

# 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*, replicável e extremamente simples de ser usada para qualquer usuário. Os resultados foram muito positivos, alcançando o intuito de criar a plataforma somente com componentes *open source* e facilmente replicável em diversas situações.

## 1 Introdução

Computação em Nuvem é um conceito que está cada vez mais presente no cotidiano de todas as pessoas. Mesmo sem perceber, uma quantidade gigantesca 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 uma infinidade de serviços como máquinas virtuais, bancos de Dados e redes virtuais sem que o usuário tenha máquinas físicas com tais serviços instalados e configurados, ainda sendo possível, de maneira simples, expor tais serviços e mecanismos para vários usuários ao redor do mundo. 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 mais relevantes:

- **IaaS(*Infrastructure as a Service*):** essa é a maneira que dá mais responsabilidade ao usuário. Essa modalidade exige que o usuário 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 publicar uma aplicação, não é necessário se preocupar em instalar compiladores e pacotes necessários para rodar a aplicação 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. Alguns exemplos que podemos citar para ilustrar são bancos de dados como serviço, onde o usuário não se preocupa com nenhum tipo de infraestrutura, somente usa o endereço que o provedor cede para o usuário 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 está 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 muito mais poder computacional do que possui no momento, seja extremamente 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 totalmente em escrever suas aplicações, sem a preocupação de gerenciar infraestrutura e plataforma, o que pode ser muito trabalhoso.

Entre os principais provedores de nuvem pública atualmente temos a Microsoft Azure<sup>1</sup>, Amazon Web Services<sup>2</sup> e Google Cloud Platform<sup>3</sup>, oferecendo opções extremamente diversas para computação em nuvem, desde máquinas virtuais até mesmo bancos de dados gerenciados.

Aplicações de alta performance computacional, ou HPC<sup>4</sup>, são uma gama de programas que necessitam de muito poder computacional para completar seus objetivos. Entre essas tarefas, estão o processamento de dados sísmicos, que são obtidos de maneira bruta e precisam de um trabalho muito pesado de tratamento para que algumas conclusões possam ser feitas a partir deles. Segundo K. Paladugu e S. Mukka [2], supercomputação é usada para realizar operações matemáticas em alta velocidade. HPC é como supercomputação, mas com o poder computacional de *cluster* computacionais. Ainda hoje, muitas dessas operações são feitas em servidores em institutos de pesquisas, por exemplo, podendo levar a um custo excessivo, muito trabalho para gerenciamento de infraestrutura além de baixíssima agilidade e flexibilidade quanto à arquitetura. Assim, tais aplicações podem tirar muito proveito dos provedores de nuvem citados acima.

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

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

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

<sup>4</sup>do inglês *High-Performance Computing*

## 2 Objetivos

Esse trabalho tem como objetivo desenvolver uma plataforma *open source* que torne extremamente 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 era que os usuários consigam processar os trabalhos sísmico sem necessariamente conhecer plataformas de nuvem. Ao final, é esperado a criação de uma plataforma web, que tinha 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 foram submetidos, baixá-los e também excluí-los, caso necessário.
- **Submissão e gerenciamento de binários:** esse componente do projeto é extremamente semelhante com o que detalhamos acima, porém, ao invés dos dados sísmicos, os artefatos que são gerenciados são os binários das aplicações que serão utilizados nos processos.
- **Definição de tarefas sísmicas:** utilizando os dados e binários submetidos a partir das duas funcionalidades citadas anteriormente, o objetivo é conseguir lançar um processamento que combina os dados e binários, além dos argumentos necessários.
- **Obtenção dos resultados:** ao final dos processos, é desejado que se consiga obter todos os resultados desse de maneira simples e rápida.

## 3 Relevância

Ao buscar por outras soluções que tem o mesmo intuito da plataforma que esse trabalho tem por objetivo desenvolver, notamos a sua extrema relevância, uma que vez que é muito difícil achar outros programas *open source* que executam tal tarefa. Além da unicidade que o trabalho possui no cenários 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 muito mais inteligente dos recursos, baixando custos e aumentando a produtividade.

