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

Vanderlei Munhoz Pereira Filho

Projeto e desenvolvimento de um sistema de software de alto desempenho para execução de competições de programação com números massivos de usuários

Florianópolis
2020

Vanderlei Munhoz Pereira Filho

Projeto e desenvolvimento de um sistema de software de alto desempenho para execução de competições de programação com números massivos de usuários

Esta monografia foi julgada no contexto da disciplina DAS5511: Projeto de Fim de Curso, e APROVADA na sua forma final pelo curso de Engenharia de Controle e Automação

Florianópolis, 3 de março de 2020.

Banca Examinadora:

_____
Flávia Dias de Carvalho
Orientadora na Empresa
IBM

_____
Prof. Márcio Bastos Castro, Dr.
Orientador no Curso
Departamento de Informática e Estatística

_____
Prof. Felipe Gomes de Oliveira Cabral, Dr.
Avaliador
Departamento de Automação e Sistemas

_____
Matheus Domingos da Silva e Silva
Debatedor
Departamento de Automação e Sistemas

_____
Gabriel José Prá Gonçalves
Debatedor
Departamento de Automação e Sistemas

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

Pereira Filho, Vanderlei Munhoz
Projeto e desenvolvimento de um sistema de software de alto desempenho para execução de competições de programação com números massivos de usuários / Vanderlei Munhoz Pereira Filho ; orientador, Márcio Bastos Castro, 2020.
141 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia de Controle e Automação, Florianópolis, 2020.

Inclui referências.

1. Engenharia de Controle e Automação. 2. computação em nuvem. 3. sistemas distribuídos. 4. virtualização de hardware. I. Castro, Márcio Bastos. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Vanderlei Munhoz Pereira Filho

Projeto e desenvolvimento de um sstema de software de alto desempenho para execução de competições de programação com números massivos de usuários

Esta monografia foi julgada no contexto da disciplina DAS5511: Projeto de Fim de Curso, e APROVADA na sua forma final pelo curso de Engenharia de Controle e Automação

Florianópolis, 3 de março de 2020.

Banca Examinadora:

_____
Flávia Dias de Carvalho
Orientadora na Empresa
IBM

_____
Prof. Márcio Bastos Castro, Dr.
Orientador no Curso
Departamento de Informática e Estatística

_____
Prof. Felipe Gomes de Oliveira Cabral, Dr.
Avaliador
Departamento de Automação e Sistemas

_____
Matheus Domingos da Silva e Silva
Debatedor
Departamento de Automação e Sistemas

_____
Gabriel José Prá Gonçalves
Debatedor
Departamento de Automação e Sistemas

“There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other is to make it so complicated that there are no obvious deficiencies.”
(C. A. R. Hoare, 1996)

RESUMO

Este documento detalha o projeto de fim de curso realizado pelo autor durante seu período de estágio na IBM, no contexto da disciplina obrigatória DAS5511 do curso de engenharia de controle e automação da UFSC. O projeto em questão envolve a concepção, desenvolvimento, teste e documentação de um sistema de software robusto, visando a substituição de um sistema protótipo já existente. O novo sistema, entregue ao fim deste trabalho, é capaz de solucionar o seguinte problema de negócio da empresa: realizar e supervisionar competições de programação com números massivos de participantes, em escala multinacional. Algumas especificações desafiadoras desse sistema são: tolerância a falhas com *zero-downtime deployment*, mínima latência possível para múltiplas requisições simultâneas, e controles rígidos de privacidade, em cumprimento com a GDPR Européia e LGPD Brasileira. Técnicas de computação paralela e programação concorrente são aplicadas para executar as tarefas mais computacionalmente intensivas, visando alcançar os requisitos de desempenho. A implementação de *proxies* e algoritmos de criptografia são adotadas para alcançar os requisitos de segurança e privacidade. Diferentes arquiteturas de implantação são analisadas desde a etapa de planejamento, até as etapas de implementação e teste, de forma a avaliar os custos e benefícios de cada uma. Ao final, testes comparativos, ou *benchmarkings*, são realizados no sistema desenvolvido, já em ambiente de produção, a fim de avaliar possíveis gargalos e o atendimento completo dos requisitos. Os resultados, em conjunto com uma análise crítica do trabalho desenvolvido como um todo, são apresentados nos últimos capítulos deste documento.

Palavras-chave: Computação paralela. Programação concorrente. Sistemas distribuídos. Servidores Web. Tolerância à falhas. Arquitetura de software. Integração de sistemas.

ABSTRACT

This document details the final project developed by the author during his internship at IBM, in the context of the required DAS5511 discipline of the UFSC automation and control engineering course. The project in question involves the development, testing and documentation of a robust software system, that will serve as a replacement for an existing prototype system. The new system, delivered by the time of publication of this document, is capable of solving the following business problem of the company: holding and overseeing programming competitions with massive numbers of participants on a multinational scale. Some challenging specifications of this system are: fault tolerance with zero downtime deployment, minimum possible latency for concurrent requirements, and strict privacy controls in compliance with the European GDPR and the Brazilian LGPD. Parallel computing and concurrent programming techniques are applied to perform the most computationally intensive tasks, enabling the system to achieve performance requirements. The implementation of proxies and the adoption of encryption algorithms are made to meet security and privacy requirements. Different deployment architectures are analyzed since the planning steps, until the implementation and testing of the system, studying the costs and benefits of each architecture. Lastly, comparative tests, or benchmarks, are performed in the complete system, already in the production environment. Possible bottlenecks are evaluated, as well as if the system completely fulfills all requirements established at the planning phase. The results, together with a critical analysis of the work as a whole, are presented in the last chapters of this document.

Keywords: Parallel computing. Concurrent programming. Distributed systems. Web servers. Fault tolerance. Software architecture. Systems integration.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE QUADROS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Este documento detalha o projeto de fim de curso realizado pelo autor durante seu período de trabalho na IBM, no contexto da disciplina obrigatória “Projeto de Fim de Curso” (DAS5511) do curso de Engenharia de Controle e Automação da Universidade Federal de Santa Catarina.

