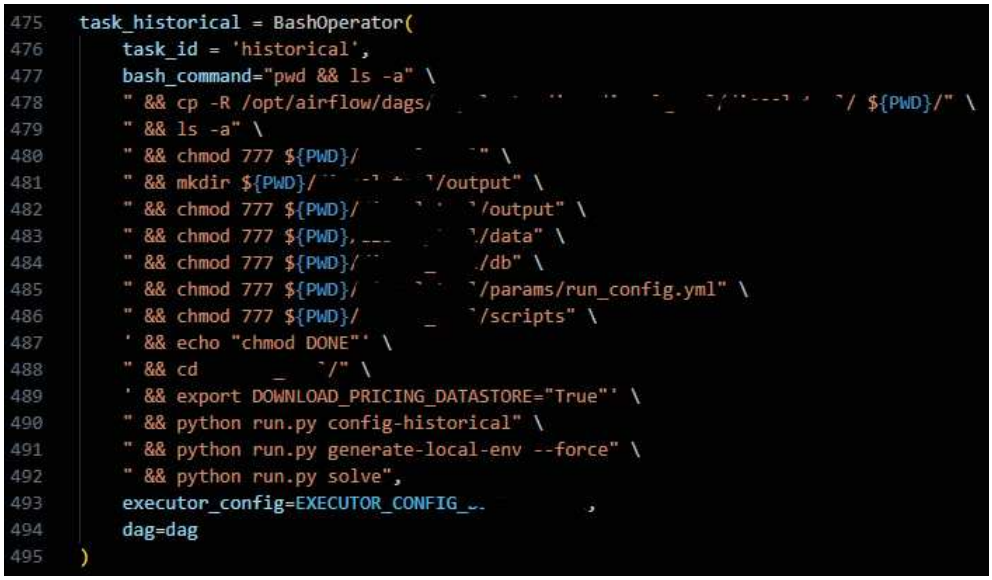
```
# SAVE TO DATA LAKE
from pyspark.sql import functions as F
from pyspark.sql import types as T
from params.lake_paths import lake_paths
from airflow.models import Variable      Daniel Franzoni, last month • Airflow2 qas (#290) …

ENV = Variable.get("AIRFLOW_ENV", default_var='DEV')
logger.info(f"Get ENV from Variable: {ENV}")
BASE_PATH = ''
if ENV.lower() == 'dev':
    BASE_PATH = 'DEV/[illegible]/'
elif ENV.lower() in ['qas', 'qa']:
    BASE_PATH = 'QAS/'

gen2 = DataLakeHelper()

if hist_results is not None:
    logger.info("Uploading historical_spread to Data Lake")
    reprocess_historical_proc_year = os.environ.get('REPROCESS_HISTORICAL')
    reprocess_historical_proc_year = str(
        os.getenv('REPROCESS_HISTORICAL') if reprocess_historical_proc_year is None
        else reprocess_historical_proc_year
        )
    logger.info(f'GET OS GETENV REPROCESS_HISTORICAL VALUE: {reprocess_historical_proc_year}')

    if reprocess_historical_proc_year is None:
        gen2.adl.upload_file(
            str(output_path / 'historical_spread.csv'),
            BASE_PATH + lake_paths.APP_HISTORICAL_SPREAD
        )
        gen2.adl.upload_file(
            str(output_path / 'heating_oil_spread_hist.csv'),
            BASE_PATH + lake_paths.APP_HISTORICAL_SPREAD_HO
        )

    else:
        gen2.adl.upload_file(
            str(output_path / 'historical_spread.csv'),
            BASE_PATH + lake_paths.APP_HISTORICAL_SPREAD_BY_YEAR + '/' + reprocess_historical_proc_year
        )
        gen2.adl.upload_file(
            str(output_path / 'heating_oil_spread_hist.csv'),
            BASE_PATH + lake_paths.APP_HISTORICAL_SPREAD_BY_YEAR_HO + '/' + reprocess_historical_proc_year
        )
```

terações realizadas no código delas foram ler os arquivos de saída salvos no container; criar a coluna que identifica qual usuário gerou o dado; criar a coluna de *timestamp* que versiona o dado e elas são mostradas na Figura 18. Na Figura 18 também é apresentada a alteração para escrever os dados no Data Lake em Parquet particionado pelas colunas de usuário e *timestamp*. Este padrão de escrita particionado é salvo no mesmo lugar que os dados das calculadoras executadas pela API. Então o PowerBI pode consumir os dados gerados por todos os usuários do *front-end*.

No arquivo de configuração da DAG, a Task das três calculadoras é configurada de forma muito similar a do Histórico. Na Figura 20 é apresentada a configuração de uma das Tasks. A única função nova que aparece é a `get-input-book` que realiza a operação de baixar um arquivo do Data Lake que é utilizado como dado de entrada para as calculadoras. Este dado não era importado do SharePoint antes, então foi implementada a Task `import_input_book` para isto.

Ao final da execução com sucesso das três Tasks das calculadoras do PowerBI

Figura 14 – Configuração da Task do Histórico no arquivo de configuração da DAG.



```
task_historical = BashOperator(
    task_id = 'historical',
    bash_command="pwd && ls -a" \
    " && cp -R /opt/airflow/dags/ ${PWD}/" \
    " && ls -a" \
    " && chmod 777 ${PWD}/" \
    " && mkdir ${PWD}/output" \
    " && chmod 777 ${PWD}/output" \
    " && chmod 777 ${PWD}/data" \
    " && chmod 777 ${PWD}/db" \
    " && chmod 777 ${PWD}/params/run_config.yml" \
    " && chmod 777 ${PWD}/scripts" \
    ' && echo "chmod DONE"' \
    " && cd /" \
    ' && export DOWNLOAD_PRICING_DATASTORE="True"' \
    " && python run.py config-historical" \
    " && python run.py generate-local-env --force" \
    " && python run.py solve",
    executor_config=EXECUTOR_CONFIG_...,
    dag=dag
)
```

é executada a Task `update_all_timestamp_tables`. Esta Task faz uma chamada para uma rota da API que realiza a atualização de tabelas dimensão, salvas em parquet no Data Lake, que são utilizadas para mapear todos os dados gerados por todas as calculadoras. Estas tabelas são consumidas posteriormente pelo PowerBI para utilizar em filtros e construir as consultas no Synapse.

Com o intuito de gerenciar melhor os dados e recursos do projeto, foram desenvolvidas Tasks para realizar limpeza de dados antigos. A Task `remove_old_pricing_datastore` remove versões antigas do arquivo do banco de dados guardados no Data Lake, dessa forma mantendo somente os últimos 15 dias de versões. As Tasks com o padrão de nome `remove_old_timestamps_<nome_caculadora>` removem os dados versionados dos usuários gerados pela API que são consumidos pelo Synapse. Além disso, foi desenvolvida a Task `file_stats_sharepoint` responsável por coletar os metadados dos arquivos atualizados no SharePoint, assim é possível monitorar quando um arquivo foi modificado; por quem ele foi modificado e entre outras informações.