## 4 Arquitetura do Sistema

Para o sucesso do projeto, ao começo dele, foi decidido que faríamos uma plataforma *web*. Essa foi a decisão tomada devido à facilidade de desenvolver para esse tipo de plataforma, além do alcance gigantesco que plataformas *web* possuem, já que é praticamente obrigatório nos dispositivos que os usuários poderiam usar para acessar o sistema (celulares e computadores) a existência de um navegador de internet instalado. Além disso, vale ressaltar que

desde o início do projeto todo o código estava totalmente aberto no Github<sup>5</sup>, 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 muito bem 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. [3]. Isso é exatamente o que foi buscado, uma vez que os trabalhos eram intensivos, demoravam um tempo significativo para serem reproduzidos 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 definição de trabalho.

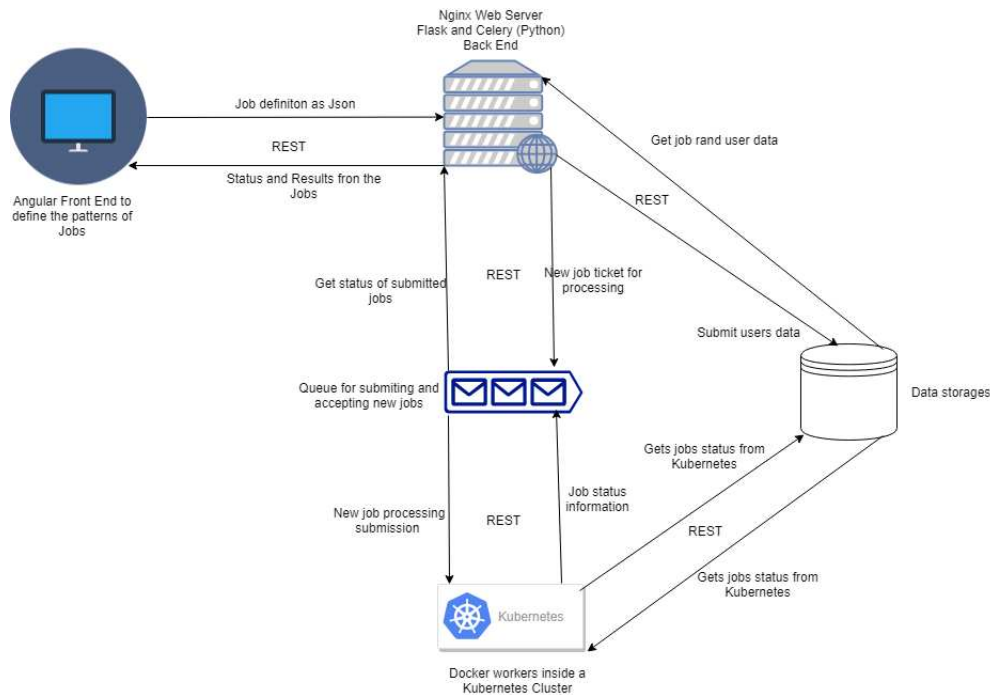


Figura 1: Arquitetura inicial do projeto e exemplo de filas de trabalho distribuídas

A Figura 1 ajuda a explicar como é o funcionamento dessa arquitetura e a arquitetura 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 ao banco de dados, autenticação,

<sup>5</sup>Serviço para compartilhamento de código fonte [www.github.com](http://www.github.com)

processamento de dados, submissão de dados para a nuvem, etc.

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

- **Docker**<sup>6</sup>: uma maneira muito simples de empacotar as aplicações, isolando dependências, garantindo que o *deploy* sempre é feito da maneira correta. Assim, é possível entregar as aplicações completas, com todos os requisitos necessários instalados, o que dá uma agilidade enorme durante o ciclo de desenvolvimento.
- **Nginx**<sup>7</sup>: servidor *web open source* extremamente 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.

#### 4.1 *Front End*

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

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

Figura 2: Tela de Login

A Figura 2 mostra como definimos quais informações deveriam estar na tela para que o usuário pudesse se conectar a plataforma. Como mostrado, as informações necessárias para entrar no sistema eram *e-mail* e senha que foram previamente cadastrados.

A Figura 3 deixa claro quais foram as informações que eram requeridas para se criar uma conta no sistema.

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

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

SPaaS		User	Sign out
-------	--	------	----------

### Create Account

Email

Password

Confirm Password

Figura 3: Tela de criação de conta

SPaaS	Borin	Sign out
-------	-------	----------

### Definitions

Menu
Definition
Status
Results

Jobs name

Code

Running Command

Job Data

Figura 4: Tela de definição dos trabalhos

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 mostradas na tela, eram o suficiente.

