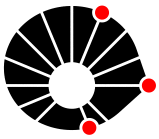
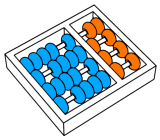




Plataforma de processamento de dados sísmicos como serviço em Nuvem

G. L. da Silva *E. Borin*

Relatório Técnico - IC-PFG-18-29
Projeto Final de Graduação
2018 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Plataforma de processamento de dados sísmicos como serviço em Nuvem

Guilherme Lucas da Silva Edson Borin

Resumo

Este trabalho busca criar uma plataforma que facilite o processamento de dados sísmicos na nuvem, sem exigir do usuário conhecimento técnico dos conceitos que a nuvem traz consigo. Esta plataforma tenta trazer mais agilidade, custos mais baixos e maior flexibilidade para os engenheiros e desenvolvedores que trabalham em aplicações que necessitam de alto poder computacional.

O projeto tem o objetivo de criar uma plataforma *open source*, simples e replicável para qualquer usuário. Os resultados foram alcançados, permitindo uma avaliação inicial da plataforma com ferramentas reais de processamento sísmico.

1 Introdução

Computação em Nuvem é um conceito que está cada vez mais presente no cotidiano de muitas pessoas. Mesmo sem perceber, uma grande quantidade de aplicações que usamos hoje em dia lançam mão desse conceito tão central para o desenvolvimento econômico e tecnológico de nossa sociedade contemporânea. Ele se baseia na capacidade de usarmos o poder computacional de serviços como máquinas virtuais, bancos de dados e redes virtuais sem que o usuário tenha máquinas físicas com tais serviços instalados e configurados. Segundo o artigo intitulado *Cloud Computing Security: A Systematic Literature Review* [1], computação em nuvem é um modelo de rede que torna possível o acesso sob demanda de recursos computacionais configuráveis. Entre as modalidades principais de serviços de computação em nuvem, temos três categorias:

- **IaaS(*Infrastructure as a Service*):** é a categoria que dá mais flexibilidade e responsabilidade ao usuário. Ela exige que o desenvolvedor gerencie de maneira integral máquinas virtualizadas, instale programas necessários, defina configurações e etc. Como um exemplo de IaaS temos as máquinas virtuais que executam sistemas operacionais GNU/Linux e são acessadas remotamente.

- **PaaS(*Plataform as a Service*):** nessa categoria o usuário não precisa se preocupar com pacotes e configurações de sistema operacional. Esse modelo de negócios visa facilitar a publicação de aplicações. Por exemplo, se um usuário quiser expor uma aplicação, não será necessário instalar compiladores ou pacotes para executar o programa naquela determinada linguagem.

- **SaaS(*Software as a Service*):** é o modelo de negócios que traz menos autonomia ao desenvolvedor, uma vez que todo o recurso é gerenciado pelo provedor de nuvem. Como exemplo desse modelo podemos citar os bancos de dados como serviço, onde o usuário não se encarrega de nenhum tipo de infraestrutura, somente usa o endereço que o provedor cede para poder usufruir das possibilidades de armazenamento que a plataforma oferece.

Entre as vantagens de se adotar um modelo de computação em nuvem ao invés de investir em máquinas físicas, podemos citar:

- **Custo:** devido à possibilidade de pagar somente pelo que está sendo usado, usuários desse tipo de plataforma não gastarão para manter sistemas parados.
- **Escalabilidade:** a computação em nuvem permite que o aumento do número de instâncias ou do tamanho das máquinas seja simples, caso a aplicação precise de mais poder computacional.
- **Foco na aplicação:** desenvolvedores podem focar em escrever suas aplicações, sem a preocupação de gerenciar infraestrutura e plataforma, o que pode ser trabalhoso.

Entre os principais provedores de nuvem pública, temos a Microsoft Azure[1], Amazon Web Services[2] e Google Cloud Platform[3], que oferecem diversas opções para computação em nuvem, desde máquinas virtuais até bancos de dados gerenciados.

Aplicações que exigem alto poder computacional são aplicações que necessitam de muito mais poder de processamento que um computador comum pode oferecer. Um exemplo de uma aplicação que exige alto poder computacional é o processamento de dados sísmicos. Várias dessas aplicações são desenvolvidas e executadas em institutos de pesquisas, podendo levar a um custo excessivo, muito trabalho para gerenciamento de infraestrutura, além de pouca agilidade e flexibilidade quanto à arquitetura. Assim, tais aplicações podem tirar muito proveito dos serviços de nuvem citados acima.

2 Objetivos

Este trabalho tem como objetivo desenvolver uma plataforma *open source* de acesso simples, ou seja, o usuário não precisa entender sobre computação em nuvem para executar suas aplicações nesse ambiente. Ao final, é esperada a criação de uma plataforma *web*, que tenha como principais funcionalidades:

- **Submissão e gerenciamento de dados:** essa funcionalidade é responsável pelo *upload* de dados sísmicos utilizados nos processamentos, por consultar quais já foram submetidos, baixá-los e também excluí-los, caso necessário.

[1] Plataforma de computação em nuvem da Microsoft https://azure.microsoft.com/
[2] Plataforma de computação em nuvem da Amazon https://aws.amazon.com/
[3] Plataforma de computação em nuvem do Google https://cloud.google.com/

- **Submissão e gerenciamento de ferramentas:** essa funcionalidade é semelhante ao que está detalhado acima, porém, ao invés dos dados sísmicos, os artefatos que são gerenciados são as ferramentas utilizadas no processamento dos dados.
- **Definição de tarefas de processamento:** utilizando os dados e as ferramentas submetidas a partir das duas funcionalidades citadas anteriormente, o objetivo é conseguir lançar um processamento que combine os dados e as ferramentas, além dos argumentos necessários.
- **Obtenção dos resultados:** ao final dos processos, é desejado que se consiga obter todos os seus resultados de maneira simples e rápida.

3 Relevância

Notamos a relevância da plataforma que esse trabalho objetiva desenvolver, visto que é difícil encontrar outros programas *open source* que executem tal tarefa. Além da unicidade que o trabalho possui no cenário atual, este permite abstrair os novos conceitos e a curva de aprendizado que vem junto com a computação em nuvem. Assim, é possível tirar proveito de todos os benefícios que foram citados acima, levando a possibilidade de um uso mais inteligente dos recursos, baixando custos e aumentando a produtividade.

4 Arquitetura do Sistema

As seguintes definições serão utilizadas ao longo do texto:

- **Dado:** dado sísmico de interesse do usuário. Pode ser um dado carregado no sistema pelo usuário ou produzido a partir do processamento de outro dado.
- **Ferramentas de processamento:** ferramenta que é capaz de processar um dado do usuário. Por exemplo: uma ferramenta de *common mid-point* (CMP) que é capaz de realizar o empilhamento de dados sísmicos com o método CMP.
- **Fluxo de processamento:** é a descrição de como ferramentas devem ser combinadas para realizar operações compostas da aplicação de múltiplas operações em dados sísmicos. Um fluxo pode envolver uma ou mais ferramentas. Um fluxo que envolve múltiplas ferramentas deve descrever o fluxo de dados entre as ferramentas. O fluxo pode ser visto como um grafo dirigido acíclico onde os vértices representam as operações das ferramentas e as arestas o fluxo de dados entre as ferramentas. O usuário pode definir e usar um fluxo para aplicar uma sequência de ferramentas no dado sísmico sem que haja a necessidade de intervenção manual ao término da execução de cada ferramenta. A Figura 1 ilustra um fluxo de processamento.
- **Tarefa:** é um conjunto de informações que descreve como um fluxo de processamento de interesse deve ser executado sobre um dado. Por exemplo, a tarefa define qual é o fluxo a ser executado, os parâmetros para execução do fluxo, os dados de entrada

e outros detalhes. A tarefa também contém informações sobre sua execução. Por exemplo: se a tarefa já foi finalizada, se houve erro, o tempo de processamento, etc.

