



Plataforma de processamento de dados sísmicos como serviço em Nuvem

G. L. da Silva

E. Borin

Relatório Técnico - IC-PFG-18-01

Projeto Final de Graduação

2018 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

Plataforma de processamento de dados sísmicos como serviço em Nuvem

Guilherme Lucas da Silva

Edson Borin

Resumo

Este trabalho busca criar uma plataforma que facilita o processamento de dados sísmicos na nuvem, sem exigir do usuário conhecimento técnico dos novos conceitos que a nuvem traz consigo. A partir disso, 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.

Desta forma, o projeto foi iniciado com o objetivo de criar uma plataforma *open source*, simples e replicável para qualquer usuário. Os resultados foram muito positivos, alcançando o intuito de criar a plataforma somente com componentes *open source* e replicável em diversas situações.

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*):** essa é a categoria que dá mais responsabilidade ao usuário. Essa exige que o desenvolvedor gerencie de maneira integral máquinas virtualizadas, instale programas necessários, defina configurações e etc. Exemplo disso são máquinas virtuais que rodam sistemas operacionais GNU/Linux e são acessadas remotamente.
- **PaaS(*Platform as a Service*):** Aqui, o usuário não precisa se preocupar com pacotes de sistema, *softwares* e configurações. Esse sabor de computação em nuvem, tenta facilitar para que desenvolvedores consigam publicar suas aplicações. Isso significa que se, por exemplo, um usuário quiser expor uma aplicação, não é necessário

instalar compiladores e pacotes necessários para rodar o programa naquela determinada linguagem.

- **SaaS(*Software as a Service*):** essa é a modalidade que traz menos autonomia para o desenvolvedor, sendo que, através dela, todo o recurso é gerenciado pelo provedor de nuvem que é consumido. Um exemplo que podemos citar para ilustrar são 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 o 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 adquirir máquinas físicas, podemos citar:

- **Custo:** devido à possibilidade de pagar somente pelo que está usando, ou seja, caso algum recurso esteja sendo pouco aproveitado, basta desligá-lo. Usuários desse tipo de plataforma não gastarão para manter sistemas parados.
- **Escalabilidade:** A computação em nuvem permite que, caso a aplicação precise de mais poder computacional do que possui no momento, seja simples aumentar o número de instâncias ou o tamanho das máquinas, para atender mais usuários, gerando assim, mais receita.
- **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 atualmente temos a Microsoft Azure¹, Amazon Web Services² e Google Cloud Platform³, oferecendo opções diversas para computação em nuvem, desde máquinas virtuais até mesmo bancos de dados gerenciados.

Alto desempenho computacional, ou HPC⁴, é um conceito que busca agregar poder de computação, de forma a oferecer um desempenho muito maior do que o encontrado em computadores comuns. Um exemplo de uma aplicação que exige alto poder computacional é o processamento de dados sísmicos. Várias dessas aplicações são feitas em servidores 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 provedores de nuvem citados acima.

2 Objetivos

Esse trabalho tem como objetivo desenvolver uma plataforma *open source* que torne simples para usuários que não conhecem conceitos de computação em nuvem executarem suas aplicações nesse tipo de ambiente. O desejado para o final da plataforma é que os usuários

¹Plataforma de computação em nuvem da Microsoft <https://azure.microsoft.com/>

²Plataforma de computação em nuvem da Amazon <https://aws.amazon.com/>

³Plataforma de computação em nuvem do Google <https://cloud.google.com/>

⁴do inglês *High-Performance Computing*

consigam processar os trabalhos sísmico sem necessariamente conhecer plataformas de nuvem. Ao final, é esperado a criação de uma plataforma *web*, que tenha como principais funcionalidades:

- **Submissão e gerenciamento de dados:** essa funcionalidade é a responsável pelo *upload* de dados sísmicos que serão usados nos processamentos, além de consultar quais já foram submetidos, baixá-los e também excluí-los, caso necessário.
- **Submissão e gerenciamento de *Tools*:** esse componente do projeto é semelhante ao que detalhamos acima, porém, ao invés dos dados sísmicos, os artefatos que são gerenciados são as *Tools* das aplicações que serão utilizados nos processos.
- **Definição de tarefas sísmicas:** utilizando os dados e *Tools* submetidas a partir das duas funcionalidades citadas anteriormente, o objetivo é conseguir lançar um processamento que combina os dados e *Tools*, 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

Ao buscar por outras soluções que têm o mesmo intuito da plataforma que esse trabalho tem por objetivo desenvolver, notamos a sua relevância, uma que vez que é difícil achar outros programas *open source* que executam tal tarefa. Além da unicidade que o trabalho possui no cenário atual, ele se torna relevante já que ele pode 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 Definições

- **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. Por exemplo: um dado sísmico que foi carregado pelo usuário.
- **Tool:** ferramenta que é capaz de processar um dado do usuário. Por exemplo: uma ferramenta (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 haja a necessidade de intervenção manual ao término da execução de cada ferramenta. A Figura XX 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 de interesse. 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.