O protótipo da tela de status dos trabalhos está mostrada na Figura 5. Extremamente simples a princípio, uma tabela mostrando o status de cada trabalho.

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

Vale ressaltar que essa ideia inicial foi pensada com poucas funcionalidades que realmente eram relevantes para a plataforma. Assim, o fluxo de todo o trabalho 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, entrando no detalhe

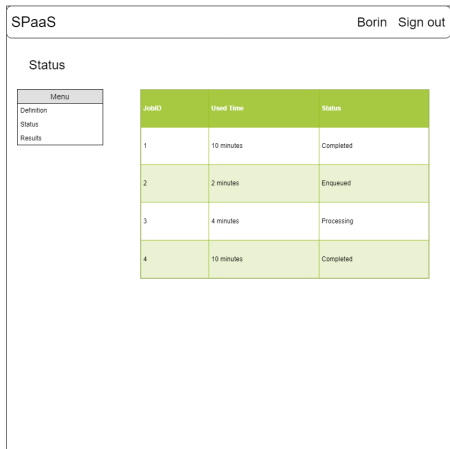


Figura 5: Tela de observação do status

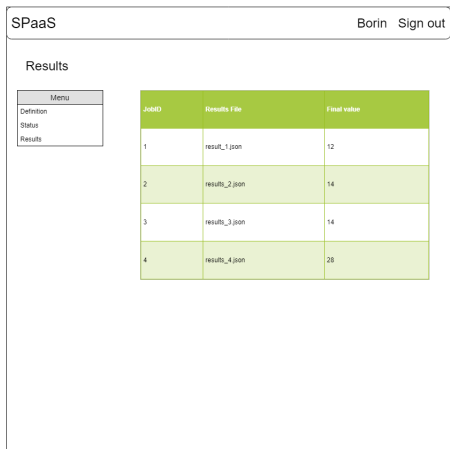


Figura 6: Tela de obtenção dos resultados

de cada componente da solução:

- **Data Management:** componente responsável pelo gerenciamento dos dados sísmicos da plataforma. A intenção nesse componente é ser capaz de adicionar, remover e movê-los de um tipo de armazenamento mais caro (porém mais rápido) para um mais barato, ligeiramente mais lento.
- **Flow Management:** componente responsável por determinar uma sequencia de passos que devem ser feitos para completar uma tarefa (*tasks*). Diferente das tarefas, aqui a definição é de um conjunto de *tasks*, encadeadas para obter um resultado maior.
- **Tools Management:** essa funcionalidade procura realizar o gerenciamento dos binários que serão executados. Assim como o componente responsável pelo geren-



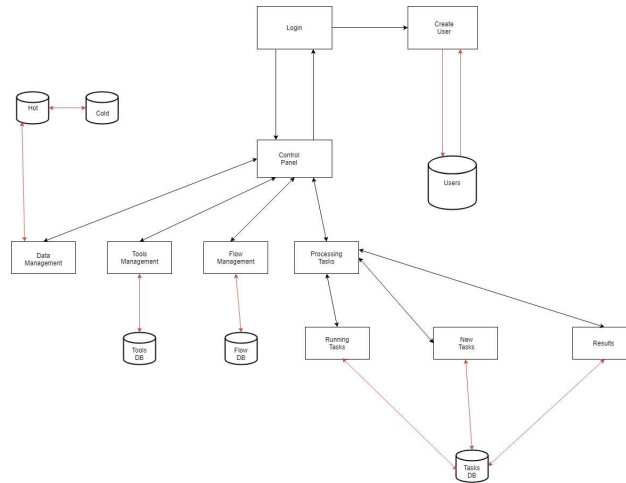


Figura 7: Fluxo entre os componentes da plataforma

ciamento dos dados, esse componente busca dar a possibilidade de adicionar, mover e excluir binários da plataforma.

- **Running Tasks:** nessa tela, é desejado que o usuário consiga ver qual é o status dos processamentos que iniciou. Status como pendente e pronto são alguns dos possíveis.
- **New Tasks:** responsável por definir novas tarefas a serem rodadas. Os atributos necessários para essa definição são binário, dado sísmico e argumentos para a execução acontecer com sucesso.
- **Results:** os resultados dos processamentos executados estarão aqui, organizados pela execução que o gerou.