1.1 SOBRE A ESTRUTURAÇÃO DESTE DOCUMENTO

Esta monografia foi organizada em sete capítulos, conforme descritos a seguir:

a) No Capítulo 1 – Introdução – é apresentada de maneira introdutória a empresa onde o autor realizou as atividades do projeto de fim de curso, e o projeto como um todo: o problema atacado, os objetivos, e as motivações;

b) no Capítulo 2 – Fundamentação Técnica e Teórica – são apresentadas as tecnologias aplicadas na construção dos sistemas propostos e os conceitos teóricos relevantes à compreensão integral deste trabalho;

c) no Capítulo 3 – Problemática do Projeto – são detalhados os problemas apresentados inicialmente no Capítulo 1, incluindo o funcionamento dos sistemas usados pela empresa, seus pontos críticos, e uma prévia das soluções propostas pelo autor;

d) no Capítulo 4 – Planejamento e Formalização da Solução Proposta – é apresentada uma descrição completa do processo de planejamento e formalização das especificações da solução proposta, assim como o detalhamento do método de desenvolvimento de *software* adotado. Finalizando, é apresentada a arquitetura do sistema proposto pelo autor para atender aos requisitos formalizados;

e) no Capítulo 5 – Desenvolvimento – são descritos os processos de desenvolvimento, teste, e implantação dos módulos do sistema, apresentando detalhes de cada um dos componentes desenvolvidos e as suas integrações;

f) no Capítulo 6 – Análise do Sistema, Testes e Resultados – são apresentados os resultados gerais deste trabalho, incluindo os métodos de testes comparativos e métricas de desempenho alcançadas pelo sistema desenvolvido;

g) no Capítulo 7 – Considerações Finais – são apresentadas propostas de soluções alternativas e eventuais melhorias a serem realizadas.

1.2 SOBRE A EMPRESA

A IBM *(International Business Machines)* é uma corporação multinacional dos Estados Unidos da América fundada em 1888 e incorporada em 1911, com sede na

cidade de Armonk, do estado de Nova Iorque. Com uma história que remonta ao século XIX, a corporação dominou o mercado de Tecnologia da Informação por décadas, e suas principais atividades permanecem centradas na área da computação e de sistemas integrados (GREULICH, 2014). Atualmente a IBM é líder mundial no desenvolvimento e comercialização de *mainframes*[1], e também na pesquisa e desenvolvimento de tecnologia de ponta em diversas áreas, como inteligência artificial e fabricação avançada de semicondutores. Em 2012 a corporação empregava cerca de 434 mil funcionários no mundo todo – número que foi reduzido gradativamente nos anos subsequentes até atingir 380 mil empregados em 2018, espalhados em 170 países (SIMSON, 2009).

A IBM Brasil foi a primeira sucursal da corporação a estabelecer-se fora dos Estados Unidos da América, no ano de 1917 – na época, ainda com o antigo nome de *Computing-Tabulating-Recording Company* (CTR). A IBM Brasil é uma das maiores unidades da multinacional, com grandes centros de pesquisas instalados nas cidades de São Paulo e Rio de Janeiro, focados nas áreas de compreensão textual e exploração de petróleo. Além dos centros de pesquisa, a IBM Brasil possui diversas unidades comerciais por todo o país, inclusive na cidade de Florianópolis.

Na última década, dentro do contexto de globalização e transferência do eixo industrial para a Ásia, a IBM buscou deslocar o pivô de suas atividades comerciais, focando em mercados mais lucrativos e de alto valor agregado. Isso incluiu a venda de diversas subsidiárias focadas em manufatura para corporações asiáticas – por exemplo a linha de computadores pessoais ThinkPad, que foi vendida para a gigante chinesa Lenovo em 2004 (CNET, 2004) – e também a compra de empresas de outros setores, como firmas de consultoria (PwC Consulting, adquirida pela IBM em 2002 (COMPUTERWORLD, 2002)), e companhias de *software* especializadas em ciência de dados (SPSS, adquirida pela IBM em 2009 (WSJ, 2009)). Em 2013 a diretoria da IBM anunciou que a corporação iria gradualmente desfazer-se de suas tradicionais atividades de manufatura, mantendo apenas as unidades responsáveis pela fabricação de componentes de alta tecnologia (COMPUTERWORLD, 2014).

A partir de 2010, o mercado de serviços de computação em nuvem tornou-se um dos grandes campos de batalha das gigantes de tecnologia do mundo ocidental, como Amazon, Google e Microsoft. A competição de empresas asiáticas, como a Fujitsu do Japão e a Baidu da República Popular da China, também é crescente nesse mercado. Embora a IBM tenha entrado nessa disputa tardiamente, a corporação realizou diversas aquisições que vêm ampliando a sua fatia de mercado no ramo. Em 2013 a IBM adquiriu a SoftLayer por 2 bilhões de dólares (FORBES, 2019), e em 2019 foi fechada a aquisição da Red Hat por 34 bilhões de dólares – a maior aquisição da

[1] Computadores poderosos projetados para atividades específicas, geralmente utilizados por organizações governamentais ou indústrias que necessitam de grande poder computacional.

história do mundo do *software* – cementando a posição da IBM como maior provedora de serviços de computação em nuvem corporativa do mundo (REUTERS, 2019).

O novo eixo econômico da IBM passou então a basear-se em pesquisa e desenvolvimento, consultoria tecnológica, e prestação de serviços relacionados à Tecnologia da Informação, sobretudo o fornecimento de serviços de computação em nuvem. Em 2018 a IBM registrou o número recorde de 9.100 patentes no *United States Patent and Trademark Office*[2], marcando o vigésimo sexto ano consecutivo em que a corporação é a campeã em números de patentes emitidas. Metade das patentes de 2018 registradas pela IBM são avanços pioneiros em inteligência artificial, computação em nuvem, cibersegurança, *blockchain*, e computação quântica (FORTUNE, 2019).