5 Arquitetura do Sistema

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 estava aberto no Github⁵, já que foi uma premissa desde o início do projeto que este seria *open source*.

Ao iniciar a análise do problema, foi possível notar que uma arquitetura de nuvem que se encaixa nesse problema 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]. Isso é o que foi buscado, uma vez que os trabalhos eram intensivos, demoravam um tempo significativo para serem executados e eram independentes um do outro. Assim, o que gostaríamos era que trabalhos fossem submetidos a uma fila e, alguma unidade de computação o usasse para executar uma tarefa.

A Figura 1 ajuda a explicar como é o funcionamento dessa arquitetura e o desenho inicial do projeto.

A plataforma foi separada em duas grandes partes:

- **Front End:** a parte que o usuário realmente vê, o código que irá rodar no navegador do seu dispositivo.
- **Back End:** essa é a porção da plataforma responsável por características que são invisíveis ao usuário. Entre elas, podemos citar acesso à camada de armazenamento, autenticação, submissão de dados para a nuvem, etc.

Existem alguns componentes que foram usados em comum entre essas duas partes:

⁵Serviço para compartilhamento de código <https://github.com/Guilhermeslucas/SPaaS>

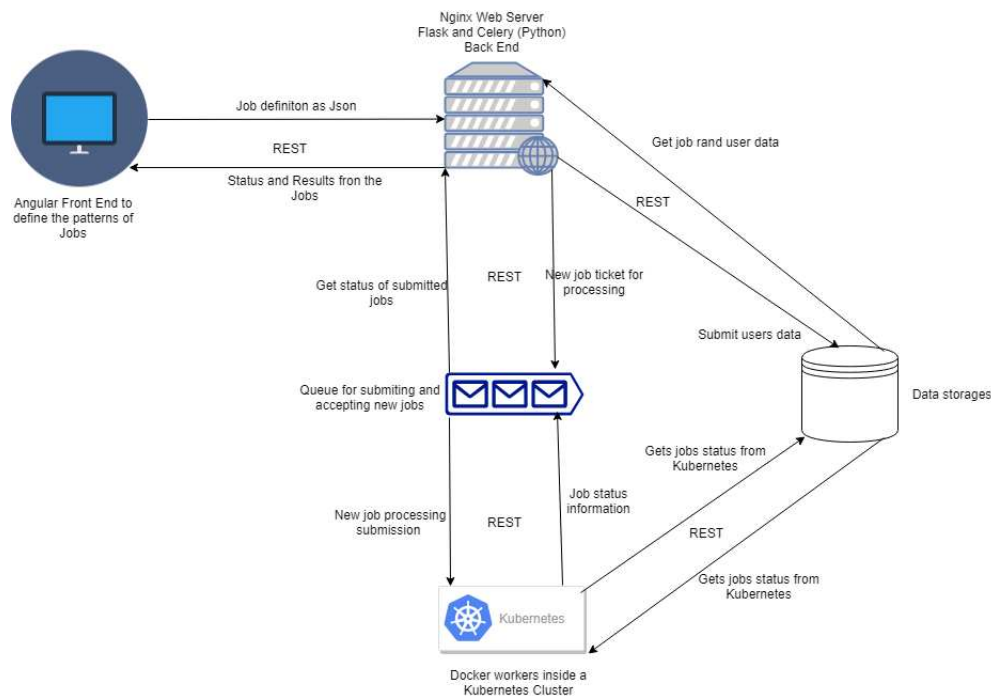


Figura 1: Arquitetura inicial do projeto e exemplo de filas da trabalho distribuídas

- **Docker⁶:** uma maneira muito simples de empacotar as 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.
- **NgInx⁷:** servidor *web open source* escalável e fácil de configurar. 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 ter aplicações rodando.

5.1 Projeto Inicial

Para o desenvolvimento do *front end*, foram definidas de antemão como seria a composição geral das telas e o fluxo da aplicação. Assim, a visão geral das telas, foi definida da seguinte maneira:

A Figura 2 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 que foram previamente cadastrados.

⁶Programa que realiza virtualização em nível de Sistema Operacional, também conhecido como contêinerização <https://www.docker.com/>

⁷Servidor Web *open source* <https://www.nginx.com/>

SPaaS		User	Sign out
Login			
Email			
<input type="text"/>			
Password			
<input type="password"/>			

Figura 2: Tela de Login

SPaaS		User	Sign out
Create Account			
Email			
<input type="text"/>			
Password			
<input type="password"/>			
Confirm Password			
<input type="password"/>			

Figura 3: Tela de criação de conta

A Figura 3 deixa claro quais foram as informações que eram requeridas para se criar uma conta no sistema, que eram *e-mail* senha e confirmação de senha.