- **Resultados:** são arquivos produzidos pela execução de tarefas. Os resultados podem ser recuperados pelo usuário através de um processo de *download* ou transformados em dados para processamento posterior pelo usuário.

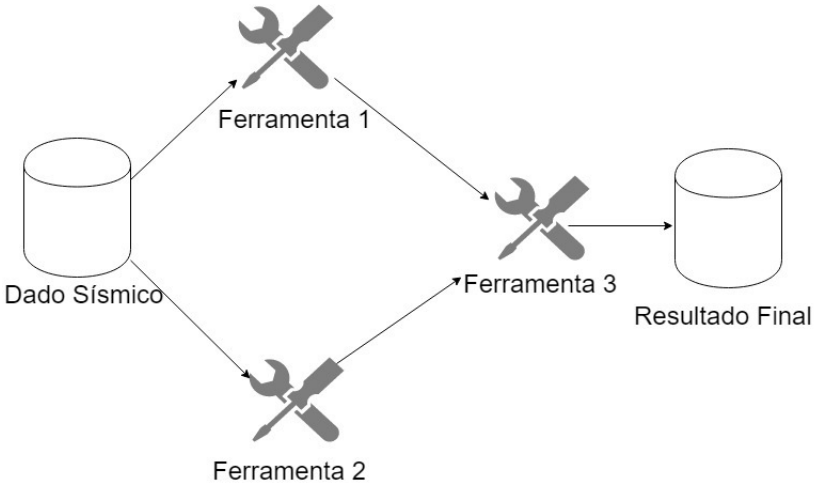


Figura 1: Exemplo de fluxo de processamento

Para o sucesso do projeto, ao começo dele, foi decidido que faríamos uma plataforma *web*. Essa decisão foi tomada devido à facilidade de desenvolver para esse tipo de plataforma, além do alcance que plataformas *web* possuem, já que é praticamente obrigatório nos dispositivos com acesso à Internet (celulares e computadores) a existência de um navegador instalado. Além disso, vale ressaltar que desde o início do projeto todo o código esteve aberto no Github[4], já que foi uma premissa do projeto que este fosse *open source*. Esse código permanecerá disponível.

Dado que os trabalhos são intensivos, demoram um tempo significativo para serem executados e são independentes um do outro, notamos que a arquitetura de nuvem que melhor se encaixa é a de *Work Queues* (Filas de Trabalho), que segundo Brendan Burns em seu livro Designing Distributed Systems, em um sistema de filas de trabalho existe um trabalho em lote para ser executado. Cada parte do trabalho é independente do outro e pode ser executado sem nenhuma interação [2]. A Figura 2 ilustra o desenho inicial do projeto e da arquitetura.

A plataforma foi separada em duas grandes partes:

- ***Front End*:** a parte que o usuário realmente vê, composta pelo código que será executado no navegador do seu dispositivo.

[4]Serviço para compartilhamento de código https://github.com/Guilhermeslucas/SPaaS

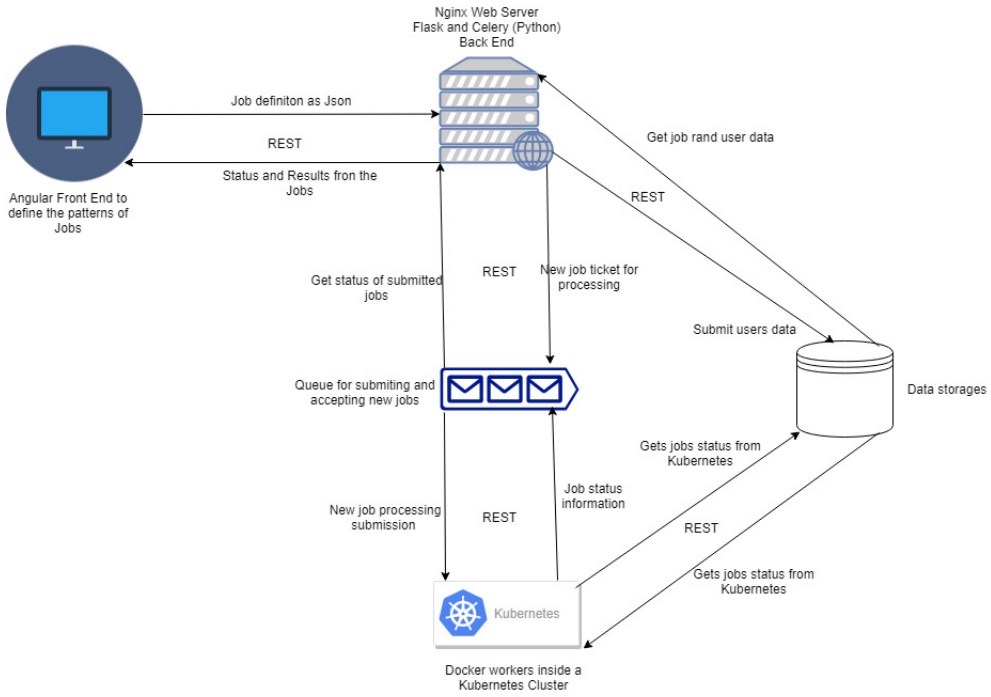


Figura 2: Arquitetura inicial do projeto e exemplo de filas de trabalho distribuídas

- ***Back End*:** a porção da plataforma responsável por características que são invisíveis ao usuário. Entre elas, podemos citar acesso à camada de armazenamento, autenticação, dentre outras.

Os seguintes componentes foram usados em comum entre essas duas partes:

- **Docker[5]:** um sistema para simplificar o empacotamento de aplicações, isolando dependências, garantindo que o *deploy* seja feito da maneira correta. Assim, é possível entregar as aplicações completas, com todos os requisitos necessários instalados, o que dá maior agilidade durante o ciclo de desenvolvimento.
- **Nginx[6]:** servidor *web open source* escalável e de fácil configuração. Foi usado tanto para servir o *back end* quanto o *front end*. A outra opção para esse trabalho seria o Apache Web Server, porém o Nginx se mostrou mais simples e rápido de ser integrado a solução.

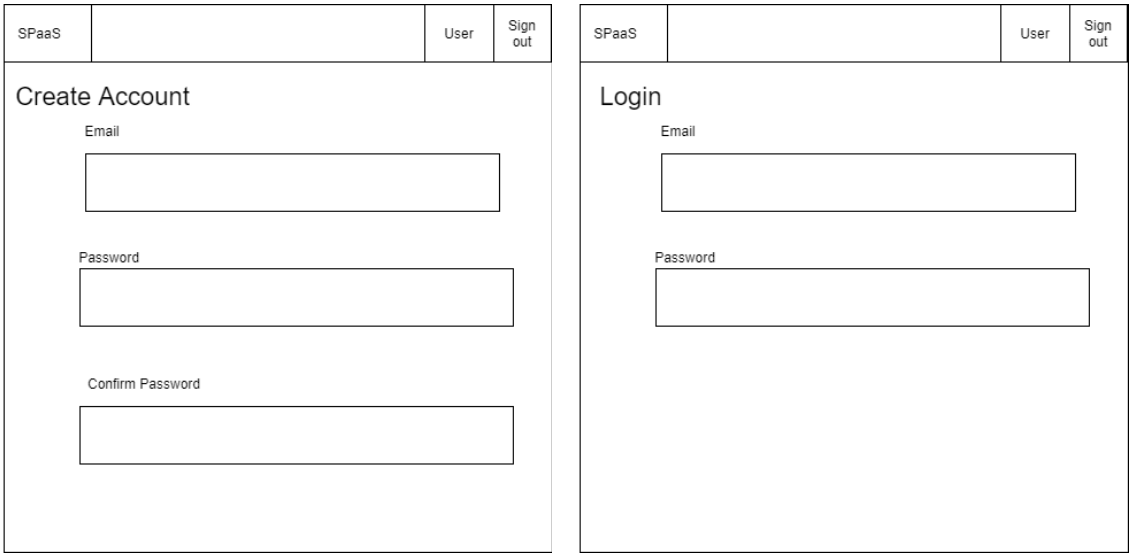
4.1 Projeto Inicial

Para o desenvolvimento do *front end*, foram definidos de antemão a composição geral das telas e o fluxo da aplicação. Uma visão geral das telas e seu conteúdo são descritos a seguir.