A área interna na qual o autor deste trabalho realizou as atividades do projeto de fim de curso é conhecida como *IBM Cognitive Applications Latin America* (sediada na unidade da IBM Brasil em São Paulo – Figura 1). Esse setor da corporação é englobado pela divisão internacional de computação em nuvem, e é responsável pela concepção e implementação de provas de conceito para produtos e sistemas inteligentes (ou "cognitivos" de acordo com o vocabulário utilizado comumente pela empresa), nos quais faz-se o uso amplo de algoritmos de aprendizado de máquina e técnicas de aprendizado supervisionado em geral. Os funcionários da *IBM Cognitive Applications* trabalham próximos de empresas clientes e desenvolvedores de *software*, realizando um trabalho de mentoria e disseminação de conhecimento acerca das soluções de inteligência artificial, tecnologias de ponta, e serviços comercializados pela IBM através da IBM Cloud[3], além de contribuírem em projetos de *software* de código-aberto que recebem suporte da IBM.

Dentre os diversos produtos e serviços comercializados por meio da plataforma de computação em nuvem da IBM, existem dezenas de arcabouços conceituais[4] para o desenvolvimento e treinamento de modelos capazes de emularem capacidades cognitivas humanas. Por exemplo, um destes arcabouços é o *Watson Knowledge Studio* (WKS), uma ferramenta voltada para a criação de modelos anotadores de texto, capazes de identificar entidades e relações textuais em diversos tipos de documentos não-estruturados. Esses tipos de ferramentas servem o propósito de facilitarem e reduzirem custos de pesquisa e desenvolvimento, possibilitando que soluções complexas de inteligência artificial sejam rapidamente desenvolvidas por organizações com recursos limitados, sejam eles técnicos ou materiais.

[2] Agência do Departamento de Comércio dos Estados Unidos que emite patentes para inventores e empresas, incluindo o registro de marcas para identificação de produtos e propriedade intelectual.

[3] Plataforma de computação em nuvem da IBM lançada em 2013, inicialmente conhecida como IBM Bluemix e rebatizada para IBM Cloud em 2017.

[4] A palavra "*framework*" da língua inglesa é casualmente utilizada em vez da variante portuguesa, entretanto essa palavra estrangeira também é usada para designar outros termos (como bibliotecas de *software*). Neste trabalho o termo refere-se a arcabouços conceituais.

Figura 1 – Edifício da IBM em São Paulo, projetado pelo escritório de arquitetura Aflalo & Gasperini em 1977.

Fonte: Arquivo pessoal.

1.3 APRESENTAÇÃO DO PROJETO

A *IBM Cognitive Applications* da América Latina organiza diversas competições de programação voltadas para serviços de computação em nuvem e inteligência artificial. Essas competições são baseadas em desafios e problemas reais, enfrentados por diversas empresas e negócios da atualidade. Tais competições são baseadas em serviços de computação em nuvem oferecidos pela IBM, e são um esforço valioso para a corporação em um contexto de disseminação de conhecimento acerca de seus produtos.

Um exemplo de competição do tipo funciona da seguinte forma: um desafio é anunciado e os participantes recebem acesso a um repositório com o enunciado, dados, e códigos-fontes necessários para a execução da solução. Alguns arcabouços conceituais disponibilizados na IBM Cloud são utilizados, e podem ser acessados com uma conta gratuita. No geral, os desafios envolvem o treinamento de um modelo a partir de um conjunto de dados fornecido pela organização do evento, utilizando diversas ferramentas da IBM Cloud. Quando um usuário cria um modelo com as ferramentas da IBM Cloud – por exemplo, o *Watson Knowledge Studio* – ele é capaz de expor o modelo criado via API[5] automaticamente. Uma URL[6] e chaves de acesso são geradas, e qualquer aplicação com acesso à *Internet* pode facilmente integrar o modelo exposto.

Para a execução de eventos do tipo, que são em maioria totalmente online com eventuais etapas finais presenciais, faz-se necessário o desenvolvimento de um sistema de avaliação automática customizado para cada tipo de desafio. Como solução dos desafios das competições, os participantes fornecerão o endereço Web e a chave de acesso para seus modelos, e o sistema desenvolvido realizará diversas chamadas para os modelos expostos por cada participante, comparando os resultados com um gabarito. Cada desafio pode utilizar tecnologias diferentes, e os algoritmos de pontuação automática devem ser cuidadosamente elaborados.

É a partir do desejo de expansão de iniciativas do tipo descrito, inicialmente para outros países da América Latina, que surge o tema deste projeto: a criação de um sistema robusto capaz de atender uma quantidade massiva de participantes, com rastreamento de métricas e ferramentas para supervisão dos eventos futuros.

1.4 OBJETIVOS

O objetivo central deste trabalho é o **projetar e desenvolver um sistema para execução e gerenciamento de novas competições de programação do tipo descrito na Seção 1.3**. O novo sistema deverá ser capaz de atender múltiplas requisições de um grande número de usuários simultaneamente, e com baixa latência. Para o desenvolvimento deste projeto será adotado um método de desenvolvimento de *software* baseado no método *cleanroom* adaptado, detalhado no Capítulo 4. Para alcançar os requisitos de desempenho propostos serão estudados o uso de técnicas como computação paralela, programação concorrente, e diversas otimizações descritas no Capítulo 5. Toda a etapa de planejamento e documentação do sistema também faz parte deste trabalho, incluindo a realização dos testes de *software*. Com o sistema final já executando em ambiente de produção, serão realizados testes de carga a fim de

[5] "Interface de Programação de Aplicações", cujo acrônimo API provém da língua inglesa: *Application Programming Interface*.

[6] "Localizador Uniforme de Recursos" cujo acrônimo URL provém da língua inglesa: *Uniform Resource Locator*.

avaliar os requisitos atingidos. Os resultados finais são apresentados no Capítulo 6.

1.4.1 Objetivos gerais

No que diz respeito a objetivos gerais, tem-se:

a) Formalizar a etapa de planejamento e documentar completamente o sistema desenvolvido;

b) entregar um sistema de *software* integrado funcional que atenda aos requisitos formalizados na etapa de planejamento; e

c) executar testes comparativos de desempenho e avaliar criticamente a nova solução desenvolvida no contexto do sistema piloto.