A Figura 4 exemplifica como imaginamos a tela de definição de um trabalho para ser executado. Nesse primeiro protótipo, foi pensado que somente as informações ilustradas na tela, eram o suficiente. Nessa tela os campos necessários eram o nome do processamento, qual o código seria executado, o comando necessário para rodar a aplicação e o dado sísmico.

O protótipo da tela de estado dos trabalhos está ilustrada na Figura 5. Simples a princípio, uma tabela mostrando o estado de cada trabalho.

A Figura 6 mostra como foi pensada a obtenção dos resultados.

SPaaS Borin Sign out

Definitions

Menu

- Definition
- Status
- Results

Jobs name

Code

Running Command

Job Data

Figura 4: Tela de definição dos trabalhos

SPaaS Borin Sign out

Status

Menu

- Definition
- Status
- Results

JobID	Used Time	Status
1	10 minutes	Completed
2	2 minutes	Enqueued
3	4 minutes	Processing
4	10 minutes	Completed

Figura 5: Tela de observação do estado

Vale ressaltar que essa ideia inicial foi pensada com poucas funcionalidades para a plataforma. Assim, o fluxo de todo o trabalho definitivo foi repensado para algo mais próximo do representado na Figura 7. Nesse fluxo definido, os quadrados representam as telas, com suas respectivas interações com as camadas de armazenamento que são necessárias para completar suas tarefas com sucesso. Assim, detalhando cada módulo da solução:

- **Data Management:** módulo responsável pelo gerenciamento dos dados sísmicos da plataforma. A intenção nesse módulo é permitir que o usuário possa gerir os dados na plataforma, sendo capaz de adicionar, remover e movê-los de um tipo de armazenamento mais caro (porém mais rápido) para um mais barato, e mais lento.
- **Tools Management:** esse módulo procura realizar o gerenciamento das *Tools* que serão executados. Assim como o módulo responsável pelo gerenciamento dos dados

SPaaS

Borin

Sign out

Results

Menu

Definition

Status

Results

JobID	Results File	Total value
1	result_1.json	12
2	result_2.json	14
3	result_3.json	14
4	result_4.json	28

Figura 6: Tela de obtenção dos resultados

sísmicos, esse módulo busca dar a possibilidade de adicionar, mover e excluir as *Tools* da plataforma.

- **Running Tasks:** nessa tela, é desejado que o usuário consiga ver qual é o estado dos processamentos que iniciou. Estados como Executando e Pronto são alguns dos possíveis.
- **New Tasks:** o usuário usa esse módulo para criar as tarefas que ele deseja rodar. Os atributos necessários para essa definição são *Tool*, dado sísmico e argumentos para a execução acontecer com sucesso.
- **Results:** os resultados dos processamentos executados estarão aqui, organizados pela processamento que o gerou.

Assim, as atividades que um usuário deve realizar para que um processamento completo seja executado são os 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.
2. **Login:** Após a criação da conta citada no item anterior, o usuário usará seu *e-mail* e senhas cadastradas para entrar na plataforma.
3. **Submissão do dado sísmico:** uma vez que o usuário foi autenticado na plataforma, ele deverá submeter um dado sísmico que deseja para a execução do processamento.
4. **Submissão da *Tool*:** além do dado, também é necessário a submissão de uma *Tool* para o processamento.