[5]Programa que realiza virtualização em nível de Sistema Operacional, também conhecido como conteinerização https://www.docker.com/

[6]Servidor Web *open source* https://www.nginx.com/

A Figura 3a mostra quais informações eram requeridas para se criar uma conta no sistema: *e-mail*, senha e confirmação de senha. Já a Figura 3b mostra como definimos quais informações deveriam estar na tela para que o usuário pudesse se conectar à plataforma. Como mostrado, as informações necessárias para entrar no sistema eram *e-mail* e senha previamente cadastrados.



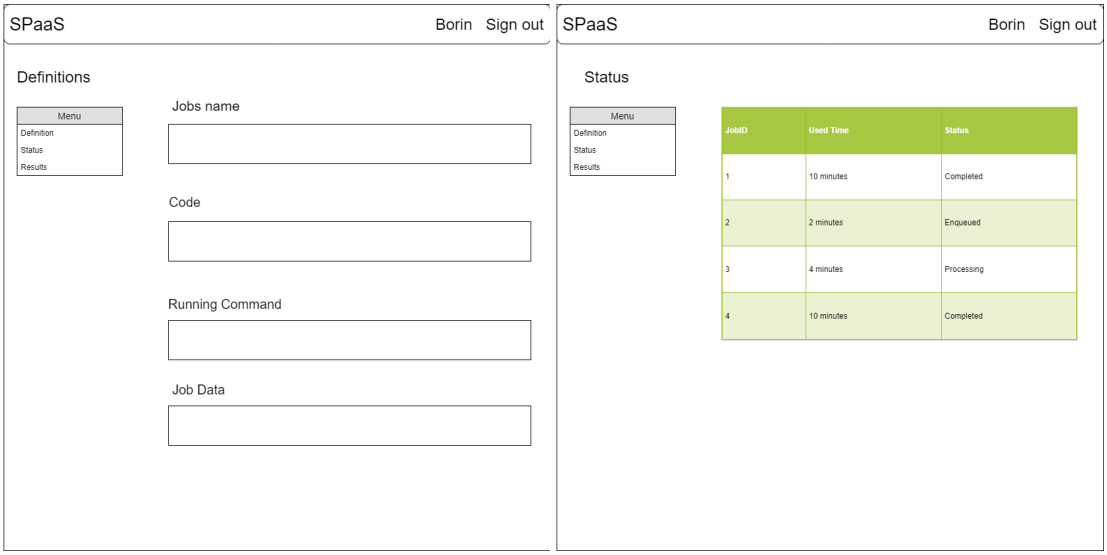
(a) Tela de criação de conta (b) Tela de login

Figura 3: Telas para gerenciamento de usuário

A Figura 4a exemplifica como idealizamos a tela de definição de um trabalho a ser executado. Nesse primeiro protótipo, foi pensado que somente as informações ilustradas na tela (nome do processamento, ferramenta a ser executada, comando necessário para executar a ferramenta e dado sísmico) eram suficientes. O protótipo inicial da tela de estado dos trabalhos está ilustrada na Figura 4b. Simples a princípio, a tela é composta por uma tabela mostrando o estado de cada trabalho. A Figura 5 mostra como foi pensada a obtenção dos resultados.

Vale ressaltar que essa ideia inicial visava permitir uma avaliação inicial do conceito. Após essa avaliação, o fluxo de todo o trabalho definitivo foi organizado em módulos como ilustrado na Figura 6. Nessa figura, as caixas pretas representam as telas, as azuis representam os módulos, com suas respectivas interações com as camadas de armazenamento. Os módulos são detalhados como segue:

- ***Data Management*:** módulo responsável pelo gerenciamento dos dados sísmicos da plataforma. A intenção desse módulo é permitir que o usuário possa gerir os dados na plataforma, sendo capaz de adicionar, remover e mover de um tipo de armazenamento mais caro (porém mais rápido) para um mais barato, e mais lento.
- ***Tools Management*:** composto por somente uma tela, esse módulo procura realizar o gerenciamento das ferramentas de processamento que serão executadas. Assim como



(a) Tela de definição dos trabalhos (b) Tela de observação do estado

Figura 4: Telas para definição e estado dos trabalhos

Figura 5: Tela de obtenção dos resultados

o módulo responsável pelo gerenciamento dos dados sísmicos, esse módulo busca dar a possibilidade de adicionar, mover e excluir as ferramentas de processamento da plataforma.

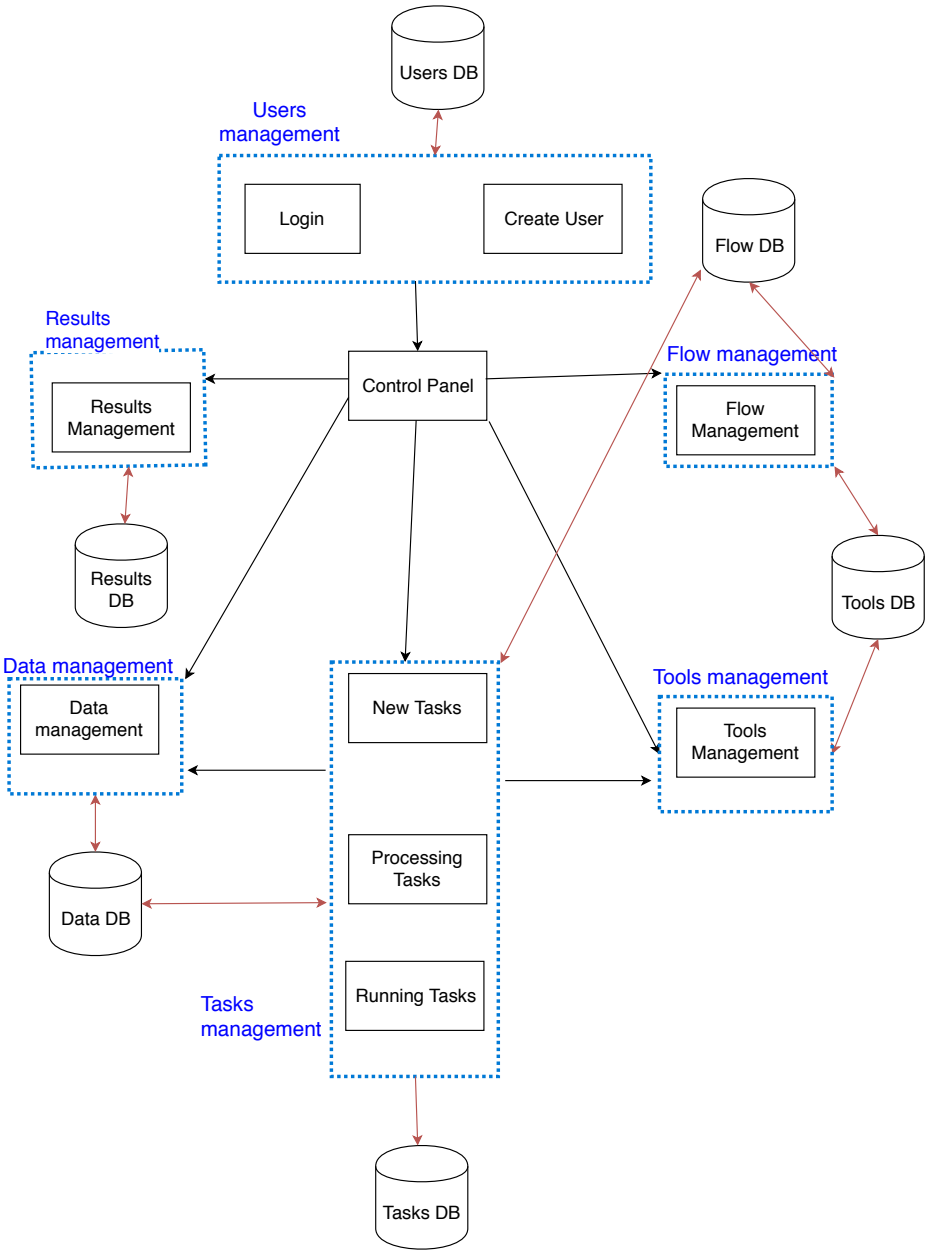


Figura 6: Fluxo de controle (arestas pretas) e de dados (arestas vermelhas) entre os componentes da plataforma