Figura 15 – Método do Python passado para a Task historical_to_parquet que lê o arquivo CSV do Histórico e salva no Data Lake o Parquet particionado.

```python
def historical_to_parquet(input_path, output_path, REPROCESS_HISTORICAL=False, **context):
    import logging
    logger = logging.getLogger(__name__)
    from pyspark.sql import functions as F
    try:
        from                                    '.common.gen2 import Gen2Tools
    except ModuleNotFoundError:
        from dags.                              '.common.gen2 import Gen2Tools

    gen2 = Gen2Tools()      Daniel Franzoni, 6 months ago • Airflow2 qas - taking processing from Historic
    spark = gen2.get_spark_session()
    spark.conf.set('spark.sql.sources.partitionOverwriteMode', 'dynamic')

    if REPROCESS_HISTORICAL:
        file_path = list_dir(gen2, input_path, file_dir='dir')
        logger.info(f'Files found for reading: {file_path}')
        for fp in file_path:
            logger.info(f'File found for reading: {fp}')
            df = spark.read \
                .format('csv') \
                .option('sep', ',') \
                .option('header', True) \
                .option('encoding', 'UTF-8') \
                .option("multiline", False) \
                .load(gen2.adls2_full_url(list_dir(gen2, fp, file_dir='file')[0]))

            logger.info(f'Saving parquet at: {output_path}')
            df.withColumn('quote_date_trunc_month', F.to_date(F.date_trunc('mon', F.col('quote_date')))) \
                .write.partitionBy('quote_date_trunc_month').mode('overwrite') \
                .parquet(gen2.adls2_full_url(output_path))

    else:
        file_path = list_dir(gen2, input_path, file_dir='file')
        logger.info(f'Files found for reading: {file_path}')
        df = spark.read \
            .format('csv') \
            .option('sep', ',') \
            .option('header', True) \
            .option('encoding', 'UTF-8') \
            .option("multiline", False) \
            .load(gen2.adls2_full_url(file_path[0]))

        logger.info(f'Saving parquet at: {output_path}')
        df.withColumn('quote_date_trunc_month', F.to_date(F.date_trunc('mon', F.col('quote_date')))) \
            .write.partitionBy('quote_date_trunc_month').mode('overwrite') \
            .parquet(gen2.adls2_full_url(output_path))
```

Figura 16 – DAG implementada no Airflow do cliente para reprocessar todos os anos do Histórico em paralelo.

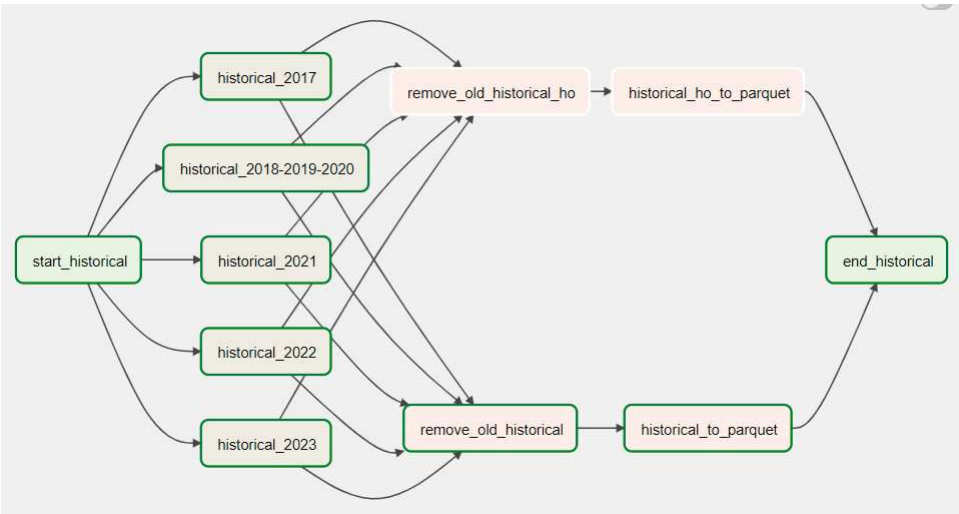


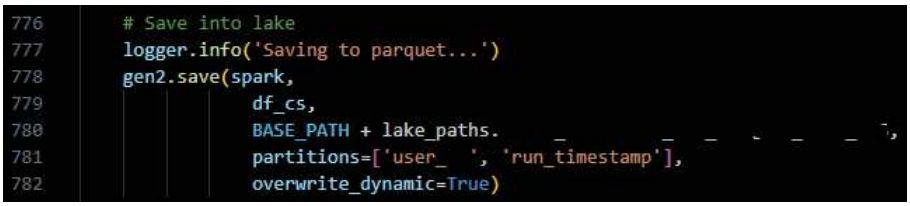
Figura 17 – Configuração da Task que reprocessa todos os anos do Histórico no arquivo de configuração da DAG.

```
77  year_list = []
78  for y in range(2017, datetime.now().year+1, 1):
79      if (y != 2018) and (y != 2019) and (y != 2020):
80          year_list.append(str(y))
81      elif y == 2020:
82          year_list.append('2018-2019-2020')
83
84  task_historical = [
85      BashOperator(
86          task_id = f'historical_{year}',
87          bash_command="pwd && ls -a" \
88          " && cp -R /opt/airflow/dags/                                  / ${PWD}/" \
89          " && ls -a" \
90          " && chmod 777 ${PWD},                                          " \
91          " && mkdir ${PWD},                                  /output" \
92          " && chmod 777 ${PWD},                                  /output" \
93          " && chmod 777 ${PWD}/                                  /data" \
94          " && chmod 777 ${PWD}/                                  /db" \
95          " && chmod 777 ${PWD}/                                  /params/run_config.yml" \
96          " && chmod 777 ${PWD}/                                  /scripts" \
97          ' && echo "chmod DONE"' \
98          " && cd                 /" \
99          f" && export REPROCESS_HISTORICAL={year}" \
100         f" && python run.py config-historical --proc-year {year}" \
101         " && python run.py generate-local-env --force" \
102         " && python run.py solve",
103         executor_config=EXECUTOR_CONFIG_                ,
104         dag=dag
105     ) for year in year_list
106 ]
```

Figura 18 – Operações de leitura de dados gerados no container e criação das colunas para partição das calculadoras executadas no Airflow.

```
717     logger.info("SAVE TO DATA LAKE")
718     from pyspark.sql import functions as F
719     from pyspark.sql import types as T
720     from params.lake_paths import lake_paths
721     from airflow.models import Variable
722
723     ENV = Variable.get("AIRFLOW_ENV", default_var='DEV')
724     logger.info(f"Get ENV from Variable: {ENV}")
725     BASE_PATH = ''
726     if ENV.lower() == 'dev':
727         BASE_PATH = 'DEV/          _ ---- /'
728     elif ENV.lower() in ['qas', 'qa']:
729         BASE_PATH = 'QAS/'
730
731     gen2 = DataLakeHelper()
732     spark = gen2.adl.get_spark_session()
733     timestamp = str(output_path).split('/')[-1]
734     user_cs = 'Airflow_pbix'
735     # Read CSVs in Spark
736     df_cs = gen2.read_csv(spark, str(output_path / '      _          .csv'), local=True)
737     if df_cs.isEmpty():      Daniel Franzoni, last month • Airflow2 qas (#290) …
738         df_cs = create_null_df(spark, df_cs)\
739             .withColumn('user_   ', F.lit(user_cs))\
740             .withColumn('run_timestamp', F.lit(timestamp))
741     else:
742         df_cs = df_cs\
743             .withColumn('user_   ', F.lit(user_cs))\
744             .withColumn('run_timestamp', F.lit(timestamp))
```

Figura 19 – Escrita com Spark de dado particionado no Data Lake das calculadoras executadas no Airflow.



```
776     # Save into lake
777     logger.info('Saving to parquet...')
778     gen2.save(spark,
779               df_cs,
780               BASE_PATH + lake_paths.      _      _   _   _   _   _   ',
781               partitions=['user_   ', 'run_timestamp'],
782               overwrite_dynamic=True)
```

Figura 20 – Configuração da Task da calculadora book_statistics_PBI no arquivo de configuração da DAG.