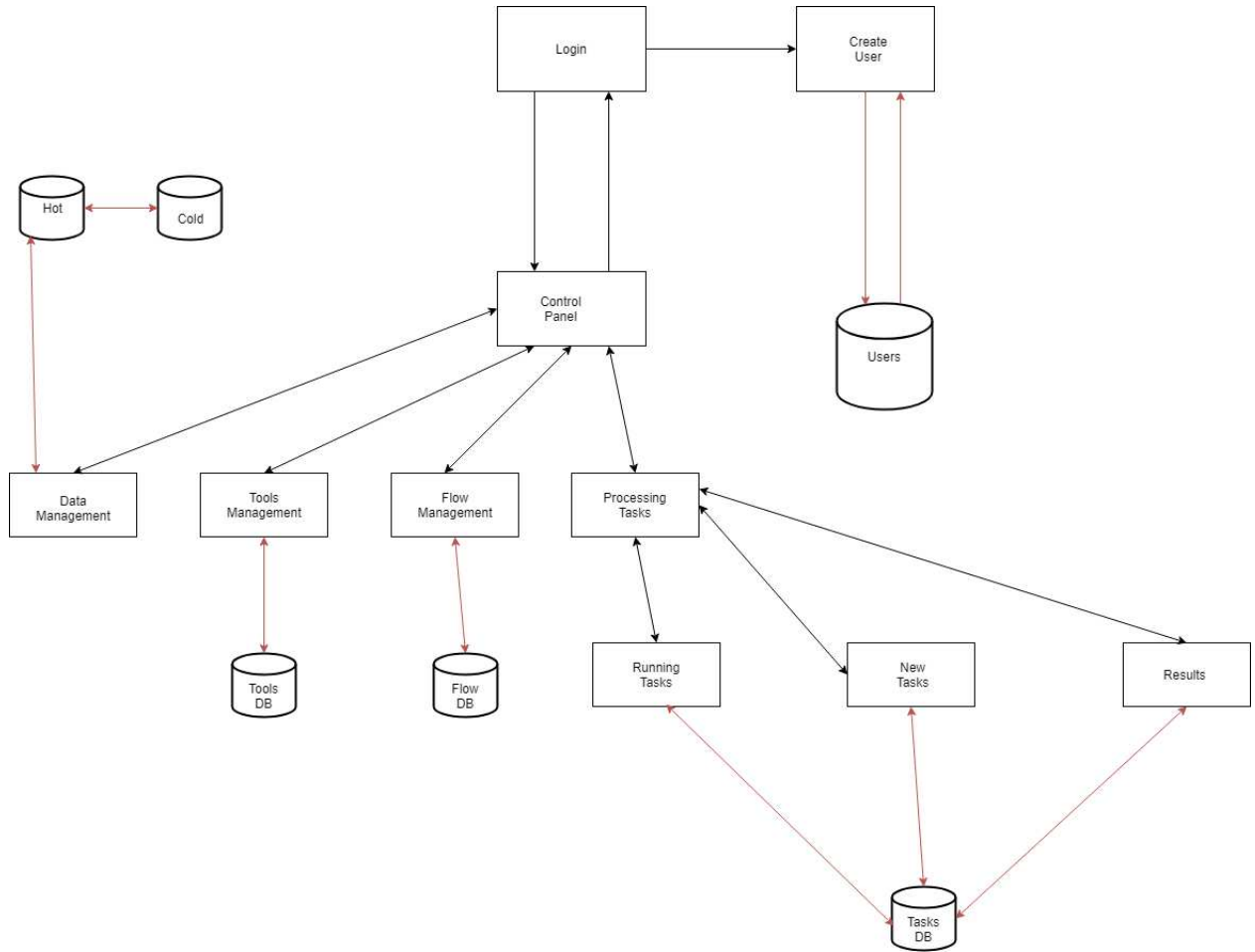


Figura 7: Fluxo entre os componentes da plataforma

5. **Criação da tarefa:** com o dado sísmico e a *Tool* submetidas, o usuário deve escolher qual o dado sísmico e *Tool* 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 rodando e esperar até ficar pronta, para coletar os resultados.
7. **Obtenção dos resultados:** ao final, o usuário poderá consultar os resultados que foram gerados. Todos os arquivos gerados estão reunidos em um arquivo disponível para *download*.

5.2 Front End

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

- **Angular**⁸: *framework* para desenvolvimento de aplicações *web open source*, desenvolvido inicialmente pelo Google e agora também mantido pela comunidade. Usa como linguagem de desenvolvimento o Typescript⁹. A escolha desse *framework* aconteceu pois ele é bastante produtivo e completo, já encapsulando todo o conceito de serviços, estilo de página e linguagem de marcação, roteamento, etc. Assim, com esse *framework* ainda é necessário conhecer linguagens de marcação como HTML e CSS, mas a junção de todos esses conceitos fica mais simples.
- **Bootstrap**¹⁰: *framework* que possui o estilo de vários componentes *web* prontos, como tabelas, menus laterais, barras superiores, etc. É o que muitos *sites* usam hoje em dia, como é o caso do Github. Estilizar páginas *web* pode se tornar um trabalho complicado e demorado, por isso optamos por usar um *framework*.

Foi garantido que o *deploy* fosse feito da maneira mais simples e flexível para o cliente. Então, foi usado o servidor *web* Nginx para isso. Além disso, também foi disponibilizado um Dockerfile, que possibilita o *deploy* da solução de maneira simples e direta sobre contêineres Docker.