- ***Tasks Management*:** módulo responsável pelo gerenciamento das ferramentas. É composto pelas telas abaixo:

  - ***Running Tasks*:** nessa tela, é desejado que o usuário consiga ver qual é o estado dos processamentos que iniciou. Os possíveis estados são: Executando, Pronto e Falhou.
  - ***New Tasks*:** o usuário usa essa tela para criar as tarefas que deseja executar. Os atributos necessários para essa definição são ferramenta, dado sísmico e argumentos.

- ***Results Management*:** módulo que mostra os resultados obtidos após os processamentos. Possui o identificador único do processamento e o *link* para realizar o *download* dos resultados. Esse módulo também é composto somente por uma tela.

- ***Flow Management*:** esse módulo é o responsável pelo gerenciamento dos fluxos, ou seja, pela descrição de como uma ou mais ferramentas são encadeadas para realizar o processamento de dados sísmicos.

- ***Users Management*:** módulo usado para o gerenciamento dos usuários. É composto por duas telas:

  - ***Login*:** tela onde o usuário entra na plataforma, com as informações previamente cadastradas.
  - ***Create User*:** tela para cadastro de novos usuários.

Assim, as atividades que um usuário deve realizar para que um processamento completo seja executado são as seguintes:

1. **Criação de conta:** ao entrar na plataforma, o usuário deverá criar sua conta no módulo de criação de conta, usando um *e-mail* que ainda não está em uso na plataforma e uma senha. Essa atividade é necessária apenas uma vez por usuário.

2. ***Login*:** Após a criação da conta, o usuário usará seu *e-mail* e senhas cadastrados para entrar na plataforma.

3. **Submissão do dado sísmico:** uma vez que o usuário foi autenticado na plataforma, ele poderá submeter um dado sísmico que deseja processar.

4. **Submissão da ferramenta:** além do dado, também é necessária a submissão de uma ferramenta para o processamento.

5. **Criação da tarefa:** com o dado sísmico e a ferramenta submetida, o usuário deve escolher qual dado sísmico e ferramenta vão compor o processamento. Além disso, o usuário também definirá quais os argumentos necessários.

6. **Acompanhamento da tarefa:** ao criar a tarefa, o usuário poderá consultar se ela ainda está em execução e esperar até ficar pronta, para coletar os resultados.

7. **Obtenção dos resultados:** ao final, o usuário poderá consultar e recuperar os resultados que foram gerados. Todos os arquivos gerados estão reunidos em um arquivo disponível para *download*.

4.2 Tecnologias utilizadas para o desenvolvimento do *Front End*

Para o desenvolvimento da nossa plataforma, especificamente para o *front end*, foram escolhidas as seguintes tecnologias:

- **Angular[7]:** *framework* para desenvolvimento de aplicações *web open source*, desenvolvido inicialmente pelo Google e agora também mantido pela comunidade. Usa como linguagem de desenvolvimento o Typescript[8]. Escolhemos este *framework* pois ele é bastante produtivo e completo, já encapsulando todo o conceito de serviços, estilo de página, linguagem de marcação, roteamento, etc. Ainda é necessário conhecer linguagens de marcação como HTML e CSS, mas a junção de todos esses conceitos fica mais simples.
- **Bootstrap[9]:** *framework* que possui o estilo de vários componentes *web*, como tabelas, menus laterais, barras superiores, etc. É utilizado por muitos *sites*, como por exemplo o Github. Como estilizar páginas *web* pode se tornar um trabalho complicado e demorado, optamos por usar este *framework*.

Para o *deploy* foi utilizado o servidor *web* Nginx. Além disso, foi disponibilizado um Dockerfile, que possibilita o *deploy* da solução de maneira simples e direta sobre *containers* Docker.

4.3 Ferramentas para o desenvolvimento do *Back End*

No início do desenvolvimento do *back end*, buscamos escolher as melhores tecnologias para esse componente e definir as rotas e contratos que essa API iria expor. Assim, os componentes que escolhemos para o *back end* foram:

- **Python[10]:** linguagem criada por Guido Van Rossum, muito presente nos dias de hoje devido à sua extensa gama de aplicações, que vão desde embarcados até sistemas de análise de dados de alto desempenho. Esta linguagem provê bibliotecas maduras para desenvolvimento de API's *back end*, filas de trabalho distribuídas e conectores com bancos de dados.
- **Flask[11]:** *framework* para desenvolvimento de API's para Python. Permite a criação de API's de modo fácil e rápido. Com ele, é simples entender como a aplicação está funcionando, fator muito crítico para sua seleção, uma vez que o projeto tem como premissa ser *open source*. Foram cogitados também ASP.NET com C# e NodeJS. A primeira foi descartada devido à complexidade para criação de uma API. NodeJS não foi selecionado devido à dificuldade que novos programadores podem enfrentar para compreender suas funcionalidades, quando precisassem expandir o projeto.

[7] *Framework open source* para desenvolvimento *web* https://angular.io/
[8] Linguagem criada pela Microsoft. https://www.typescriptlang.org/
[9] Conjunto de ferramentas *open source* que ajuda na estilização de páginas https://getbootstrap.com/
[10] https://www.python.org/
[11] *Framework web open source* para Python http://flask.pocoo.org/

- **Celery**[12]**:** é um componente do Python responsável por gerenciar, submeter e obter resultados de uma fila de trabalho, de uma maneira acessível. Além disso, dá suporte a vários componentes de entrega de mensagens, por exemplo, RabbitMQ[13], Redis, Amazon SQS[14]. Uma vez escolhidas Python e Flask como ferramentas para o desenvolvimento da solução, foi natural a escolha do Celery como ferramenta para filas de trabalho. Existem casos de sucesso documentados por engenheiros e desenvolvedores que usaram essa combinação com eficácia.

- **Redis**[15]**:** após a escolha da biblioteca para facilitar a execução das filas de trabalho, a Celery, foi preciso escolher o componente para a entrega de mensagens do projeto, ou seja, o componente que realmente é responsável por armazenar e distribuir as mensagens relacionadas aos trabalhos submetidos na solução. Entre as opções estavam RabbitMQ, Redis e Amazon SQS. Essas são as que melhor se integravam com o Celery, sendo possível obter todas as suas funcionalidades. A Amazon SQS é uma opção dependente de um provedor de nuvem, o que dificulta o desenvolvimento do projeto, caso seja preciso utilizar outra *cloud* diferente da AWS. RabbitMQ e Redis oferecem funcionalidades semelhantes. O Redis foi escolhido, devido à sua documentação e suas outras funcionalidades que podem ser aproveitadas, por exemplo, *cache* e armazenamento de dados em memória.

- **MongoDB**[16]**:** para o armazenamento dos dados de cadastro de usuários, informações sobre argumentos de cada ferramenta e etc., escolhemos um banco de dados NoSQL, o MongoDB. Tratou-se de uma opção natural, uma vez que os dados que estavam trafegando entre um serviço e outro tinham uma estrutura muito semelhante com os documentos que são armazenados nesse tipo de banco.