1.4.2 Objetivos específicos

Quanto aos objetivos específicos, temos:

a) Formalizar o planejamento e as especificações dos sistemas desenvolvidos, conforme as recomendações do padrão internacional ISO/IEC/IEEE 29148:2018[7];

b) desenvolver e entregar um sistema de *software* funcional para pontuar automaticamente pelo menos um tipo de desafio (para este trabalho foi escolhido o desafio de compreensão de linguagem natural, dada a sua alta complexidade);

c) desenvolver e entregar um sistema de *software* funcional para gerenciar e analisar métricas sobre os eventos, com interface Web completa e camadas de acesso e segurança;

d) desenvolver e entregar um sistema de *software* funcional para tratar inscrições no evento, e também a submissão de soluções pelos participantes, com interface *Web* completa;

e) todos os sistemas desenvolvidos devem possuir mecanismos que os mantenham em cumprimento com a Lei Geral de Proteção de Dados Pessoais (LGPDP)[8] brasileira, e a *General Data Protection Regulation (GDPR)*[9] europeia;

f) implantar em ambiente de produção, integrar, e executar testes funcionais em todos os sistemas desenvolvidos; e

[7] Especificação da Organização Internacional de Normalização para a engenharia de *software*, ciclo de vida de processos e levantamento de requisitos. Disponível em `https://www.iso.org/standard/72089.html` (acessado em: 11/12/2019).

[8] Lei nº 13.709/2018, é a legislação brasileira que regula as atividades de tratamento de dados pessoais e que também altera os artigos 7º e 16 do Marco Civil da Internet.

[9] Refere-se ao regulamento do direito europeu sobre privacidade e proteção de dados pessoais, criado em 2018, aplicável a todos os indivíduos na União Europeia e Espaço Econômico Europeu.

g) executar testes comparativos de desempenho em ambiente de produção e avaliar criticamente a nova solução desenvolvida no contexto do sistema piloto.

1.5 RELEVÂNCIA E MOTIVAÇÕES

Devido à ótima recepção de eventos e competições de programação passadas, a IBM planeja escalar esses tipos de projetos para outros países da América Latina, e possivelmente em outros continentes no futuro. Para que esses futuros eventos ocorram de maneira excepcional, um sistema robusto que suporte uma grande quantidade de participantes sem prejuízo na experiência dos usuários torna-se essencial. Uma grande motivação é o potencial de que este novo sistema, cujo desenvolvimento será iniciado neste trabalho, torne-se realmente um sistema de *software* utilizado em produção, e não seja apenas um protótipo.

O tema do projeto em si é extremamente relevante atualmente, dado que técnicas aplicadas na solução criada neste trabalho podem ser exportadas para outros projetos. A otimização do uso de recursos computacionais é um tema de grande interesse econômico, visto que os esquemas de precificação de serviços de computação em nuvem são geralmente proporcionais ao processamento de informação por unidade de tempo. Ademais, a capacidade de atender a mais usuários com uma infraestrutura computacional reduzida é economicamente vantajosa. Em um mundo de diversos *frameworks* que prezam pela fácil curva de aprendizado em detrimento do desempenho e otimização das soluções, e bibliotecas de *software* populares com buracos de segurança, este trabalho trata diversos aspectos de grande importância para o estudo de melhores práticas de desenvolvimento e adequações nas escolhas de tecnologias para a construção de sistemas integrados.

No contexto do curso de engenharia de controle e automação, os temas abordados neste trabalho são intimamente relacionados às disciplinas obrigatórias e optativas oferecidas pelo Departamento de Automação e Sistemas: “Introdução à Informática para Automação (DAS5334)”, “Fundamentos da Estrutura da Informação (DAS5102)”, “Metodologia para Desenvolvimento de Sistemas (DAS5312)”, “Redes de Computadores para Automação (DAS5314)”, “Programação Concorrente e Sistemas de Tempo Real (DAS5131)”, “Sistemas Distribuídos para Automação (DAS5315)”, e “Integração de Sistemas Corporativos (DAS5316)”. A disciplina “Fundamentos de Sistemas de Banco de Dados (INE5225)”, optativa oferecida pelo Departamento de Informática e Estatística, também é relevante.

2 FUNDAMENTAÇÃO TÉCNICA E TEÓRICA

Neste capítulo são detalhadas toda as teorias e tecnologias relevantes à compreensão do funcionamento do sistema piloto e do novo sistema proposto neste trabalho, descritos de maneira introdutória no Capítulo 1. Primeiramente, são apresentados os conceitos de programação concorrente, computação paralela e programação assíncrona, conceitos que são comumente usados de maneira equivalente porém são diferentes, embora sejam relacionados. É apresentada a linguagem de programação Rust, e em seguida são apresentados os principais protocolos que formam a *Web* e padronizam a comunicação via *Internet*, assim como algumas bibliotecas de *software* utilizadas no desenvolvimento deste trabalho. É detalhado o projeto de *design* unificado da IBM conhecido como *Carbon Design System*, no contexto do desenvolvimento de interfaces de aplicações *Web*. Também são apresentados os conceitos de virtualização de *hardware* e virtualização de sistemas operacionais, precursores do que se conhece hoje como computação em nuvem, paradigma que também recebe sua própria seção. São detalhadas algumas tecnologias de virtualização em nível de sistema operacional relevantes, como a plataforma *Docker*, assim como ferramentas de gerenciamento de aplicações conteinerizadas, como o *Kubernetes* e o *Red Hat OpenShift*. É apresentada de maneira geral a plataforma de computação em nuvem da IBM, conhecida como *IBM Cloud*, além de alguns serviços de *software* ofertados por ela, sobretudo a *IBM Cloud Functions* e o *Watson Knowledge Studio*, relacionados aos sistemas desenvolvidos neste trabalho. Por fim, são apresentados os projetos *Apache Kafka* e Scylla. O Scylla é um banco de dados distribuído *NoSQL*[1] voltado para aplicações de tempo-real, e o *Apache Kafka* é uma plataforma distribuída para a troca de mensagens e processamento de fluxos de dados, também em tempo-real. Ambos projetos são ferramentas complexas de *software* que são aplicadas no novo sistema proposto pelo autor neste trabalho.