5 EXECUÇÃO EM TEMPO REAL DE PROCESSOS

No intuito de manter com a migração para nuvem o mesmo comportamento em tempo real da execução local das calculadoras com entradas de usuário, foi realizado o desenvolvimento de uma API utilizando FastAPI. A FastAPI é uma estrutura web moderna para construção de APIs RESTful em Python e, como citado anteriormente no Capítulo 3, era necessário que o desenvolvimento utilizasse linguagem Python.

Uma API fornece todas as características necessárias para a execução dos processos que se deseja executar em nuvem. Primeiramente, o Excel consegue se comunicar com a API. Os arquivos gerados pelo *front-end* que são configurações personalizadas de cada usuário para as calculadoras podem ser enviadas na requisição para a API. A API pode receber requisições não somente do Excel como qualquer outra ferramenta que suporte comunicação com protocolo HTTPS. A API consegue se comunicar com o Data Lake e qualquer outra ferramenta que seja acessível através do Python, assim facilitando integrações futuras. Pode-se centralizar todos os processamentos em uma única API e utilizar a mesma base de dados gerada pelo Airflow. E também as requisições para execução das calculadoras são executadas no momento em que são realizadas, desta forma provendo o comportamento de tempo real desejado.

A API é implementada no Kubernetes em um container que está dentro de um POD dedicado a ela. Neste POD têm-se também uma instância da Memcached em container, que é utilizada para armazenar variáveis compartilhadas entre os processos executados pela API. O container da API utiliza uma imagem similar a que as Tasks com BashOperator utilizam, mas para a API foram adicionadas as dependências necessárias para o funcionamento da FastAPI. E, como citado anteriormente no Capítulo 3, em cada um dos ambientes QAS e PRD foi implementado pelo time de infraestrutura um POD dedicado para a API.

A partir da documentação da FastAPI, foram desenvolvidas as rotas que recebem as requisições com os seus devidos parâmetros. O primeiro parâmetro a ser desenvolvido foi o que recebe os arquivos gerados pelo Excel. Neste parâmetro é recebido, através da requisição, uma pasta compactada que contém todos os arquivos gerados pelo Excel que são utilizados para um determinado processo. Ao receber o arquivo, foi implementado um método que descompacta e organiza estes arquivos em pastas exclusivas para cada usuário, pois pode-se ter múltiplas requisições sendo executadas simultaneamente. No gerenciamento dos arquivos da API, os novos arquivos de um usuário sempre sobrescrevem os antigos dentro da pasta que pertencem a ele. Na Figura 21 é apresentado o método desenvolvido para realizar a descompactação.

O desenvolvimento das rotas para executar as calculadoras pode ser dividido em 8 etapas principais que são executadas em sequência. Estas etapas aparecem

Figura 21 – Implementação do método Python que descompacta os arquivos recebidos pela API.

```python
def unzip_uploaded_file(file, dest_folder=os.getcwd(), dest_zipfile='uploaded_files', user_cs=''):
    if not file.filename.endswith('.zip'):
        raise ValueError(f"File extension must be .zip")
    upload_dir = os.path.join(os.getcwd(), dest_zipfile, user_cs)
    # Create the upload directory if it doesn't exist
    if not os.path.exists(upload_dir):
        os.makedirs(upload_dir)
    # get the destination path      You, 5 months ago • Added new route generatelocalenv and chang
    dest = os.path.join(upload_dir, file.filename)
    logger.info(f'destination UPLOAD: {dest}')
    # copy the file contents
    with open(dest, "wb") as buffer:
        shutil.copyfileobj(file.file, buffer)
    with zipfile.ZipFile(dest, 'r') as zip:
        zipinfos = zip.infolist()
        for zipinfo in zipinfos:
            zip_filename = zipinfo.filename.split('\\')
            zip_filename = zip_filename if len(zip_filename)>1 else zip_filename[0].split('/')
            zipinfo.filename = os.path.join(*zip_filename)
            zip.extract(zipinfo, dest_folder)

    logger.info(f'zip_filename -> value:{zip_filename} | type:{type(zip_filename)}')
    unzip_dest_folder = os.path.join(dest_folder, zip_filename[-2])
    lf = {}
    lf[file.filename] = {
        "unzip_dest_folder": unzip_dest_folder,
        "unzip_files": os.listdir(unzip_dest_folder)
        }

    return ({"uploaded_zipfiles": lf}, unzip_dest_folder)
```

numeradas em um exemplo de uma rota apresentada na Figura 22. O comportamento da execução das rotas das calculadoras é ilustrado pelo Diagrama de Atividades da Figura 23. Na etapa 1, define-se o nome da rota utilizada para realizar a requisição, os parâmetros que podem ser passados na requisição e se será necessário ter autenticação. A etapa 2 mostra a verificação do retorno da autenticação. Caso o *token* enviado pelo usuário não for autorizado, retorna um erro.

Após passar pela autenticação, são inicializadas algumas variáves e em seguida inicia a etapa 3. Nesta etapa, a pasta compactada recebida na requisição é descompactada e o método que inicializa a calculadora é chamado. O caminho dentro da API onde foi descompactado o arquivo do usuário que realizou a requisição é passado para a calculadora, assim como outros parâmetros vindos da requisição quando necessário. Nos métodos que inicializam as calculadoras, também é implementada a lógica que realiza o controle de acesso de recursos que será explicado melhor posteriormente neste capítulo.

Na etapa 4, é encontrada a pasta versionada com *timestamp* onde se tem os arquivos de saída gerados pela calculadora - para então verificar se existe algum arquivo de aviso que informa a ocorrência de problemas na execução da calculadora, mas que não impossibilitam de finalizar o processamento. No final desta etapa, é

adicionado ao dicionário que é utilizado de resposta da requisição para o *front-end* a ocorrência detectada.

Para tratar as exceções que podem ocorrer na execução do código e conseguir administrar e registrar elas nas métricas, foi desenvolvida a etapa 5 que conversa diretamenta com a etapa 6. Na etapa 6, são introduzidas as exceções tratadas pela etapa anterior no dicionário da resposta da requisição.

A ferramenta, em sua versão local, foi desenvolvida para criar pastas versionadas com *timestamp* para cada execução de calculadora. E então, ao final do processamento, colocar todos os arquivos gerados dentro da respectiva pasta da execução. Mas esta funcionalidade nunca exclui estas pastas do sistema de arquivos, no entanto não se quer manter estas pastas dentro da API com dados que já são armazenados no Data Lake. Para resolver este problema, foi implementada a etapa 7 que exclui a pasta versionada encontrada na etapa 4, depois de tudo já movido para o Data Lake. Na útilma etapa, a oitava, é chamado o método que realiza o registro das métricas de uso da ferramenta no Data Lake.

Dentro das rotas, é utilizado para realizar a execução das etapas 3 e 4 o bloco *try* do Python. Este bloco permite que qualquer erro que aconteça dentro do que está sendo executado nele seja tratado pelo bloco *except* que está na etapa 5. Estes dois blocos são os que permitem retornar as exceções da execução da calculadora para o *front-end* e registrar nas métricas, assim como excluir as pastas versionadas geradas. E o bloco *finally* é o que garante que as etapas 6, 7 e 8 sejam executadas sempre que é realizada uma requisição para uma rota.

Como mencionado no Capítulo 4, todas as calculadoras utilizam a base de dados gerada pela Task do Airflow `generate_local_env`. Então a Task `gen_local_env` realiza uma requisição na API para a rota mostrada na Figura 24. Esta rota chama um método do Python para baixar o último arquivo gerado do banco do Data Lake. Nesta rota, tem-se a opção de gerar o arquivo dentro da API em vez de baixar do Data Lake ao modificar o valor padrão do parâmetro `download_or_process`.