- **Blob Storage**[17]**:** precisávamos de um armazenamento de propósito geral. Além do Blob Storage da Microsoft existem outras opções: Amazon S3[18] e máquinas virtuais. O Amazon S3 oferece serviços muito semelhantes ao Blob. Por outro lado, usar máquinas virtuais aumentaria a complexidade do projeto, já que seria necessário gerenciá-las. O Blob Storage da Microsoft Azure possui integração com Python está sendo usado para armazenamento dos dados sísmicos, ferramentas e resultados.

4.4 Ferramentas para *Deploy*

Como dito anteriormente, ao iniciar o projeto, não era desejado que utilizássemos somente um modelo de *deployment*. Assim, para facilitar o *deploy* da aplicação, criamos uma série de *scripts* em ferramentas específicas:

[12] *Framework open source* para tarefas distribuídas em Python http://www.celeryproject.org/
[13] Plataforma *open source* de mensagens https://www.rabbitmq.com/
[14] Sistema de mensagens da plataforma AWS https://aws.amazon.com/sqs/
[15] Armazenamento de dados em memória *open source* https://redis.io/
[16] Banco de Dados NoSQL *open source* https://www.mongodb.com/
[17] Armazenamento de caráter genérico no Microsoft Azure https://azure.microsoft.com/en-us/services/storage/blobs/
[18] Armazenamento genérico na AWS https://aws.amazon.com/pt/s3/

- **Kubernetes**[19]**:** renomado orquestrador de *containers* que tem como principal funcionalidade simplificar o *deploy* de aplicações baseadas em *containers*. Nascido dentro do Google, a partir do Borg, que segundo o artigo publicado pelo Google [3], é um gerenciador de *cluster* que roda centenas de milhares de processamentos, de diferentes aplicações entre uma série de *clusters*. Atualmente está sob a jurisdição da Cloud Native Computing Foundation[20] e é *open source*. Foi uma das plataformas visadas por ter apresentado relevância nos últimos tempos, com todos os grandes provedores de nuvem oferecendo opções gerenciadas da ferramenta.

- **Ansible**[21]**:** um modelo comum de *deploy* de aplicações é o baseado em máquinas virtuais. Nesse caso, é bom garantir que todas as dependências necessárias para que a aplicação seja executada com sucesso estejam instaladas. Nesse aspecto, o Ansible se mostra eficaz. Essa ferramenta surgiu na Red Hat e também é mantida *open source*. É um ótimo gerenciador de configurações, o que torna possível escrever configurações como código, garantindo que a aplicação sempre tenha sucesso ao ser executada.

Após combinar todas as tecnologias descritas nas seções anteriores, foi alcançado um resultado muito próximo ao esperado. A arquitetura final do projeto, está ilustrada na Figura 7.

5 Resultados

5.1 Arquitetura

Dada a arquitetura final do projeto, os componentes ilustrados têm, cada um, a seguinte função:

- ***Web Client*:** componente que é executado no lado do cliente. Seguimos com o que foi proposto no início do projeto e explicado nas seções anteriores. Todo o código do lado do cliente foi escrito em Typescript, com o auxílio do *framework* Angular. A estilização das páginas foi feita com Bootstrap e exposto via Nginx.

- **Flask *Back End*:** responsável por receber todas as requisições que saem do *front end*. Funciona como um *gateway*. Também foi seguido o plano inicial de escrever todo o *back end* com Python, Flask e Celery. Esse é o componente que possui mais responsabilidades: autenticação, comunicação com a fila com o auxílio do Celery, acesso aos Blobs e acesso ao banco.

- **Linux *Workers*:** componente responsável pela execução dos pedidos de trabalho. Com o auxílio do Celery, esperam os pedidos entrarem na fila, e assim que isso acontece, os executam. É possível ter um *cluster* de máquinas com esse papel, aumentando o paralelismo das execuções.

[19] Orquestrador de *containers open source* https://kubernetes.io/
[20] Cloud Native Computing Foundation https://www.cncf.io/
[21] Gerenciador de configuração *open source* https://www.ansible.com/

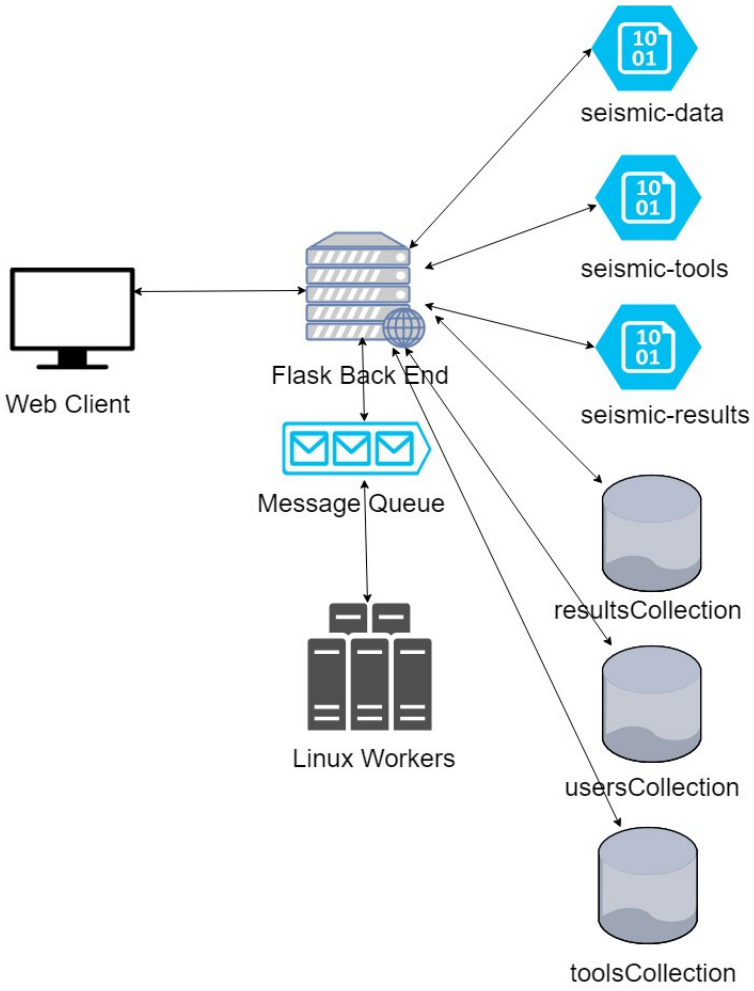


Figura 7: Arquitetura final do projeto

- ***Message queue*:** também parte do plano inicial, a fila de mensagens com o Redis foi a maneira mais eficaz e resiliente que encontramos de desacoplar o *back end* dos Linux *workers*. O *back end* submete uma mensagem com as definições para a execução de um trabalho na fila e algum *worker* obtém essa requisição, realizando o processamento necessário. Vale ressaltar que a escolha do Celery e do modelo de filas de trabalhos distribuídas, que já foi explicada anteriormente, torna possível desacoplar o *back end* dos *workers* e a execução de processamentos em paralelo e distribuídos.