2.1 PROGRAMAÇÃO CONCORRENTE

Na maioria dos sistemas modernos, os sistemas operacionais gerenciam múltiplos processos simultaneamente. No código-fonte de um *software* que é executado por um processo, também é possível a definição de tarefas em partes separadas e independentes que são executadas de maneira entrelaçada (aparentemente simultânea) pelo sistema – as entidades que executam tais partes são as chamadas *threads*. A divisão de um programa em múltiplas *threads* pode melhorar em muito o desempenho do sistema, dada a capacidade de execução de várias tarefas “simultaneamente”. En-

[1] O acrônimo refere-se a expressão de língua inglesa *Not only SQL*, para designar genericamente bancos de dados não-relacionais, ou que suportem consultas diferentes do padrão usado pela Linguagem de Consulta Estruturada, ou *Structured Query Language (SQL)*.

tretanto, esse conceito conhecido como programação concorrente, trás consigo uma pesada carga de complexidade, e uma problemática diversa. Dentre os problemas mais comuns, temos:

a) As chamadas “condições de corrida”, ou *race conditions*, onde duas ou mais *threads* acessam dados ou recursos computacionais em uma ordenação inconsistente, prejudicando de maneira potencialmente crítica o funcionamento do sistema; ou

b) os “impasses”, ou *deadlocks*, onde duas *threads* permanecem esperando uma pela outra a terminarem de usar recursos, impossibilitando a continuação da execução dos processos.

Esses são apenas dois exemplos clássicos de falhas que podem ocorrer em sistemas concorrentes. Em grandes sistemas, diversos *bugs* que ocorrem apenas em situações específicas e difíceis de serem reproduzidas são comuns, e muitos tornam-se falhas críticas de segurança em sistemas largamente utilizados. Por exemplo, o próprio *kernel*[2] do Sistema *Linux* possui um número enorme de falhas de segurança, decorrentes de erros de acesso de memória e outras falhas clássicas de sistemas concorrentes – uma análise dessas vulnerabilidades, que são de conhecimento público, é disponível no CVE[3].

Em um trabalho apresentado no *Linux Security Summit* em 2017 por Dmitry Vyukov[4], centenas de *bugs* foram encontrados no *Kernel Linux*, relacionado somente ao gerenciamento de portas USB[5]. Um agente com acesso físico a uma máquina alvo pode facilmente se aproveitar das vulnerabilidades identificadas por meio do uso de um simples *pen-drive* USB com *software* malicioso, podendo até mesmo obter acesso irrestrito à máquina alvo – uma falha de segurança gravíssima. Dentre as vulnerabilidades encontradas (com a ajuda de um *software* automatizado), cerca de quase metade é relacionada a falhas comuns na programação de sistemas concorrentes. A distribuição de tipos de falhas encontradas no trabalho de Vyukov é apresentada na Tabela 1, com alguns exemplos de vulnerabilidades registradas no CWE[6].

Nota-se que quase metade das falhas de segurança são decorrentes de erros de programação comuns em sistemas concorrentes, cometidos até mesmo pelos pro-

[2] Em termos de computação, o *kernel* é o componente principal do sistema operacional, e tem como objetivo primário gerenciar os recursos de hardware do sistema.

[3] *Common Vulnerabilities and Exposures (CVE)* é um sistema de referências e método de catalogação para a exposição de vulnerabilidades de software publicamente conhecidas, mantido pelo Departamento de Segurança Nacional dos Estados Unidos da América. As vulnerabilidades do *kernel Linux* são disponíveis em `https://tinyurl.com/rb9vrmf` (acessado em: 17/01/2020).

[4] A gravação da apresentação do trabalho é disponível em `https://tinyurl.com/rcuxhf9` (acessado em: 17/01/2020).

[5] *Universal Serial Bus (USB)* é uma porta de hardware para entrada e saída de dados disponível em virtualmente todos os computadores modernos.

[6] *Common Weakness Enumeration (CWE)* é um sistema de registro de tipos de falhas de segurança de software (também chamadas de *exploits*), e serve como uma linha de base para os esforços de identificação, mitigação e prevenção de vulnerabilidades.

Tabela 1 – Distribuição de causas de falhas de segurança no módulo de comunicação USB do Kernel Linux (2017).

Causa (tipo de falha)	**Registro de um exemplo de falha no CWE**	**Total**
“WARNING”	-	21.1%
“General-protection-fault”	`https://cwe.mitre.org/data/definitions/424.html`	20.0%
“Use-after-free”	`https://cwe.mitre.org/data/definitions/416.html`	18.5%
“deadlock/hang/stall”	`https://cwe.mitre.org/data/definitions/833.html`	12.5%
“BUG/panic/div0”	`https://cwe.mitre.org/data/definitions/369.html`	10.3%
“Heap-out-of-bounds”	`https://cwe.mitre.org/data/definitions/125.html`	5.2%
“Wild-access”	`https://cwe.mitre.org/data/definitions/843.html`	4.8%
“Un-init-memory”	`https://cwe.mitre.org/data/definitions/824.html`	4.0%
“Stack-out-of-bounds”	`https://cwe.mitre.org/data/definitions/787.html`	2.4%
“Double-free”	`https://cwe.mitre.org/data/definitions/415.html`	0.8%

Fonte: (VYUKOV. . . , s.d.).

gramadores mais experientes. Somente falhas do tipo *deadlock* são responsáveis por 12.5% das vulnerabilidades de seguranças encontradas no módulo USB do *kernel Linux*.