Para garantir a atualização correta do banco de dados na API e o uso do banco para os cálculos nas calculadoras sem gerar erros, foram desenvolvidos semáforos em Python utilizando o Memcached. Para utilizar o Memcached com a API, foi desenvolvida uma classe para realizar o papel de interface entre eles. O método gerador `lock`, apresentado na Figura 25, é a base para realizar os bloqueios na lógica dos semáforos. Este método gerador, ao ser chamado com a declaração `with` como mostrado na etapa D da Figura 27, tenta armazenar na Memcached utilizando o método `self.cache.add` a chave passada como parâmetro *key*. O diagrama ilustrado na Figura 26 representa o bloqueio com os semáforos desenvolvidos como na etapa D.

O resultado do método `self.cache.add` é retornado pela declaração `yield` e capturado pela chamada do método gerador. O código continua sua execução dentro

Figura 22 – Implementação de uma rota que excuta uma calculadora.

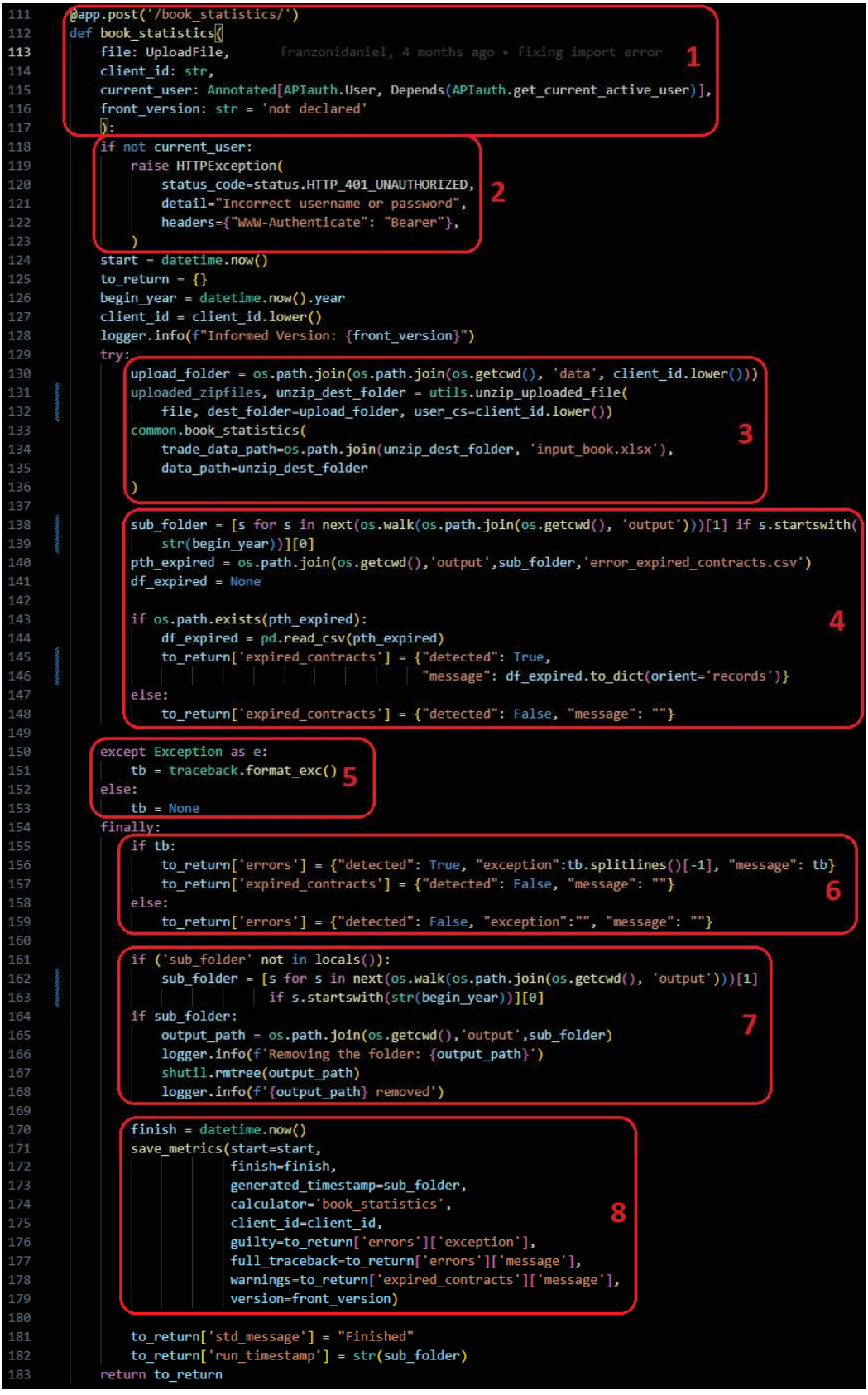
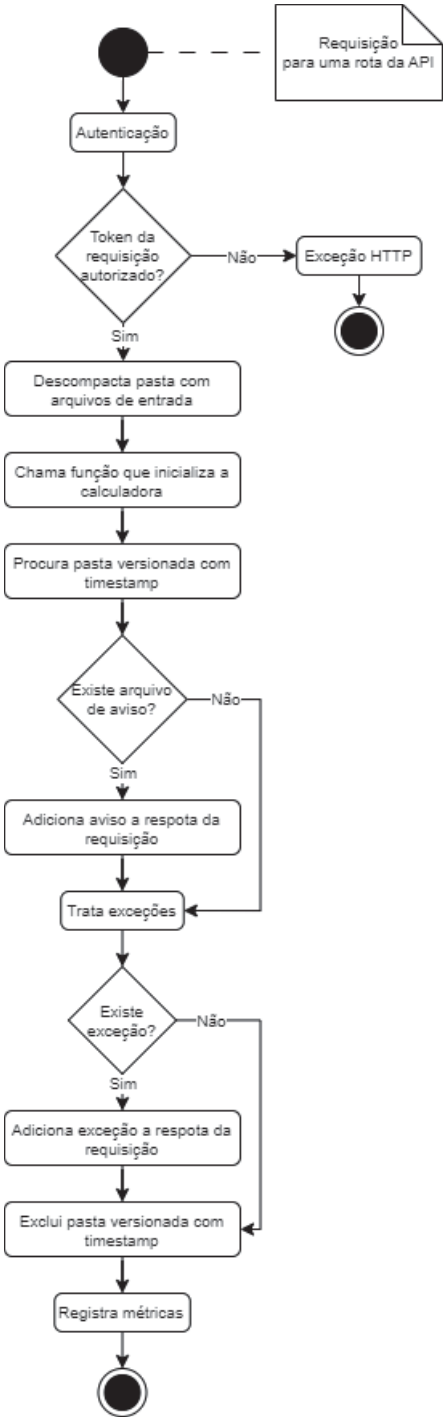


Figura 23 – Diagrama de Atividades das rotas que executam as calculadoras.



do contexto do `with`. Ao término da declaração `with`, a execução volta para o `lock` e o código abaixo do `yield` é executado. E por fim é dado sequência na execução do código após o `with`.

Na etapa de tentativa para armazenar a chave na Memcached do `lock`, quando o retorno do método que armazena é `True` significa que o recurso que a chave representa está livre para ser usado e ele foi bloqueado para o processo. Caso seja

Figura 24 – Implementação da rota que realiza a operação de baixar o arquivo do banco de dados do Data Lake.

```python
@app.get('/generatelocalenv/')
def generatelocalenv(
    current_user: Annotated[APIauth.User, Depends(APIauth.get_current_active_user)],
    download_or_process: str = 'download'
    ):
    if not current_user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    return generate_local_env_no_auth(download_or_process)
```

`False`, o recurso já está sendo utilizado. No final do método gerador `lock` é realizada a operação de exclusão da chave `self.cache.delete(key)`, que significa que o recurso foi liberado.