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

- **Angular<sup>8</sup>:** *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<sup>9</sup>. A escolha desse *framework* aconteceu uma vez que ele é extremamente produtivo e por ser 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 muito mais simples.
- **Bootstrap<sup>10</sup>:** *framework* que possui o estilo de vários componentes web prontos, como tabelas, menus laterais, barras superiores, etc. É o que muitos sites usam

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

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

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

hoje em dia, como é o caso do Github. Estilizar páginas web pode ser um trabalho extremamente 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 extremamente simples e direta sobre contêineres Docker.

## 4.2 Back End

Ao início do desenvolvimento do *back end*, buscamos definir 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, um total desacoplamento entre os dois componentes. Assim, os componentes que decidimos para o *back end* foram os seguintes:

- **Python**<sup>11</sup>: linguagem criada por Guido Van Rossum, extremamente relevante nos dias de hoje devido a sua enorme gama de aplicações, que vão desde embarcados até sistemas de análise de dados de alto desempenho. A escolha dela foi devido principalmente a bibliotecas muito maduras para desenvolvimento de API's *back end*, filas de trabalho distribuídas e conectores com bancos de dados.
- **Flask**<sup>12</sup>: framework para desenvolvimento de APIS para Python. Foi escolhido devido a extrema agilidade para colocar uma API rodando. Além disso, é muito 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* e facilitar o entendimento de todos que o usam. Ao invés de Python/Flask, foram cogitadas também ASP.NET com C# e NodeJS. A primeira foi descartada devido a complexidade para simplesmente 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.
- **Celery**<sup>13</sup>: uma vez escolhido Python e Flask como ferramentas para o desenvolvimento da solução, foi muito natural a escolha do Celery como ferramenta para filas de trabalho. Existem muitos casos de sucesso espalhados pela internet de engenheiros e desenvolvedores que usaram essa combinação com total 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, RabbitMQ<sup>14</sup>, Redis, Amazon SQS<sup>15</sup>.

---

<sup>11</sup><https://www.python.org/>

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

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

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

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

- **Redis**<sup>16</sup>: após a escolha de qual biblioteca iríamos usar para facilitar a tarefa de filas de trabalho, que foi o Celery, foi preciso escolher algum 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 qualquer um desses três. Porém, Amazon SQS é uma opção presa a um provedor de nuvem, o que complica o andar do projeto, caso seja preciso usar outra *cloud* diferente da AWS. RabbitMQ e Redis ofereciam funcionalidades muito parecidas, porém o Redis foi escolhido, devido a infinidade de materiais sobre o uso dele e suas outras funcionalidades que podem ser tiradas proveito também, como, por exemplo, *cache* e armazenamento de dados em memória.
- **MongoDB**<sup>17</sup>: para o armazenamento dos dados de cadastro de usuários, informações sobre argumentos de cada tipo de binário e etc., a escolha foi um banco de dados NoSQL, o MongoDB. A escolha deste foi muito direta, uma vez que os dados que estavam trafegando entre um serviço e outro tinham uma estrutura muito parecida 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, o que agiliza muito o trabalho.
- **Blob Storage**<sup>18</sup>: o Blob Storage da Microsoft Azure foi a opção aqui pois era necessário uma espécie de armazenamento de propósito geral. Aqui ele está sendo usado para armazenamento dos dados sísmicos e binários. Foi escolhido devido a excelente integração que possui com Python. Outras opções existentes eram, por exemplo, Amazon S3<sup>19</sup>, que oferece serviços muito parecidos. Outra opção era deixar esses dados em no disco de máquinas virtuais e servi-los por meio de servidores, porém, isso aumentaria muito a complexidade e dificultaria a tarefa de gerenciar esse tipo de infraestrutura.

### 4.3 Ferramentas para *Deploy*

Como dito anteriormente, ao iniciar o projeto, não era desejado que estivéssemos presos a um modelo específico 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**<sup>20</sup>: orquestrador de containêres extremamente famoso nos dias de hoje, que tem como principal funcionalidade tornar mais simples, escalável e performática

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

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

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

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

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

o *deploy* de aplicações baseadas em *containers*. Nascida dentro do Google, a partir do Borg, que segundo o paper publicado pelo Google [4], 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 Foundation<sup>21</sup> e é *open source*. Foi uma das plataformas que foi visada por ter apresentado extrema relevância nos últimos tempos, com todos os grandes provedores de nuvem oferecendo opções gerenciadas dessa ferramenta.

- **Ansible<sup>22</sup>:** um modelo muito comum de *deploy* de aplicações é baseada 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 extremamente 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.