Para a programação de sistemas concorrentes, a maioria dos sistemas operacionais provê uma API para a criação de novas *threads*, e as linguagens de programação também podem implementar *threads* de maneiras diferentes. O modelo onde uma linguagem executa uma chamada diretamente para o sistema operacional para a criação de uma *thread* é chamado “1:1” (um para um), resultando em uma *thread* de sistema operacional para uma *thread* de linguagem de programação. Algumas linguagens possuem suas próprias implementações especiais de *threads*, que são conhecidas como *green threads*. As *green threads* são agendadas para execução por um *runtime*, ou máquina virtual, em vez de nativamente pelo sistema operacional. Isso possibilita a emulação de ambientes *multithreaded* sem a dependência de capabilidades nativas do sistema operacional subjacente. Por essa razão, esse modelo é chamado de “M:N” (M por N), onde existem M *green threads* por N *threads* de sistema operacional, e M e N não são necessariamente o mesmo número. A maioria das linguagens mais populares do mundo, como *Python* e *Java*, implementam *green threads*.

Além dos diferentes tipos de *threads*, existem basicamente dois grandes padrões para a criação de sistemas concorrentes: comunicação por memória compartilhada *(shared-memory communication)*, e comunicação por troca de mensagens *(message-passing communication)*. A comunicação por memória compartilhada é basicamente a possibilidade de múltiplas *threads* acessarem a mesma memória, entretanto com a necessidade de um gerenciamento de acesso para evitarem escritas simultâneas no mesmo recurso. Existem diversos mecanismos para implementar tal controle,

como exclusão mútua *(mutex)*, que é um sistema de travas que impede a escrita simultânea na memória compartilhada por *threads* diferentes. Como mecanismo de controle na comunicação por memória compartilhada também existem os semáforos, criados em 1965 por Edsger Dijkstra, como maneira alternativa para a implementação de exclusão mútua.

Os *mutexes* e semáforos são implementações de baixo nível que são notórias por serem complicadas de se implementar corretamente em grandes sistemas, e por causa disso muitos programadores preferem outras abordagens de sincronização de *threads*. Uma alternativa de mais alto nível, é a comunicação por troca de mensagens. Nesse tipo de sistema concorrente, as *threads* não se comunicam por meio do compartilhamento de memória, mas sim pela transferência de memória através de mensagens. A transferência de mensagens é uma abordagem muito utilizada em sistemas de larga escala, e priorizada em várias linguagens de programação voltadas para sistemas distribuídos como *Go*[7] e *Erlang*[8].

Um dos modelos mais conhecidos, baseado na comunicação via troca de mensagens, é o modelo de atores, apresentado pela primeira vez em 1973 na *International Joint Conferences on Artificial Intelligence (IJCAI)* por Carl Hewitt, Peter Bishop e Richard Steiger. No modelo de atores, um ator é definido como unidade primitiva universal da programação concorrente – um ator é capaz de enviar e receber mensagens, criar outros atores e modificar seu próprio estado privado. Como cada ator só pode afetar outro ator indiretamente (via mensagem), é desnecessária a sincronização por trava ou exclusão mútua. O modelo de atores provê então uma base teórica para a criação de sistemas altamente escalonáveis (HEWITT; BISHOP; STEIGER, 1973).

2.2 COMPUTAÇÃO PARALELA

Computação paralela é um termo frequentemente confundido com programação concorrente, embora ambas estejam intimamente relacionadas. Na computação paralela múltiplos processos, ou tarefas, são executados literalmente de maneira simultânea, com o emprego de múltiplos computadores ou processadores *multi-core*. Na programação concorrente, não necessariamente existe paralelismo nas operações, já que é possível que múltiplas tarefas executem “simultaneamente”, porém de maneira entrelaçada, em um processador *single-core*.

Tecnologias de processamento paralelo já são praticamente onipresentes na maioria dos processadores produzidos atualmente, desde equipamento para computação especializada até computadores pessoais. A principal razão para a adoção

[7] *Go* é uma linguagem de programação compilada e focada em produtividade e programação concorrente, lançada em 2009 pela Google.

[8] *Erlang* é uma linguagem de programação desenvolvida pela Ericsson em 1986 para suportar aplicações distribuídas e tolerantes a falhas, executadas em ambiente de tempo-real e ininterrupto.

receptiva dessas tecnologias é o fato de que a computação paralela teoricamente pode ampliar substancialmente o poder de processamento de um computador, utilizando componentes padrões já existentes. Atualmente alguns processadores *multi-core* de computadores pessoais chegam a ultrapassar em poderio os supercomputadores de algumas décadas atrás. Entretanto, a utilização correta de arquiteturas *multi-core* requer o uso de algoritmos adequados, e o projeto de tais algoritmos para o aproveitamento de múltiplos núcleos de processamento não é uma tarefa simples. Embora esses temas sejam estudados há mais de 50 anos, e grande progresso tenha sido alcançado, muitos problemas ainda precisam ser resolvidos, e existe uma grande deficiência em métodos e ferramentas de suporte para o desenvolvimento de *software* adequado.

Outros fatores que impulsionaram a adoção de processadores *multi-core* pela indústria foram os problemas de consumo de energia e sobreaquecimento atrelados ao aumento da frequência de operação dos processadores. Para contornar os problemas térmicos os fabricantes, desde meados de 2010, optaram por produzir processadores *multi-core* mais energeticamente eficientes, em vez de escalonar a frequência de operação de processadores *single-core*. A partir dessa mudança de paradigma e a difusão da tecnologia para os computadores pessoais, as aplicações modernas passaram a necessitar que os programadores as paralelizassem para aproveitamento adequado das novas arquiteturas (HILL; MARTY, 2008).

Todavia, a redução do tempo de execução de algoritmos por meio de paralelização possui limites teoricamente estabelecidos. Em 1967, na *American Federation of Information Processing Societies (AFIPS)*, Gene Myron Amdahl, cientista que trabalhou no desenvolvimento de *mainframes* na IBM durante as décadas de 50 e 60, apresentou um modelo matemático que estabelece os limites teóricos em redução de latência para a execução de tarefas paralelizadas. Esse modelo ficou conhecido como Lei de Amdahl, e é apresentado na Equação (1) (HILL; MARTY, 2008).