5.3 Back End

Ao início do desenvolvimento do *back end*, buscamos escolher quais seriam as melhores tecnologias para esse componente, além de definir como seriam as rotas e contratos que a API do *back end* iria expor para que o *front end* pudesse consumir de forma eficaz e direta, garantindo assim, o desacoplamento entre os dois componentes. Assim, os componentes que escolhemos para o *back end* foram os seguintes:

- **Python**¹¹: linguagem criada por Guido Van Rossum, muito presente nos dias de hoje devido a sua extensa gama de aplicações, que vão desde embarcados até sistemas de análise de dados de alto desempenho. A escolha dela foi devido principalmente a bibliotecas maduras para desenvolvimento de APIs *back end*, filas de trabalho distribuídas e conectores com bancos de dados.
- **Flask**¹²: *framework* para desenvolvimento de APIs para Python. Foi escolhido devido a agilidade para colocar uma API rodando. Além disso, é simples entender como a aplicação está funcionando, fator muito crítico para decidirmos a escolha desta, uma vez que o projeto tem como premissa ser *open source*. Foram cogitadas também ASP.NET com C# e NodeJS. A primeira foi descartada devido a complexidade para criarmos uma API. NodeJS não foi escolhido devido a dificuldade que novos programadores podem enfrentar para compreender as funcionalidades em um primeiro momento, quando necessitassem aumentar o projeto.

⁸Framework *open source* para desenvolvimento *web* <https://angular.io/>

⁹Linguagem criada pela Microsoft. <https://www.typescriptlang.org/>

¹⁰Conjunto de ferramentas *open source* que ajuda na estilização de páginas <https://getbootstrap.com/>

¹¹<https://www.python.org/>

¹²Framework *web open source* para Python <http://flask.pocoo.org/>

- **Celery**¹³: uma vez escolhido 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. O Celery é um componente do Python que é o responsável por gerenciar, submeter e obter resultados de uma fila de trabalho, de uma maneira muito simples. Além disso, dá suporte a vários componentes de mensageria, como por exemplo, RabbitMQ¹⁴, Redis, Amazon SQS¹⁵.
- **Redis**¹⁶: após a escolha de qual biblioteca iríamos usar para facilitar a execução das filas de trabalho, que foi o Celery, foi preciso escolher o componente para a mensageria 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 com os três. Porém, Amazon SQS é uma opção dependente a um provedor de nuvem, o que complica o desenvolvimento do projeto, caso seja preciso usar outra *cloud* diferente da AWS. RabbitMQ e Redis ofereciam funcionalidades muito semelhantes, porém o Redis foi escolhido, devido à sua documentação e suas outras funcionalidades que podem ser aproveitadas também, como, por exemplo, *cache* e armazenamento de dados em memória.
- **MongoDB**¹⁷: para o armazenamento dos dados de cadastro de usuários, informações sobre argumentos de cada *Tool* e etc., a escolha foi um banco de dados NoSQL, o MongoDB. A decisão deste foi natural, uma vez que os dados que estavam trafegando entre um serviço e outro tinham uma estrutura muito semelhante com os documentos que são guardados nesse tipo de banco, não sendo necessário se preocupar em transformar os dados para os esquemas presentes em tabelas SQL.
- **Blob Storage**¹⁸: o Blob Storage da Microsoft Azure foi selecionado pois era necessário uma espécie de armazenamento de propósito geral. Aqui ele está sendo usado para armazenamento dos dados sísmicos e *Tool*. Foi escolhido devido à integração que possui com Python. Outras opções existentes eram, por exemplo, Amazon S3¹⁹, que oferece serviços semelhantes. Outra opção era deixar esses dados no disco de máquinas virtuais e compartilhá-los por meio de servidores, porém, isso aumentaria a complexidade e dificultaria a tarefa de gerenciar esse tipo de infraestrutura.

¹³*Framework open source* para tarefas distribuídas em Python <http://www.celeryproject.org/>

¹⁴Plataforma *open source* de Mensageria <https://www.rabbitmq.com/>

¹⁵Sistema de mensageria da plataforma AWS <https://aws.amazon.com/sqs/>

¹⁶Armazenamento de dados em memória *open source* <https://redis.io/>

¹⁷Banco de Dados NoSQL *open source* <https://www.mongodb.com/>

¹⁸Armazenamento de caráter genérico no Microsoft Azure <https://azure.microsoft.com/en-us/services/storage/blobs/>

¹⁹Armazenamento genérico na AWS <https://aws.amazon.com/pt/s3/>

5.4 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 independente do modo que foi escolhido para isso, foi criada uma série de *scripts* em ferramentas específicas para ajudar. Existem dois modelos que estão sendo facilitados:

- **Kubernetes**²⁰: orquestrador de *containers* famoso nos dias de hoje, que tem como principal funcionalidade tornar mais simples, escalável e performático o *deploy* de aplicações baseadas em *containers*. Nascida 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 milhares de diferentes aplicações entre uma série de *clusters*, atualmente está sob a jurisdição da Cloud Native Computing Foundation²¹ e é *open source*. Foi uma das plataformas que foi visada por ter apresentado relevância nos últimos tempos, com todos os grandes provedores de nuvem oferecendo opções gerenciadas dessa ferramenta.
- **Ansible**²²: 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 rode com sucesso estejam instaladas. Nesse aspecto, o Ansible se mostra eficaz. Essa ferramenta surgiu da Red Hat e também é mantida *open source*, sendo um ótimo gerenciador de configurações, o que torna possível escrever configurações como código, garantindo que a aplicação sempre terá sucesso para ser executada.