## 5 Resultados

### 5.1 Arquitetura

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

Dada a arquitetura final do projeto mostra na Figura 8 os componentes mostrados 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.
- **Linux Workers:** componente que é responsável pela execução propriamente dita dos pedidos de trabalho. Com o auxílio do Celery, espera 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*

---

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

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

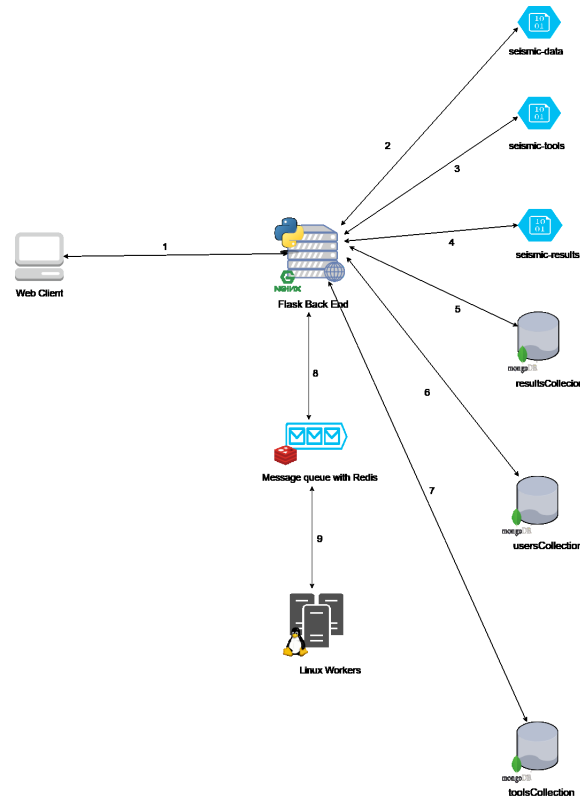


Figura 8: Arquitetura final do projeto

dos Linux *workers*. Assim O *back end* somente precisava submeter uma mensagem com as definições para a execução de um trabalho na fila e algum *worker* obteria 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 um gargalo na solução, possivelmente ocasionado pelo grande número de pedidos na fila esperando por um único *worker*.

- **toolsCollection:** coleção dentro do banco que guarda informações sobre os binários que estão armazenados. Novos dados são inseridos nela quando submetidos um novo binário é armazenado, quando um novo pedido de processamento será incluído, já que é necessário consultar quais os parâmetros para esse e também quando um binário é excluído, apagando o registro. O modelo que está sendo usado para guardar essas informações é o que está na Figura 9
- **resultsCollection:** coleção de documentos no banco para armazenar os dados do resultado, armazena o id do processamento, dado e binário usado, além dos argumentos. Essa coleção é acessada após o término de cada processamento, para salvar as

```
{
  "_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: Modelo presente na coleção de ferramentas binárias

```
{
  "_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: Modelo presente na coleção de resultados

informações do resultado e também quando é desejado que todos os resultados sejam mostrados na tela de gerenciamento de resultados. O modelo segue o padrão da Figura 10.

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

Figura 11: Modelo presente da coleção de usuários

- **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, o modelo usado é o da Figura 11.

- **seismic-tools:** Blob usado para armazenar os binários submetidos pelos usuário. Acessado pelo componente de gerenciamento de ferramentas quando uma nova ferramenta é inserida na plataforma e quando o processamento vai acontecer, pois nesse momento é necessário baixar esses dados para execução em um dos *workers*.
- **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, já que é necessário baixar esses dados.
- **seismic-results:** outro Blob. Esse, responsável por armazenar os resultados dos processamentos. Todos os resultados são empacotados em um arquivo tar.gz<sup>23</sup> e disponibilizados para *download*, no componente responsável por gerenciar os resultados.