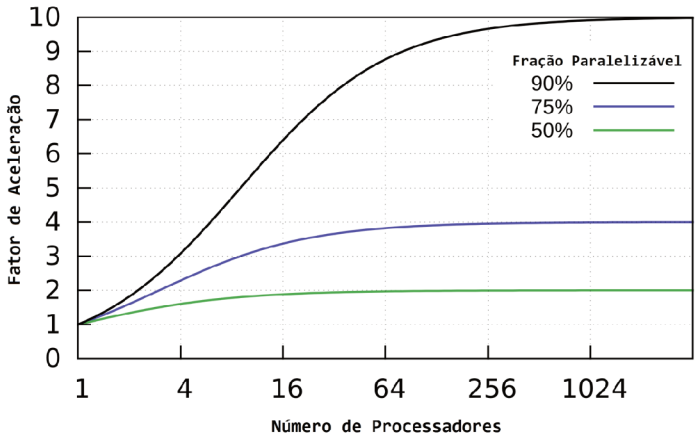
$$S_{lat}(s) = \frac{1}{1 - p + \frac{p}{s}} \tag{1}$$

Sendo S_{lat} o fator de aceleração potencial em latência de execução de uma tarefa inteira a ser paralelizada; s é o fator de aceleração em latência de execução das partes paralelizáveis de uma tarefa inteira; e p é a porcentagem, do tempo total de execução da tarefa inteira, relacionado à parte paralelizável da tarefa antes da paralelização.

Considerando um algoritmo com partes paralelizáveis e partes não-paralelizáveis, dado que $S_{lat} < 1/(1 - p)$, a Lei de Amdahl mostra que a existência de uma parte não-paralelizável em um algoritmo, mesmo que pequena, irá limitar o fator de aceleração possível de ser obtido por meio de paralelização. Por exemplo, se a parte não-paralelizável de um algoritmo representar 10% do tempo total de execução,

de acordo com a Lei de Amdahl é impossível atingir uma redução de mais de 10 vezes no tempo de execução, independentemente da quantidade de processadores utilizada na tarefa.

Figura 2 – Representação gráfica da Lei de Amdahl.



Fonte: Arquivo pessoal.

No entanto, a Lei de Amdahl é conservadora no sentido de que assume que o problema a ser computado é de tamanho fixo – o que na prática não é verdadeiro em muitos casos. Por exemplo, com o crescimento de um conjunto de dados de entrada, no geral o tempo de execução da parte paralelizável cresce mais rapidamente que o tempo de execução da parte não-paralelizável, tornando possível um fator de aceleração menos pessimista do que o definido pela Lei de Amdahl, conforme o problema a ser computado aumenta de tamanho.

A paralelização de algoritmos é relacionada diretamente com o paradigma da programação concorrente, e trás consigo um rol de problemas difíceis de serem tratados no âmbito da engenharia de *software*, conforme apresentado na Seção 2.1. Em relação à dificuldade de paralelização, as aplicações podem ser classificadas de acordo com a frequência necessária de sincronização entre cada sub-tarefa, e no geral as aplicações que possuem baixa ou nenhuma necessidade de sincronização são as mais fáceis de serem paralelizadas. Apesar da Lei de Amdahl formulada em 1967 ter se mostrado razoável por muitas décadas, a paralelização de tarefas é essencial em arquiteturas *multi-core* modernas para o aumento de desempenho (HILL; MARTY, 2008).

2.3 PROGRAMAÇÃO ASSÍNCRONA

No contexto deste trabalho, a programação assíncrona, ou mais basicamente o assincronismo, refere-se ao fluxo de execução de um *software*. O modelo padrão de

execução – síncrono – é baseado na execução de uma sequência de tarefas ordenadas, com uma relação causal entre a execução de cada sub-tarefa, isto é, o computador só executa a próxima tarefa após terminar a execução da tarefa anterior. Essa abordagem produz um resultado obviamente negativo: se uma tarefa for demasiadamente demorada, o sistema inteiro ficará bloqueado enquanto essa tarefa é executada. Na maioria dos casos é desejável executar outras tarefas, que não possuem relação causal com a tarefa bloqueante, e também não acessam e modificam os mesmos recursos. Por exemplo, considerando que uma tarefa de leitura de dados ou mesmo uma requisição enviada através da *Internet* é geralmente demorada, o processador fica apenas esperando a resposta de um sistema externo. Com o uso de técnicas de programação assíncrona, é possível definir um algoritmo que irá passar para outras tarefas enquanto a requisição não é resolvida, reduzindo o tempo de ociosidade do sistema. Geralmente algoritmos assíncronos são implementados com *multi-threading* e *loops* de eventos que checam periodicamente se os métodos assíncronos finalizaram.

2.4 A LINGUAGEM DE PROGRAMAÇÃO RUST

A linguagem de programação *Rust* nasceu na *Mozilla Research* – departamento da *Mozilla Foundation*, uma organização sem fins lucrativos conhecida pelo desenvolvimento e manutenção do *Mozilla Firefox*, um popular *Web browser* de código-aberto. A linguagem *Rust* foi concebida para ser focada no desenvolvimento de grandes sistemas de *software* paralelos, dando atenção minuciosa à segurança de memória. Semelhante às linguagens *C* e *C++*, *Rust* oferece controle granularizado acerca do *hardware* subjacente, com suporte direto à alocação de memória. A linguagem realiza um balanço a esse controle granularizado com alguns requisitos absolutos de segurança, como por exemplo, o compilador não permite a compilação de código onde erros comuns de memória podem ocorrer, como por exemplo *null pointers*, *dangling pointers*, *data races*, *buffer overflows*, acessos à memória não-inicializada ou não alocada. No geral, a linguagem *Rust* não permite a ocorrência de comportamento não-definido – exceto quando o programador explicitamente encapsula código em blocos com o macro "*unsafe*" (THE RUST REFERENCE, 2020).