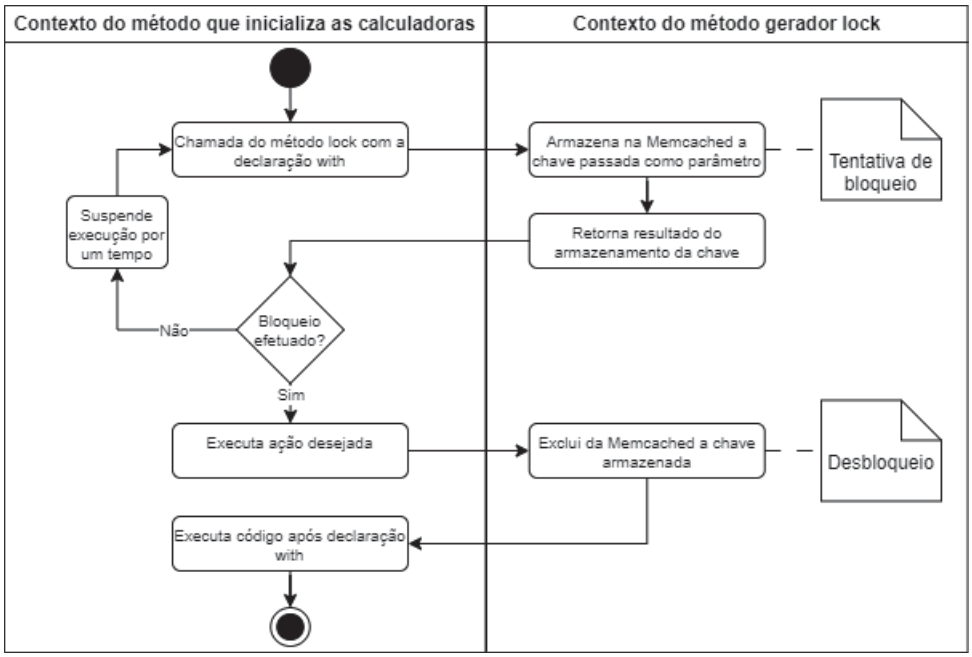
Figura 25 – Método que realiza o bloqueio na lógica do semáforo implementado na classe desenvolvida para ser a interface entre a API e o Memcached.

```python
@contextmanager
def lock(self, key: str, timeout: int =60):
    '''
    Lock calculator to avoid simultaneous process of the same data
    '''
    lock = self.cache.add(key, True, noreply=False, expire=timeout)
    yield lock
    if lock:
        self.cache.delete(key)
```

Os métodos que inicializam as calculadoras e que implementam a lógica para realizar o controle de acesso de recursos são todos muito similares. Então é apresentado na Figura 27 um desses métodos com as etapas mais importantes destacadas. No início dos métodos de inicialização é instanciada a classe de interface do semáforo com a linha `sm = SemaphoreMemcache()`. Em seguida é executado o código da etapa A, que verifica na Memcached se o processo de atualização do arquivo do banco de dados `generate_local_env` está em execução ou aguardando para inicializar. Caso esteja em um desses dois casos fica aguardando a finalização do processo para prosseguir.

Na etapa B da Figura 27, ocorre a sinalização para o semáforo responsável pela informação do número de calculadoras em execução no momento que uma nova calculadora será inicializada. Então, o método `sm.lock_running_calculators` realiza o incremento de um na chave da Memcached do semáforo e a calculadora é inicializada. Em sequência, a etapa C atualiza as tabelas dimensão utilizadas para mapear todos os dados gerados pela calculadora após a correta execução dela. Dentro do método que atualiza as tabelas têm-se um semáforo que controla a atualização delas para que

Figura 26 – Diagrama de Atividades que representa o bloqueio com os semáforos desenvolvidos.

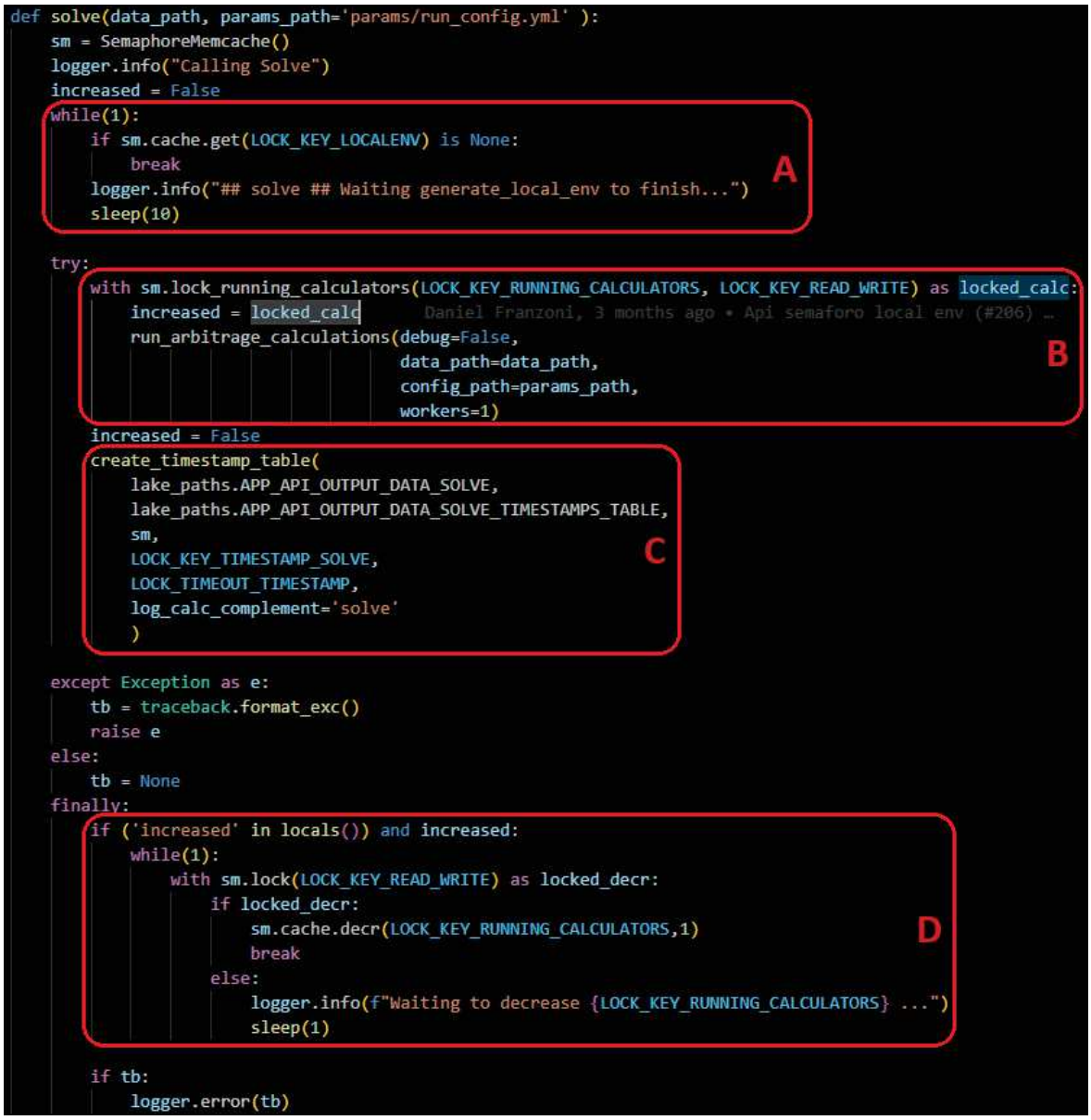


não seja atualizado por calculadoras concorrentes de dois usuários diferentes e ocorra um erro de escrita no arquivo Parquet.