6 Resultados

6.1 Arquitetura

Após seguir todo o experimento citado acima, foi alcançado um resultado muito próximo do esperado. A arquitetura final do projeto, está ilustrada na Figura 8.

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 durante o procedimento, ou seja, 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 servido 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.

²⁰Orquestrador de *containers open source* <https://kubernetes.io/>

²¹Cloud Native Computing Foundation <https://www.cncf.io/>

²²Gerenciador de configuração *open source* <https://www.ansible.com/>

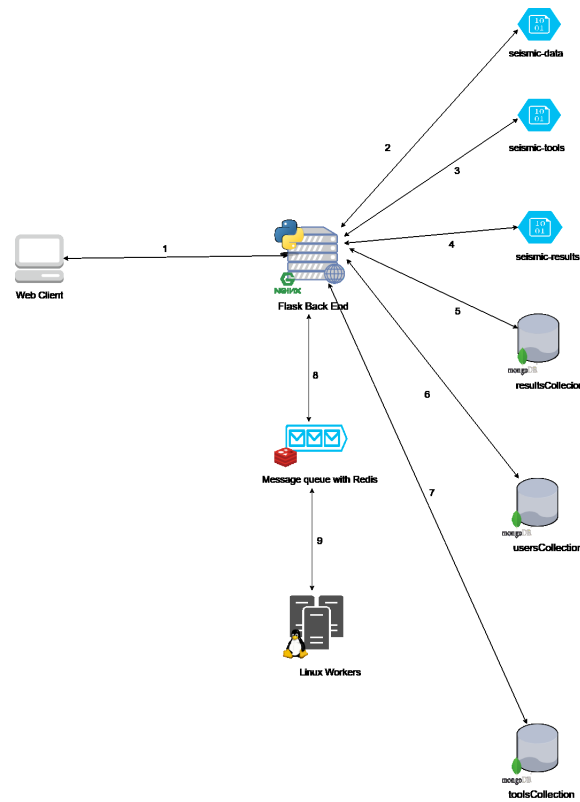


Figura 8: Arquitetura final do projeto

- **Linux Workers:** componente que é responsável pela execução propriamente dita dos pedidos de trabalho. Com o auxílio do Celery, esperam por 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.
- **Message queue with Redis:** 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 obterá essa requisição, realizando o processamento necessário. Aqui vale uma observação importante: A escolha do Celery e do modelo de filas de trabalhos distribuídas, que já foi explicada anteriormente, torna possível, além de desacoplar o *back end* dos workers, a execução de processamentos em paralelo e distribuídos, não criando demora na solução, possivelmente ocasionado pelo grande número de pedidos na fila.
- **toolsCollection:** coleção dentro do banco que guarda informações sobre as *Tools* que estão armazenadas. Essas informações são nome da *Tool* e seus argumentos. Para cada argumento, é armazenado seu nome e descrição. Novos dados são inseridos nela quando uma nova *Tool* é armazenada, quando um novo pedido de processamento é

```

{
  "_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 afastamento (Empilhamento)",
      "name" : "6"
    },
    {
      "description" : "Dado prestack",
      "name" : "7"
    },
    {
      "description" : "Pasta de armazenamento dos dados postack",
      "name" : "8"
    }
  ],
  "name" : "cmp"
}

```

Figura 9: Exemplo de um documento presente na coleção de *Tools*

incluído e também quando uma *Tool* é excluída. Um exemplo de descrição de uma *Tool* que segue o modelo esperado está mostrado na Figura 9.

```

{
  "_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 10: Exemplo de um documento presente na coleção de resultados

- **resultsCollection:** coleção de documentos no banco que armazena os dados dos resultados, armazena o id do processamento, dado e *Tool* usada, além dos argumentos. Essa coleção é acessada após o término de cada processamento, para salvar as informações do resultado e também para listar resultados. Um exemplo de documento que segue o modelo esperado está ilustrado na Figura 10.
- **usersCollection:** 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 um novo usuário. Para que todos esses processos sejam executados com sucesso, um exemplo de