- **toolsCollection:** coleção dentro do banco que guarda informações sobre as ferramentas que estão armazenadas. Essas informações são o nome da ferramenta e seus argumentos. Para cada argumento, é armazenado seu nome e descrição. Novos dados são inseridos nela quando: uma nova ferramenta é armazenada, um novo pedido de processamento é incluído e uma ferramenta é excluída. Um exemplo de descrição de uma ferramenta que segue o modelo esperado está expresso na Figura 8.

```
{
    "_id" : ObjectId("5be74b71c0f26007790bc37d"),
    "args" : [
        {
            "description" : "Azimute (em graus) do dado sol",
            "name" : "1"
        },
        {
            "description" : "Velocidade inicial",
            "name" : "2"
        },
        {
            "description" : "Velocidade final",
            "name" : "3"
        },
        {
            "description" : "Step da velocidade",
            "name" : "4"
        },
        {
            "description" : "Abertura limite em meio afastamento (Semblance)",
            "name" : "5"
        },
        {
            "description" : "Abertura limite em meio afastamente (Empilhamento)",
            "name" : "6"
        },
        {
            "description" : "Dado prestack",
            "name" : "7"
        },
        {
            "description" : "Pasta de armazenamento dos dados postack",
            "name" : "8"
        }
    ],
    "name" : "cmp"
}
```


Figura 8: Exemplo de um documento presente na coleção de ferramentas

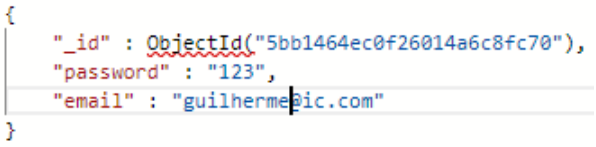
- **resultsColleciton:** coleção de documentos no banco que armazena os dados dos resultados. Armazena o id do processamento, nome do dado e da ferramenta usada, além dos argumentos. Essa coleção é acessada após o término de cada processamento para listar e salvar resultados. Um exemplo de documento que segue o modelo esperado está ilustrado na Figura 9.

- **usersCollection:** coleção usada para armazenamento de dados dos usuários, como senha e *e-mail* para autenticação. Um novo registro é criado nessa coleção quando um usuário cria uma nova conta. A coleção também é acessada para autenticação de

```
{
    "_id" : ObjectId("5be7726ec0f2600daa14e026"),
    "tool" : "cmp",
    "data" : "sol.su",
    "id" : "4b1785ef-485e-4a96-8fd2-0fa5ec929f8b",
    "args" : {
        "1" : "0.0",
        "2" : "1500.0",
        "3" : "4500.0",
        "4" : "50.0",
        "5" : "2000.",
        "6" : "2000.",
        "7" : "sol.su",
        "8" : "sol"
    }
}
```


Figura 9: Exemplo de um documento presente na coleção de resultados

um novo usuário. Um exemplo de documento armazenado com as informações dos usuários está ilustrado na Figura 10.



```
{
    "_id" : ObjectId("5bb1464ec0f26014a6c8fc70"),
    "password" : "123",
    "email" : "guilherme@ic.com"
}
```


Figura 10: Exemplo de um documento presente da coleção de usuários

- **seismic-tools:** Blob usado para armazenar as ferramentas submetidas pelos usuários. Acessado pelo componente de gerenciamento de ferramentas quando uma nova ferramenta é inserida na plataforma ou quando um novo processamento acontece.
- **seismic-data:** Blob que armazena os dados sísmicos submetidos. Também são acessados pelo componente de gerenciamento de dados quando um novo dado é incluído, excluído e na execução dos trabalhos.
- **seismic-results:** esse Blob é responsável por armazenar os resultados dos processamentos. Todos os resultados são empacotados em um arquivo tar.gz[22] e disponibilizados para *download* no componente de resultados.

Vale notar que, usamos somente uma instância de bancos de dados, com coleções diferentes para cada aplicação, as *Collections*. Também usamos a mesma estratégia para o Blob. Foi usada somente uma conta de armazenamento no Microsoft Azure, com pastas separadas para cada uso: seismic-data, seismic-tools e seismic-results.

[22]utilitário de compressão de dados https://www.gzip.org/

5.2 Executando a plataforma

Esta seção é dedicada à explicação de como executar cada componente da plataforma. Desse modo, descreveremos o *front end, back end* e Linux *workers*.

5.2.1 *Front End*

Para executar o *front end*, as variáveis de ambiente estão em `front-end/src/environments`. Nesse caminho existem dois arquivos, `environment.ts` e `environment.prod.ts`. O primeiro contém os valores das variáveis de ambiente que serão consideradas durante o desenvolvimento, ou seja, quando o comando for executado:

```
$ ng serve
```

Assim, a única variável que deve ser alterada é a apiUrl, que deve apontar para o *back end*. Para compilar o projeto e serví-lo usando um servidor *web*, os comandos que devem ser executados no servidor são os seguintes:

```
$ npm install
$ npm run build -- --output-path=./dist/out --configuration production
```

Isso gerará uma pasta `./dist/out` que, quando colocada no caminho `/usr/share/nginx/html` de um computador com Nginx instalado, irá servir a aplicação do *front end*. Nesses últimos comandos, o arquivo de variáveis que será usado é o `environment.prod.ts`. Outra maneira de executar o *front end* é via Docker. Existe um Dockerfile dentro da pasta *front end* que facilita o *deploy*. Neste caso,existem dois comandos que devem ser executados, o primeiro para construir a imagem do Docker e segundo para executá-la:

```
$ docker build -t my-angular-project:prod .
$ docker run -p 80:80 my-angular-project:prod
```

Ao acessar o *localhost*, a tela inicial do projeto será apresentada.

5.2.2 Tecnologias utilizadas para o *Back End*

Aqui, assim como no *front end*, também existe a possibilidade de executar o código nativamente ou dentro de um *container* Docker. Para executar o *back end* nativamente, dentro da pasta *back end* existe o arquivo `main.py` e o `requirements.txt`. O arquivo `requirements.txt` mostra quais são os pacotes necessários para executar a aplicação. Também é preciso definir três variáveis de ambiente, que são:

- `SPASS_CONNECTION_STRING`: *string* de conexão do MongoDB.
- `SPASS_DATA_BLOB_KEY`: chave referente ao Blob que guardará todos os componentes citados anteriormente: resultados, dados e ferramenta.

- `SPASS_CELERY_BROKER`: *string* de conexão do *broker* do Celery, que, no caso do nosso experimento é o Redis.

Assim, após adicionar essas variáveis de ambiente, dentro da pasta *back end* execute os seguintes comandos:

```
$ pip3 install -r requirements.txt
$ python3 main.py
```

Caso a opção seja por executar o *back end* com Docker, os comandos são:

```
$ docker build -t flask-back .
$ docker run -p 5000:5000 -e SPASS\_CONNECTION\_STRING="<value>"
-e SPASS\_DATA\_BLOB\_KEY="<value>"
-e SPASS\_CELERY\_BROKER="<value>" flask-back
```

Nesse caso, o *back end* estará executando na porta 5000.

5.2.3 Worker

Para os *workers*, dentro da pasta *back end*, basta executar:

```
$ pip3 install -r requirements.txt
$ celery worker -A main.celery -l info
```

Nos *workers* também é necessário declarar as mesmas variáveis de ambiente do *back end*, citadas anteriormente. Como a proposta é ter vários *workers* para executar as tarefas em paralelo, foi criado um *playbook* Ansible, para configurar várias máquinas para executar essas tarefas. Para isso, com o Ansible instalado basta executar dentro da pasta `devops`:

```
$ ansible-playbook worker_config.yaml
```

5.3 Telas finais

Após apresentar a arquitetura final, ilustramos os componentes, com suas respectivas telas. Na tela final de *login*, Figura 11, existem dois campos: *e-mail* e senha. Além disso, caso o usuário não possua uma conta, é possível criar uma no canto superior direito. Ao escolher a opção de criar uma conta, o usuário será direcionado à tela da Figura 12, onde será possível criar uma nova conta com *e-mail* e senha que deve ser confirmada.

Na Figura 13 é possível observar como todas as funcionalidades que foram planejadas no início estão sendo contempladas. Na seção de *upload*, é possível definir um nome para o dado, e escolher o arquivo que será submetido. Além disso, a tabela presente nessa tela mostra todos os dados que foram submetidos, com o *link* para *download* e também a opção de remover os existentes.