Por último, para assegurar que em caso de problemas durante a execução da calculadora e o semáforo não receber o comando de que a calculadora não está mais em execução, implementou-se a etapa D. Nesta etapa é verificado se foi realizada a operação de incremento na chave da Memcached do semáforo e caso tenha ocorrido é realizada a operação de decremento para atualizar o semáforo.

A rota apresentada na Figura 24 chama o método `generate_local_env` mostrado na Figura 28. Este método é o que efetivamente realiza a operação de baixar o arquivo do banco de dados do Data Lake ou gerar ele dentro da API, esta parte é realizada pela etapa 3A destacada na Figura 28. Mas, além destas operações, o método também utiliza os semáforos para controlar quando é permitido realizá-las. Na etapa 1A, é bloqueada a utilização do recurso do arquivo do banco de dados, mas caso ele já esteja bloqueado por outro processo executando o método `generate_local_env`, a execução é direcionada para a etapa 1B e aguarda o fim deste outro processo e sinaliza que não foi realizado nada, apenas esperou. No outro cenário, caso esteja livre e o bloqueio é realizado, então a execução é direcionada para a etapa 2A e isto significa que a partir deste momento nenhuma calculadora nova pode ser inicializa até o final da etapa 3A. Na etapa 2A, é verificado se alguma calculadora está em execução, caso esteja é esperado até o valor da chave que informa o número de calculadoras chegar em 0 ou *None* e então é iniciada a etapa 3A. Dessa forma, pode-se garantir

Figura 27 – Método que inicializa as calculadoras na API e implementa a lógica para utilizar os semáforos.




```
def solve(data_path, params_path='params/run_config.yml' ):
    sm = SemaphoreMemcache()
    logger.info("Calling Solve")
    increased = False
    while(1):
        if sm.cache.get(LOCK_KEY_LOCALENV) is None:
            break
        logger.info("## solve ## Waiting generate_local_env to finish...")
        sleep(10)

    try:
        with sm.lock_running_calculators(LOCK_KEY_RUNNING_CALCULATORS, LOCK_KEY_READ_WRITE) as locked_calc:
            increased = locked_calc      Daniel Franzoni, 3 months ago • Api semaforo local env (#206) …
            run_arbitrage_calculations(debug=False,
                                       data_path=data_path,
                                       config_path=params_path,
                                       workers=1)
        increased = False
        create_timestamp_table(
            lake_paths.APP_API_OUTPUT_DATA_SOLVE,
            lake_paths.APP_API_OUTPUT_DATA_SOLVE_TIMESTAMPS_TABLE,
            sm,
            LOCK_KEY_TIMESTAMP_SOLVE,
            LOCK_TIMEOUT_TIMESTAMP,
            log_calc_complement='solve'
            )

    except Exception as e:
        tb = traceback.format_exc()
        raise e
    else:
        tb = None
    finally:
        if ('increased' in locals()) and increased:
            while(1):
                with sm.lock(LOCK_KEY_READ_WRITE) as locked_decr:
                    if locked_decr:
                        sm.cache.decr(LOCK_KEY_RUNNING_CALCULATORS,1)
                        break
                    else:
                        logger.info(f"Waiting to decrease {LOCK_KEY_RUNNING_CALCULATORS} ...")
                        sleep(1)

        if tb:
            logger.error(tb)
```

que nehuma calculadora está utilizando ou irá começar a utilizar o banco de dados enquanto ele é atualizado.

As alterações de código realizadas dentro das calculadoras para armazenar os dados de saída no Data Lake são muito similares às desenvolvidas para a execução no Airflow, como apresentado no Capítulo 4. A única diferença é que junto com a transformação e escrita em Parquet no Data Lake é feito o carregamento de um arquivo compactado para realizar auditoria de cada execução de calculadora. No arquivo compactado são inseridos todos os arquivos, tanto de entrada como de saída, utilizados nos cálculos e nomeado com o mesmo *timestamp* da versão utilizada na partição do Parquet.

Figura 28 – Método que realiza a operação de baixar o arquivo do banco de dados do Data Lake ou gerar ele dentro da API e implementa a lógica para utilizar os semáforos.


```
def generate_local_env(download_or_process: str = 'download'):
    sm = SemaphoreMemcache()
    logger.info("Calling generate_local_env ... ")
    processed_or_stand_by = None
    with sm.lock(LOCK_KEY_LOCALENV, timeout=LOCK_TIMEOUT_LOCALENV) as locked:
        if locked:
            logger.info("Locked calculator resource generate_local_env!")
            sleep_time = 0
            while(1):
                with sm.lock(LOCK_KEY_READ_WRITE) as locked_red:
                    if locked_red:       Daniel Franzoni, 3 months ago • Api semaforo local env (#206) …
                        running_calcs = sm.cache.get(LOCK_KEY_RUNNING_CALCULATORS)
                        if (running_calcs == '0') or (running_calcs is None):
                            break
                        else:
                            logger.info(f"Waiting {running_calcs} running calculators to finish...")
                            sleep_time = 10
                    else:
                        logger.info(f"Waiting to read {LOCK_KEY_RUNNING_CALCULATORS} ...")
                        sleep_time = 2
                sleep(sleep_time)

            if download_or_process == 'download':
                logger.info(f'Download pricing_datastore from: {lake_paths.APP_PRICING_DATASTORE} ...')
                gen2 = DataLakeHelper()
                local_path = str(BASE_DB_PATH)
                file_to_download = gen2.list_dir(
                    max(gen2.list_dir(lake_paths.APP_PRICING_DATASTORE, file_dir='directory')),
                    file_dir='file'
                    )[0]
                logger.info(f'File to download: {file_to_download}')
                local_file_path = local_path + '/pricing_datastore.db'
                if os.path.exists(local_file_path):
                    os.remove(local_file_path)
                    logger.info(f"Old file removed: {local_file_path}")

                gen2.adl.download_file(file_to_download, local_path, rename_file='pricing_datastore.db')
                logger.info(f'File downloaded!!')
                processed_or_stand_by = 'download'
                output = 0
            elif download_or_process == 'process':
                output = subprocess.call(['python', 'run.py', 'generate-local-env', '--force'])
                processed_or_stand_by = 'processed'
            else:
                raise NotImplementedError(f'Input param must be [download, process]: {download_or_process}')

            consolidate_metrics_to_one()
        else:
            logger.info("Could not lock calculator generate_local_env because is already processing!")
            logger.info("Waiting...")
            output = 0
            # Wait until other call to API to run generate_local_env finish to use the outputs
            while(1):
                sleep(15)
                logger.info("#### Checking calculator processing status generate_local_env...")
                if sm.cache.get(LOCK_KEY_LOCALENV) is None:
                    processed_or_stand_by = 'stand_by'
                    break

    logger.info("Finished generate_local_env!")
    return processed_or_stand_by, output
```

6 INTEGRAÇÃO FRONT-END E COLETA DE MÉTRICAS

Com a migração da execução das calculadoras da máquina dos usuários para a API, foi preciso desenvolver funções no VBA que implementam as requisições na API para substituir as chamadas das calculadoras em *Shell script* dentro das Macros dos botões do *front-end*. As requisições realizadas em VBA enviam uma pasta compactada e parâmetros, como qual usuário e versão do front-end está executando a requisição, para então receber como resposta um JSON com as informações de como foi a execução e qual o *timestamp* adicionado como nova partição no Parquet armazenado no Data Lake para a consulta no Power Query passar ao Synapse.