```
{
  "_id" : ObjectId("5bb1464ec0f26814a6c8fc70"),
  "password" : "123",
  "email" : "guilhermepic.com"
}
```

Figura 11: Exemplo de um documento presente da coleção de usuários

documento armazenado com as informações dos usuários está ilustrado na Figura 11.

- **seismic-tools:** Blob usado para armazenar as *Tools* submetidas pelos usuário. Acesado pelo componente de gerenciamento de *Tools* quando uma nova é inserida na plataforma e quando um novo processamento acontece.
- **seismic-data:** Blob que armazena os dados sísmicos submetidos. Também são acesados 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.gz²³ e disponibilizados para *download*, no componente de resultados.

Vale notar aqui que, usamos somente uma instância de bancos de dados, com coleções diferentes para cada tipo de 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.

6.2 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*:

6.2.1 Front End

Para rodar 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

```
$ ng serve
```

é usado. Assim, a única variável que deve ser alterada é a *apiUrl*, que deve apontar para o *back end*. Assim para compilar o projeto e servi-lo usando um servidor *web*, os comandos que devem ser rodados são os seguintes:

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

²³utilitário de compressão de dados <https://www.gzip.org/>

Isso gerará uma pasta `./dist/out` que, quando colocadas 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 rodar o *front end* é via Docker. Existe um Dockerfile dentro da pasta *front end* que facilita o *deploy*. Existem dois comandos que devem ser rodados nesse caso, um para construir a imagem do Docker e outro para rodá-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 mostrada.

6.2.2 Back End

Aqui, assim como no *front end*, também existe a possibilidade de rodar 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 de `requirements.txt` mostra quais são os pacotes necessários para rodar 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 *Tools*.
- `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, basta, dentro da pasta *back end*, rodar os seguintes comandos:

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

Caso a opção seja por rodar 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 escutando na porta 5000.

6.2.3 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
```

6.3 Telas finais

Após apresentar a arquitetura final, ilustramos os componentes, com suas respectivas telas.

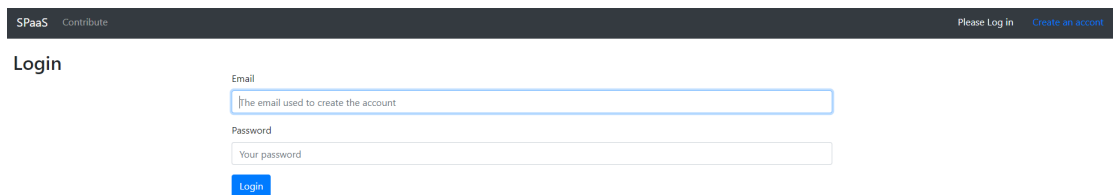


Figura 12: Tela final do *login*

Na tela final de *login*, Figura 12, existem dois campos: *e-mail* e senha. Além disso, caso você não possua uma conta, é possível criar no canto superior direito.

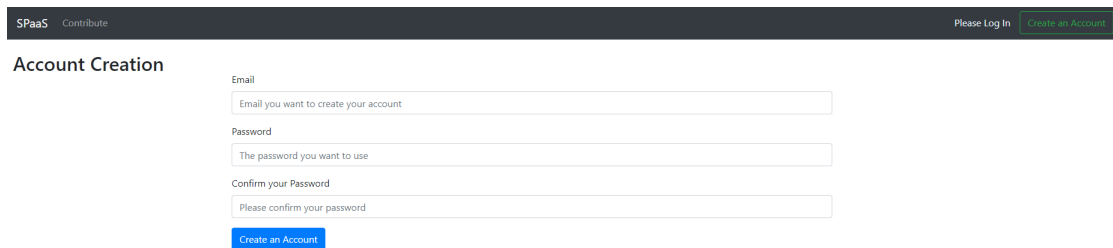


Figura 13: Tela final de criação de contas

Ao escolher a opção de criar uma conta, o usuário será direcionado à Figura 13, onde será possível criar uma nova conta com *e-mail* e senha, que deve ser confirmada.

Na Figura 14 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 abaixo mostra todos os dados que foram submetidos, com o *link* para *download* e também a opção de deletar os já existentes.

Semelhante a tela referente aos dados, a Figura 15 mostra o componente de gerenciamento das *Tools*. Também devem ser selecionado um arquivo, um nome e quais os parâmetros necessários para a execução, juntamente com suas explicações. Os parâmetros são submetidos da seguinte maneira: 1:explicação do parâmetro 1, 2:explicação do parâmetro 2, etc.