Semelhante à tela referente aos dados, a Figura 14 mostra o componente de gerenciamento das ferramentas. Também devem ser selecionados um arquivo, um nome, os

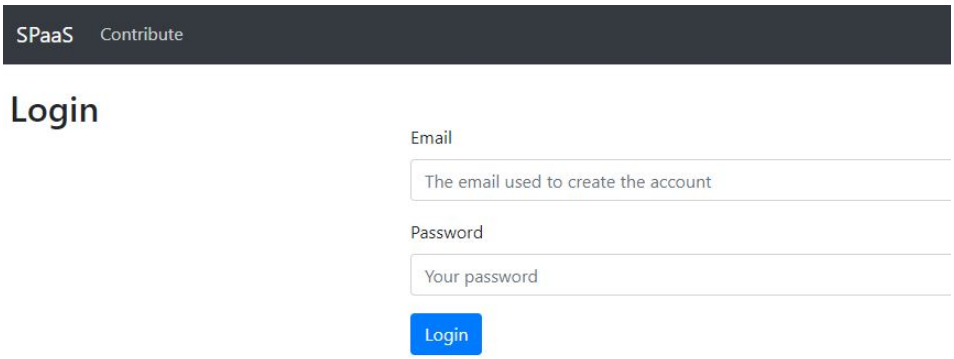


Figura 11: Tela final do *login*

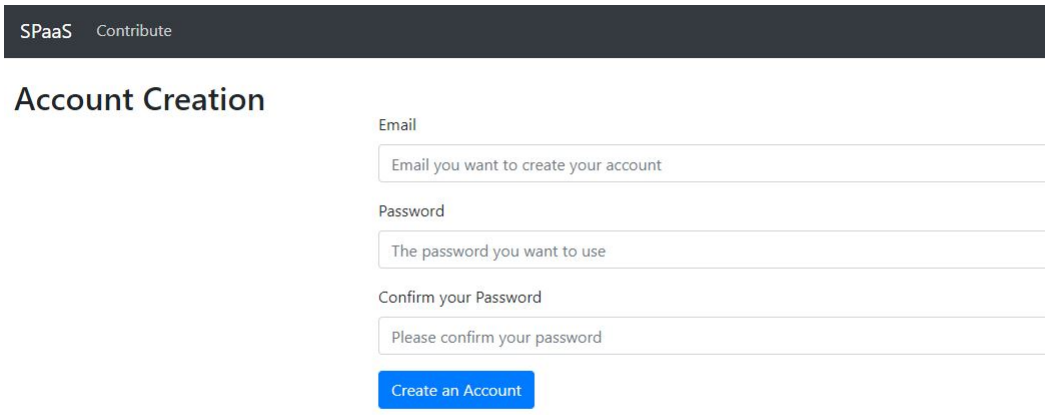


Figura 12: Tela final de criação de contas

parâmetros necessários para a execução e suas explicações. Os parâmetros são submetidos da seguinte maneira: 1:explicação do primeiro parâmetro, 2:explicação do segundo parâmetro, etc.

A Figura 15 mostra como é a definição de um pedido de processamento. Nesse primeiro momento, só é possível escolher uma ferramenta e o dado que serão usados. Porém como mostra a Figura 16, após selecionar a ferramenta, serão automaticamente exibidos os parâmetros necessários para a execução dessa tarefa. Esses argumentos foram submetidos juntamente com a ferramenta na tela representada pela Figura 14.

A tela de resultados, ilustrada pela Figura 17, mostra como podemos obter os resultados depois de terminados os processamentos. Além disso, caso o usuário queira verificar quais foram os dados, ferramenta e argumentos utilizados, o botão de *details*, pode ser usado para gerar um Json com as características daquele processamento, como ilustrado na Figura 18.

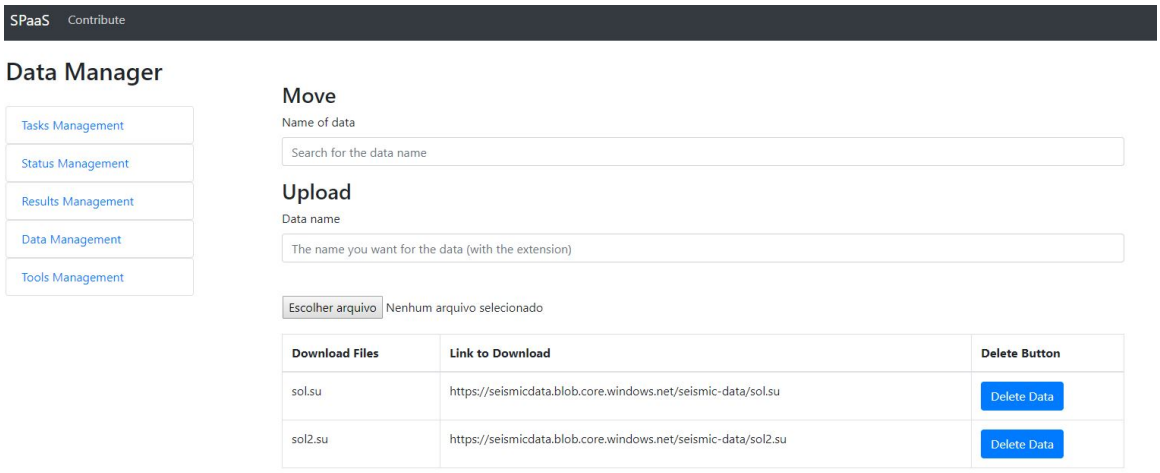


Figura 13: Tela final de gerenciamento de dados sísmicos

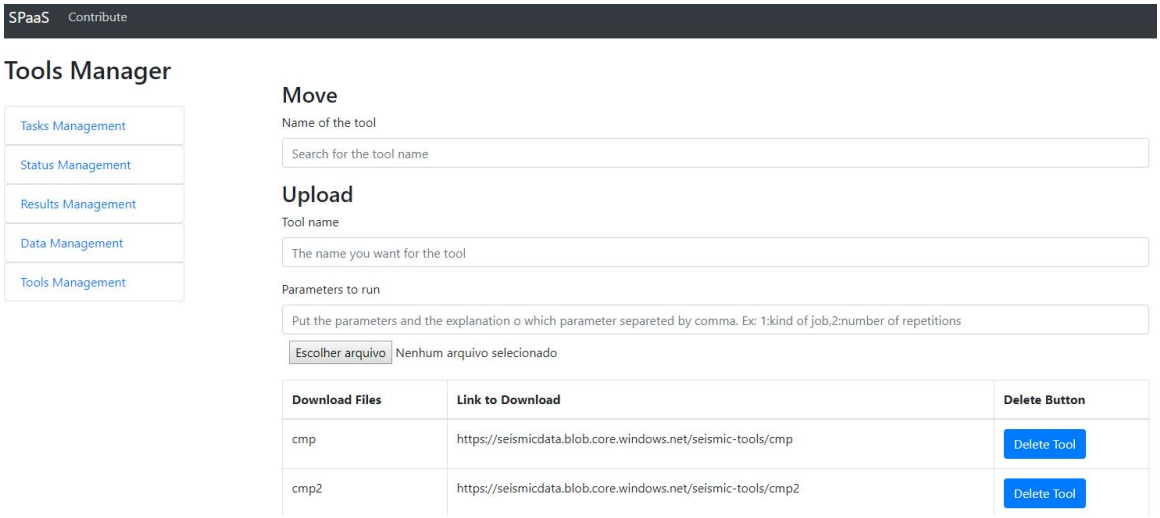


Figura 14: Tela final de gerenciamento de ferramentas

6 Conclusão

Ao final do processo de implementação da plataforma, é possível notar que os principais objetivos foram alcançados. Foi criada uma aplicação *web* somente com componentes *open source*, com todo o código fonte aberto desde o início no Github. Todo o *front end* foi desenvolvido com o *framework* Angular, o *back end* construído com Python, Celery e Flask. Além disso, usamos o Redis como fila de trabalho para alcançar o objetivo de desacoplar o *front end* do *back end*. Para a camada de armazenamento, usamos o Blob do Microsoft Azure e o banco de dados não relacional MongoDB. Para executar um processamento sísmico, o usuário final não necessita de conhecimento de computação em nuvem, o que também era