No Excel, os arquivos locais gerados pelas calculadoras eram lidos através de consultas do Power Query. Então, com a utilização do Data Lake para armazenar os dados após a migração passou a se utilizar o Synapse como interface para ler os dados através do Power Query. O Synapse abstrai o formato como os dados são salvos no Data Lake e permite que eles sejam consumidos como consultas em SQL. Desse modo, foram criadas *views* no Synapse para ler dos arquivos Parquet que possuem os dados que antes vinham de arquivos CSV e JSON. E assim foram implementadas as mudanças nas consultas do Power Query, como mostrado na Figura 29. As consultas utilizam os parâmetros `user_id` e `run_id` para já trazer o dado filtrado por usuário e *timestamp* da versão do dado.

Figura 29 – Consulta do Power Query do Excel para consumir dados gerados pelas calculadoras executadas na API através do Synapse.

Antes da migração da ferramenta para a Azure, era necessário perguntar aos usuários e olhar dentro das pastas de cada computador quantas vezes e quais calculadoras eles estavam usando. Isto era feito para determinar qual era o engajamento de uso e captura de valor da ferramenta. Mas com os processamentos centralizados, foi possível criar uma lógica para coletar estas informações de forma automática conforme as *requests* na API são realizadas. Na etapa 8, mostrada na Figura 22, pode-se observar o método `save_metrics`. Este método realiza o armazenamento no Data Lake de informações que posteriormente são utilizadas para extrair métricas de uso

da ferramenta. As informações coletadas dentro da rota são o momento de início do processo, o momento de fim do processo, o *timestamp* da versão do dado gerado pela calculadora, qual calculadora foi executada, qual usuário realizou a requisição, qual a exceção do Python do erro que ocorreu durante a execução, qual a mensagem do erro que ocorreu durante a execução, qual o conteúdo do arquivo de aviso que informa a ocorrência de problemas na execução da calculadora e a versão do *front-end* que foi utilizada para realizar a requisição.

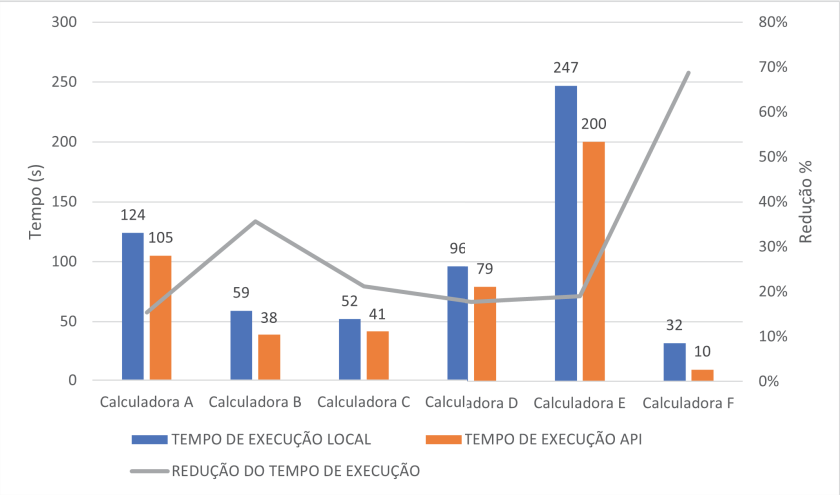
7 RESULTADOS

O projeto de migração para nuvem da ferramenta hospedou todos os processos em Python na Azure. Esta ação trouxe uma jornada de uso aos usuários onde eles não precisam se preocupar em atualizar os dados, pois o que precisa de atualização é feito automaticamente e não é mais utilizado Python na máquina dos usuários. Além disso, o ambiente dedicado à execução das calculadoras melhorou o desempenho dos processos, equalizou a performance entre os usuários, possibilitou novas funcionalidades sem causar impacto na performance e removeu o atraso do VBA executar o *Shell script* que chamava a função do Python para inicializar a calculadora.

Na Figura 30 é apresentado o gráfico de comparação da duração da execução das calculadoras em uma máquina local de um usuário com a nuvem. Para realizar esta comparação, foram utilizados os mesmo dados de entrada em cada calculadora e nos dados em nuvem utilizou-se a coleta de métricas para aferir o tempo da execução. Ao observar os resultados apresentados pelo gráfico, pode-se perceber que em todas as calculadoras houve redução na duração da execução.

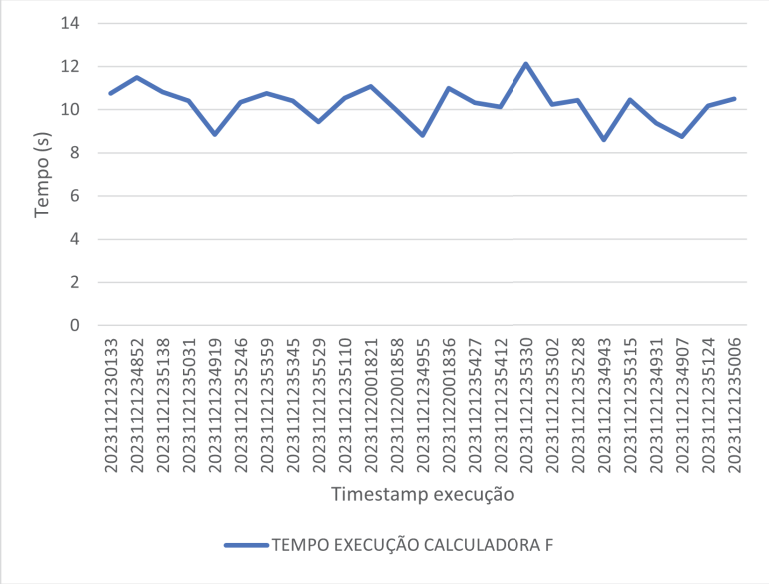
Os resultados apresentados pelo gráfico da Figura 31 mostram que diferentes solicitações de execução na API para uma mesma calculadora resultam em uma duração da execução equalizada, o que garante que todos os usuários tenham um tempo de execução similar. Neste gráfico são apresentados os tempos de duração de 25 execuções consecutivas da Calculadora F utilizando os mesmos dados de entrada em todas as execuções.

Figura 30 – Gráfico de comparação do tempo da execução das calculadoras na máquina local de um usuário com a nuvem considerando os mesmos dados de entrada.



A migração melhorou a governança e segurança dos dados e da ferramenta com a utilização do Airflow, da API e do Data Lake. Pois o processamento dos dados agora

Figura 31 – Gráfico do tempo de duração de 25 execuções consecutivas da mesma calculadora considerando os mesmos dados de entrada.



está em um ambiente controlado com todas as execuções monitoradas. Os dados gerados pelos processos agora necessitam de autorização por credencial para serem acessados e o Data Lake garante maior segurança do que armazenar na máquina dos usuários.

Anteriormente, na execução dos códigos em Python nas máquinas dos usuários, era comum a ocorrência de erros que eram exclusivos a um usuário ou erros distintos entre usuários quando era liberada uma nova versão da ferramenta. Além disso, quando era identificado um erro no código - mesmo que só ocorrendo para um usuário - e havia necessidade de conserto, era preciso gerar uma nova versão e todos os usuários reinstalarem a ferramenta. Porém, com a centralização feita pela migração, foi possível reduzir os erros e replicá-los em um outro ambiente quando necessário. Também se tornou possível identificar erros antes de chegarem ao usuário e quando necessário realizar modificações no código, todos os usuários tem acesso à nova versão no momento em que é disponibilizada a atualização no ambiente de produção.

Outra vantagem da migração é a possibilidade de construir um *dashboard* executivo com dados automáticos, ou até mesmo mostrar os dados gerados por um usuário com as configurações personalizadas nesses relatórios.