The screenshot shows the SPaaS Data Manager interface. On the left is a sidebar with navigation links: Tasks Management, Flow Management, Results Management, Data Management (selected), and Tools Management. The main content area is titled 'Data Manager' and contains two sections: 'Move' and 'Upload'. The 'Move' section has a 'Name of data' label and a search input field. The 'Upload' section has a 'Data name' label and an input field with a placeholder 'The name you want for the data (with the extension)'. Below the input fields is a file selection area with a button 'Escolher arquivo' and the text 'Nenhum arquivo selecionado'. At the bottom, there is a table with three columns: 'Download Files', 'Link to Download', and 'Delete Button'. The table contains one row with the file 'sol.su', its download link 'https://seismicdata.blob.core.windows.net/seismic-data/sol.su', and a 'Delete Data' button.

Figura 14: Tela final de gerenciamento de dados sísmicos

The screenshot shows the SPaaS Tools Manager interface. On the left is a sidebar with navigation links: Tasks Management, Flow Management, Results Management, Data Management, and Tools Management (selected). The main content area is titled 'Tools Manager' and contains two sections: 'Move' and 'Upload'. The 'Move' section has a 'Name of the tool' label and a search input field. The 'Upload' section has a 'Tool name' label and an input field with a placeholder 'The name you want for the tool'. Below the input fields is a 'Parameters to run' section with a text area for parameters and a button 'Escolher arquivo' with the text 'Nenhum arquivo selecionado'. At the bottom, there is a table with three columns: 'Download Files', 'Link to Download', and 'Delete Button'. The table contains one row with the file 'cmp', its download link 'https://seismicdata.blob.core.windows.net/seismic-tools/cmp', and a 'Delete Tool' button.

Figura 15: Tela final de gerenciamento de *Tools*

The screenshot shows the SPaaS Tasks Manager interface. On the left is a sidebar with navigation links: Tasks Management (selected), Flow Management, Results Management, Data Management, and Tools Management. The main content area is titled 'Tasks Manager' and contains two sections: 'Tool' and 'Data'. The 'Tool' section has a 'Tool name' label and a dropdown menu. The 'Data' section has a 'Data' label and a dropdown menu. Below the dropdown menus is a blue 'Submit Task' button.

Figura 16: Tela final de gerenciamento de processamentos

A Figura 16 mostra como é a definição de um pedido de processamento. Nesse primeiro momento, só é possível escolher uma *Tool* e o dado que serão usados, porém como mostra a Figura 17, após selecionar a *Tool*, serão automaticamente carregados quais são os argumentos necessários para a execução dessa tarefa. Esses argumentos foram submetidos juntamente com a *Tool* na tela representada pela Figura 15.

Na tela de resultados, ilustrada pela Figura 18, fica claro como podemos obter os resul-

SPaaS

Contribute

gui@ic.com

Sign out

Tasks Manager

Tasks Management

Flow Management

Results Management

Data Management

Tools Management

Tool

Tool name

cmp

1

Azimute (em graus) do dado sol

2

Velocidade inicial

3

Velocidade final

4

Step da velocidade

5

Abertura limite em meio afastamento (Semblance)

6

Abertura limite em meio afastamento (Enpiilhamento)

7

Dado prestack

8

Pasta de armazenamento dos dados postack

Data

Completed

Figura 17: Tela final de processamento detalhada

SPaaS

Contribute

gui@ic.com

Sign out

Results

Tasks Management

Flow Management

Results Management

Data Management

Tools Management

Id	Link to Results	Job Details
4b1785ef-485e-4a96-8fd2-0fa5ec929f8b	https://seismicdata.blob.core.windows.net/seismic-results/4b1785ef-485e-4a96-8fd2-0fa5ec929f8b.tar.gz	<div>Details</div>
ea897c0c-c61b-4cb9-9649-9cde0019a746	https://seismicdata.blob.core.windows.net/seismic-results/ea897c0c-c61b-4cb9-9649-9cde0019a746.tar.gz	<div>Details</div>

Figura 18: Tela final de gerenciamento de resultados

Job Details

X

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

Close

Figura 19: Tela final com detalhes do processamento nos resultados

tados depois de terminados os processamentos. Além disso, caso o usuário queira verificar quais foram os dados, *Tool* e argumentos que geraram aqueles resultados, é possível através

do botão de *details*, que gera o Json com as características daquele processamento, como ilustrado na Figura 19.

7 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 precisa de conhecimento de computação em nuvem, o que também era uma das premissas iniciais do projeto. Como próximos passos do projeto, é desejada a implementação de uma maneira dos resultados dos processamentos serem usados 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*.

Referências

- [1] A. Back and H. Lindén., *Cloud Computing Security: A Systematic Literature Review*, (2015).
- [2] B. Burns, *Designing Distributed Systems*, (2017).
- [3] A. Verma, L. Pedrosa et al., *Large-scale cluster management at Google with Borg*, (2015).