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

## 5.2 Rodando a aplicação

Esta seção é dedicada à explicação de como executar cada componente da plataforma:

### 5.2.1 *Front End*

Para rodar o *front end*, todas as variáveis de ambiente estão no caminho 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
```

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*.

Outra maneira de rodar o *front end* é via Docker. Existe um Dockerfile dentro da pasta *front end* que facilita muito o *deploy*. Existem dois comandos que devem ser rodados nesse caso, um para construir a imagem do Docker e outro para rodá-la:

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

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

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

### 5.2.2 Back End

Aqui, assim como no *front end*, também existe a possibilidade de rodar o código nativamente ou dentro de um container Docker. Para rodar 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. Além desses dois arquivos, é preciso definir três variáveis de ambiente, que são:

- **SPASS\_CONNECTION\_STRING**: String de conexão do MongoDB que será usado para a aplicação.
- **SPASS\_DATA\_BLOB\_KEY**: Chave referente ao Blob que irá guardar todos os componentes citados anteriormente: resultados, dados e binários.
- **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\_DAT
```

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

### 5.2.3 Worker

Para os *workers*, também é possível rodá-lo nativamente. Para isso, dentro da pasta *back end*, basta executar:

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

Vale lembrar que nos *workers* também é necessário declarar as mesmas variáveis de ambiente do *back end*, citadas anteriormente. Porém, como a proposta é ter vários workers para executar as tarefas em paralelo, foi criado um *playbook* Ansible, para configurar deixar várias máquinas prontas para executar essas tarefas. Para isso, com o Ansible instalado e os *hosts* determinado no arquivo de configuração da ferramenta, basta executar dentro da pasta *devops*:

```
$ ansible-playbook worker-config.yaml
```



### 5.3 Telas finais

Assim, após apresentar a arquitetura final, são apresentados os componentes, com suas respectivas telas.

Figura 12: Tela final do *login*

Na tela final de *login*, Figura 12, existe dois campos muito simples e comuns em sites que necessitam autenticação: *e-mail* e senha. Além disso, caso você não possua uma conta, é possível criar uma no canto superior direito.

Figura 13: Tela final de criação de contas

Caso, na tela de *login*, for escolhido a tela que o usuário será direcionado é a Figura 13, onde será possível criar uma nova conta com *e-mail* e senha, que deve ser confirmada. Aqui, existe a verificação se o *e-mail* já foi usado, para garantir que cada usuário só tenha uma conta cadastrada.

Download Files	Link to Download	Delete Button
sol.su	https://seismicdata.blob.core.windows.net/seismic-data/sol.su	Delete Data

Figura 14: Tela final de gerenciamento de dados sísmicos

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.

**Tools Manager**

Tasks Management  
Flow Management  
Results Management  
Data Management  
Tools Management

**Move**  
Name of the tool  
Search for the tool name

**Upload**  
Tool name  
The name you want for the tool

Parameters to run  
Put the parameters and the explanation o which parameter separated by comma. Ex: 1:kind of job,2:number of repetitions

Escolher arquivo Nenhum arquivo selecionado

Download Files	Link to Download	Delete Button
cmp	https://seismicdata.blob.core.windows.net/seismic-tools/cmp	Delete Tool

Figura 15: Tela final de gerenciamento de binários

Extremamente parecida com a tela referente aos dados, a Figura 15 mostra a tela de gerenciamento dos binários que foram submetidos para a plataforma. Aqui também deve ser selecionado um arquivo, um nome e quais os parâmetros necessários para a execução, juntamente com suas explicações. Assim, os parâmetros são submetidos da seguinte maneira: 1:explicação do parâmetro 1, 2:explicação do parâmetro 2, etc.

**Tasks Manager**

Tasks Management  
Flow Management  
Results Management  
Data Management  
Tools Management

**Tool**  
Tool name

**Data**

Submit Task

Figura 16: Tela final de gerenciamento de processamentos

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

Na tela de resultados, representada pela Figura 18, fica claro como podemos obter os resultados depois de terminados com sucesso os processamentos. Além disso, caso o

SPaaSContribute

gui@ic.comSign 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 (Enpihlamento)

7

Dado prestack

8

Pasta de armazenamento dos dados postack

Data

Completed

Figura 17: Tela final de processamento detalhada

SPaaSContribute

gui@ic.comSign 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

usuário queira verificar quais foram os dados, binários e argumentos que geraram aqueles resultados, é possível através do botão de *details*, que gera o Json com as características

daquele processamento, como é possível ver na Figura 19.

## Referências

- [1] A. Back and Hugo Lindén., *Cloud Computing Security: A Systematic Literature Review*, (2015).
- [2] K. Paladugu and S. Mukka, *Systematic Literature Review and Survey on High Performance Computing in Cloud*, (2012).
- [3] B. Burns, *Designing Distributed Systems*, (2017).
- [4] A. Verma, L. Pedrosa et al., *Large-scale cluster management at Google with Borg*, (2015).