A transferência dos dados históricos para o Airflow permitiu o recebimento de novas funcionalidades e o processamento de um número maior de combinações sem que o usuário final tivesse percepção sobre este tempo gasto. Como este processo agora é realizado através de agendamento, o usuário não necessita mais ficar espe-

rando a requisição finalizar. Também tornou-se possível o processamento incremental destes dados. Desta forma, ao atualizar os dados históricos diariamente, apenas os dois últimos meses são recalculados, economizando recursos e tempo.

Uma nova funcionalidade também possibilitada pela migração é a implementação de um monitoramento em tempo real de utilização da ferramenta. Este monitoramento dá acesso às métricas de uso e permite resolver eventuais problemas com maior rapidez e detalhamento, assim como verificar a aderência e o uso da ferramenta.

Com a utilização de um ambiente de computação em nuvem é possível realizar reestruturação para melhorias de perfomance de forma mais flexível sem a preocupação de mau funcionamento no computador de um usuário. Também pode-se escalar o ambiente caso o volume de dados e processamento aumentem, sem haver perda de performance

Contudo, a elaboração de um projeto deste âmbito exige alguns comprometimentos da equipe. Muitas da atividades realizadas durante o desenvolvimento do mesmo resultaram em mais trabalho e maior dificuldade do que o esperado no início. Como, por exemplo, a criação da imagem para as Tasks do Airflow que permitem a execução do processo histórico e das calculadoras. Tais atividades necessitaram de horas de dedicação, pesquisa e discussões dentro do Time.

Outro problema encontrado e sem possibilidade de contornar foi a dificuldade de trabalhar com uma ferramenta como o Excel. Esta ferramenta possui erros exclusivos e pouco explorados pela Microsoft, demandando muitas horas de implementações com tentativas e erros. Além disso, o Excel possui uma perfomance ruim ao se trabalhar com um alto volume de dados.

Por fim, a equipe teve que se desdobrar para trabalhar e manter 3 versões diferentes de desenvolvimento. Isto por que, além de trabalhar no desenvolvimento e nas alterações da nova versão do projeto, a versão anterior que rodava localmente nas máquinas dos usuários precisava evoluir continuamente e ser mantida enquanto a migração não era concluída.

8 CONCLUSÃO

O projeto de migração para nuvem de ferramenta de auxílio à tomada de decisão em operações de compra e venda de ativos utilizando Fast API e Airflow foi um desafio. Todas as decisões relativas ao desenvolvimento da parte técnica que englobam a Engenharia de Dados ficaram a cargo do autor deste trabalho, sendo assim a sua primeira experiência com tal responsabilidade.

Com a hospedagem da ferramenta na Azure, foi possível proporcionar uma jornada mais rápida e fluida para os usuários, possibilitando novas funcionalidades sem causar impacto na performance. Além disso o ambiente dedicado a execução das calculadoras melhorou o desempenho dos processos e equalizou a performance entre os usuários.

A execução da ferramenta em um ambiente de computação em nuvem possibilita realizar a reestruturação das calculadoras utilizando o Spark. Além das calculadoras, o processamento do Histórico e do banco de dados poderiam receber melhorias de perfomance e economia de resursos com esta reestruturação.

O Excel também poderia ser uma outra parte da ferramenta a ser migrada agora que os dados dados e os processamentos estão em núvem, pois já é notável que ele não suporta mais o volume de dados que é carregado nele.

REFERÊNCIAS

ANDRADE, Marcio Roberto. Metodologia Scrum: O Que é, Métodos Ágeis e Guia Prático. 2023. Disponível em: https://blog.contaazul.com/metodologia-scrum/. Acesso em: 06 nov. 2023.

DRUMOND, Claire. O que é scrum e como começar. 2023. Disponível em: https://www.atlassian.com/br/agile/scrum. Acesso em: 06 nov. 2023.

DESSOLDI, Flávia. Método SCRUM — Um resumo de tudo o que você precisa saber. 2019. Disponível em: https://medium.com/reprogramabr/scrum-um-breve-resumo-f051e1bc06d9. Acesso em: 06 nov. 2023.

EKUAN, Martin; ZIMMERGREN, Tobias; MOORE, Gary; PARKER, David; BUCK, Alex; COULTER, David. Como funciona o Azure? 2023. Disponível em: https://learn.microsoft.com/pt-br/azure/cloud-adoption-framework/get-started/what-is-azure. Acesso em: 06 nov. 2023.

MICROSOFT. O que é um data lake? Disponível em: https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-a-data-lake. Acesso em: 06 nov. 2023.

HUMMEL, Guy. Using Azure Data Lake Storage Gen2. Disponível em: https://cloudacademy.com/course/using-azure-data-lake-storage-gen2. Acesso em: 06 nov. 2023.

ESTABROOK, Norm; MYERS, Tamra; SHARKEY, Kent; YOSHIOKA, Hiroshi; COULTER, David; LEE, Dennis; KING, Jeff; JENKS, Alma; SCHONNING, Nick; WOOLEY, Tessa. Introdução ao Azure Data Lake Storage Gen2. 2023. Disponível em: https://learn.microsoft.com/pt-pt/azure/storage/blobs/data-lake-storage-introduction. Acesso em: 06 nov. 2023.

GOOGLE. O que são contêineres? Disponível em: https://cloud.google.com/learn/what-are-containers?hl=pt-br. Acesso em: 07 nov. 2023.

DOCKER. Docker overview. Disponível em: https://docs.docker.com/get-started/overview/. Acesso em: 08 nov. 2023.

DOCKER. Docker Compose overview. Disponível em: https://docs.docker.com/compose/. Acesso em: 08 nov. 2023.

KUBERNETES. Orquestração de contêineres prontos para produção. Disponível em: https://kubernetes.io/pt-br/. Acesso em: 08 nov. 2023.

KUBERNETES. Pods. Disponível em: https://kubernetes.io/docs/concepts/workloads/pods/. Acesso em: 08 nov. 2023.

SUTHERLAND, Jeff; SUTHERLAND, J.J.. Scrum: a arte de fazer o dobro do trabalho na metade do tempo. Rio de Janeiro: Sextante, 2014.

RADIX (Brasil). A Radix. Disponível em: https://www.radixeng.com.br/sobre. Acesso em: 14 nov. 2023.

DATA SCIENCE ACADEMY. Vantagens e Desvantagens do Apache Spark. Disponível em: https://blog.dsacademy.com.br/vantagens-e-desvantagens-do-apache-spark/. Acesso em: 12 nov. 2023.

MEMCACHED. What is Memcached? Disponível em: https://memcached.org/. Acesso em: 12 nov. 2023.

DATA SCIENCE ACADEMY. Apache Airflow: Características, Vantagens e Desvantagens no Cenário da Engenharia de Dados. Disponível em: https://blog.dsacademy.com.br/apache-airflow-caracteristicas-vantagens-e-desvantagens-no-cenario-da-engenharia-de-dados/. Acesso em: 12 nov. 2023.

LOEWEN, Craig; BUCK, Alex; WOJCIAKOWSKI, Matt; JUNKER, Aaron; LEMOINE, Pierre. Perguntas frequentes sobre o Subsistema Windows para Linux. Disponível em: https://learn.microsoft.com/pt-br/windows/wsl/faq. Acesso em: 12 nov. 2023.

FABRO, Clara. O que é API e para que serve? Disponível em: https://www.techtudo.com.br/listas/2020/06/o-que-e-api-e-para-que-serve-cinco-perguntas-e-respostas.ghtml. Acesso em: 10 nov. 2023.

FAST API. Fast API. Disponível em: https://fastapi.tiangolo.com/. Acesso em: 10 nov. 2023.

ALEXANDER, Patrick. What is Azure Synapse ? Disponível em: https://medium.

com/microsoftazure/what-is-azure-synapse-e56f2a8b8d31. Acesso em: 22 nov. 2023.