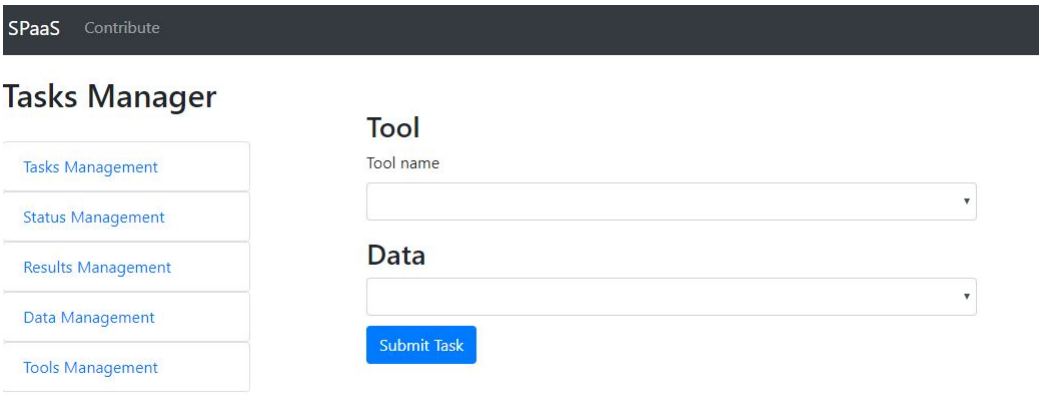


Figura 15: Tela final de gerenciamento de processamentos

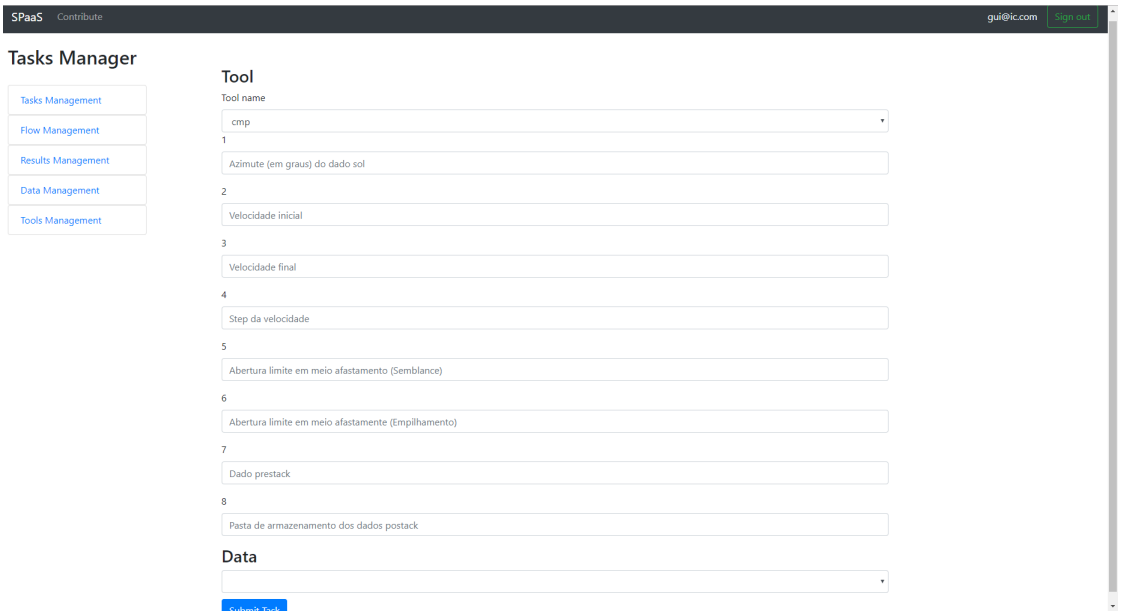


Figura 16: Tela final de processamento detalhada

uma das premissas iniciais do projeto. Executamos a prova de conceito completa no Microsoft Azure. Vale ressaltar que uma pessoa não participante do projeto conseguiu executar o sistema seguindo as instruções desse relatório. Como sugestão para projetos futuros, é desejado utilizar os resultados dos processamentos como entrada em outros processamentos. Além disso, existe o plano de implementar uma maneira do usuário conseguir analisar as imagens sísmicas geradas na própria interface *web* e a implementação de um módulo responsável pelo fluxo de processamento.

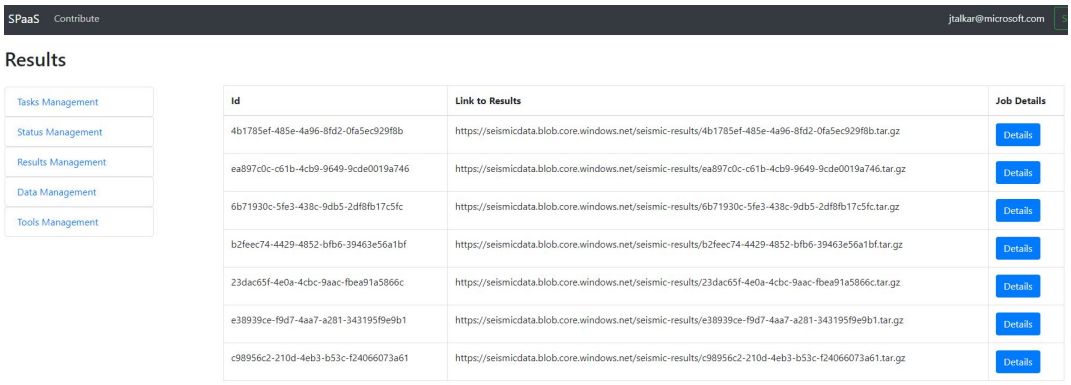


Figura 17: Tela final de gerenciamento de resultados

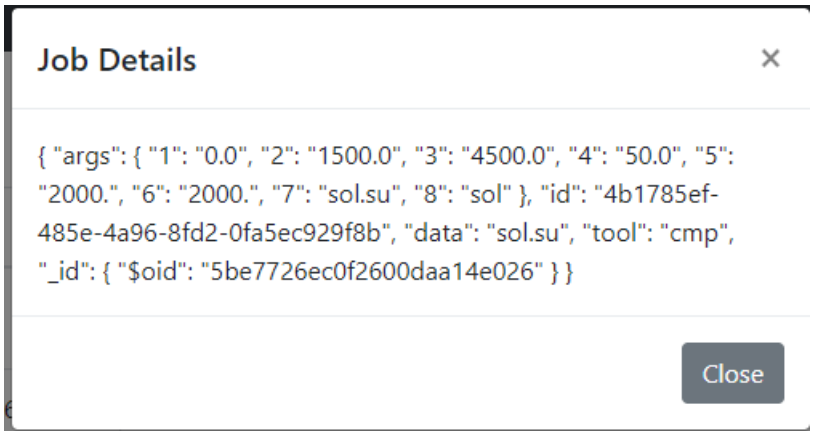


Figura 18: Tela final com detalhes do processamento nos resultados

Referências

[1] A. Back and H. Lindén. (2015). *Cloud Computing Security: A Systematic Literature Review,* Uppsala University, Uppsala, Sweden.

[2] B. Burns. (2017). *Designing Distributed Systems,* Sebastopol, CA: O'Reilly Media, Inc.

[3] A. Verma, L. Pedrosa et al. (2015). *Large-scale cluster management at Google with Borg,* The European Conference on Computer Systems.