A linguagem *Rust* preza por desempenho, e não possui sistema de coleta de lixo automática (que possui comportamento não-previsível), como as linguagens *Java* e *Go*. Alternativamente, o gerenciamento de memória é realizado com base na convenção *Resource Acquisition is Initialization* (RAII), definida por Bjarne Stroustrup – criador da primeira implementação da linguagem *C++*. Segundo a convenção RAII, o uso de memória por um objeto inicia-se na sua declaração, e termina quando o mesmo sai de escopo, onde a liberação de memória é executada por meio de funções destruidoras de objetos (STROUSTRUP, 1994). Dessa forma, a linguagem *Rust* provê gerenciamento determinístico dos recursos de *hardware*. No *Rust* todas as variáveis são imutáveis por

padrão, e existe também um sistema de *ownership*, onde cada variável possui um único proprietário, e o escopo da variável é o mesmo que o escopo de seu proprietário. Uma variável "*foo*" pode ser referenciada em outro escopo por meio da notação "*&foo*", ou transformada em uma variável mutável por meio da notação "*&mut foo*" – o compilador *Rust* garante por meio de checagem automática que a todo momento só pode existir uma única referência mutável, ou múltiplas referências imutáveis, para uma mesma variável.

A primeira versão estável do *Rust* foi lançada em maio de 2015. Os principais motivos pelo uso de tal linguagem neste trabalho são basicamente as garantias de segurança e bom desempenho devido às abstrações de custo zero, a concorrência segura, e a interoperabilidade com a linguagem *C*, permitindo o uso de bibliotecas otimizadas e já consolidadas com anos de uso na indústria. Além disso, o *Rust* possui um sistema de gerenciamento de bibliotecas chamado *cargo*, e uma comunidade de desenvolvedores crescente, que já construiu projetos completos de bibliotecas com código seguro e eficiente para a implementação de métodos assíncronos e algoritmos paralelos, com a aplicação de exclusão mútua, semáforos, e também o modelo de atores.

2.5 SERVIDORES WEB

Servidores *Web* são softwares dedicados a satisfazerem requisições de clientes na *World Wide Web (WWW)*, ou simplesmente *Web*. As requisições são processadas sob diversos protocolos, como HTTP, TCP e IP – e muitos outros, padronizados pelo modelo *Open Systems Interconnection (OSI)*[9]. No geral, a principal tarefa de um servidor *Web* é processar e entregar páginas *Web* para clientes, geralmente em formato de conteúdo estático como documentos HTML, CSS e *Javascript*. Neste trabalho de fim de curso, a programação de algumas aplicações *Web* foi realizada utilizando-se bibliotecas em duas linguagens de programação: *Python* e *Rust*, e nas seguintes Subseções são detalhadas as arquiteturas de comunicação entre cada tipo de aplicação e servidor *Web* construído neste trabalho.

2.5.1 Python Flask e a PEP3333

O *Python Flask* é uma biblioteca enxuta voltada para a criação de aplicações *Web*, em conformidade com a especificação PEP3333, ou *Python Web Server Gateway Interface v1.0.1 (WSGI)*. Aplicações *Flask* podem ser servidas através de qualquer servidor em conformidade com o padrão WSGI. O WSGI, ou PEP3333, basicamente define uma interface de comunicação padrão entre aplicações e servidores *Web*.

[9] O Modelo OSI é um modelo de referência da ISO, criado em 1971 e formalizado em 1983, com objetivo de ser um padrão para protocolos de comunicação entre sistemas computacionais.

O *Flask* é considerado um "*microframework*", pois existe apenas suporte básico e simples para o funcionamento de uma aplicação mínima, e o suporte para funcionalidades mais complexas como abstração de bancos de dados e validação de formulários é inexistente. Entretanto, o núcleo do *Flask é extensível*, e existem milhares de bibliotecas de código-aberto capazes de adicionarem funcionalidades extras. Um exemplo de aplicação *Python Flask* mínima é apresentado no Código 1.

Código 1 – Aplicação Web mínima escrita com Python Flask.

```
1 from flask import Flask, escape, request
2
3 app = Flask(__name__)
4
5 @app.route("/")
6 def hello():
7     name = request.args.get("name", "World")
8     return f"Hello, {escape(name)}!"
```

A aplicação apresentada no Código 1 utiliza anotadores personalizados (precedidos pelo caractere "@"), que alteram funções *Python* tradicionais – que são efetivamente utilizadas na programação da aplicação *Web* – para realizarem a comunicação com um servidor *Web* através do padrão WSGI. No exemplo, a rota "/" é definida para a função "*hello()*", que basicamente captura dois argumentos que são enviados junto à URL da requisição do tipo GET, e retorna uma mensagem de saudação para o cliente.

2.5.2 Nginx e uWSGI

O *Nginx* é um servidor *Web* levíssimo de código-aberto e desenvolvido em 2005 por Igor Vladimirovich Sysoev. Ele é escrito majoritariamente em linguagem *C*, com uso de alguns *scripts Perl*. O *Nginx* é um dos servidores *Web* mais utilizados do mundo, e muitas vezes é usado como *proxy* reverso em conjunto com servidores *Apache*.

O uWSGI (não confundir com o padrão WSGI), assim com o *Nginx*, é também um servidor *Web* completo escrito em linguagem *C*, capaz de servir aplicações em conformidade com a PEP3333. Existem vários servidores similares como *Gunicorn* e *Werkzeug*, entretanto para este projeto foi definido o uso do uWSGI, principalmente devido à sua grande flexibilidade e desempenho testados em campo, incluindo o fato de que o servidor *Nginx* pode ser usado como *proxy* reverso para o servidor uWSGI.

Neste trabalho, a aquitetura dos servidores *Web* escritos em linguagem *Python* foi definida com base no uso em conjunto do *Nginx* e uWSGI. O *Nginx* possui capacidades poderosas de *caching*, capazes de desonerarem muito o sistema na entrega de conteúdo estático, que não precisa ser enviado repetidas vezes para os mesmos clientes. Devido ao fato de que o *Nginx* não atende a especificação PEP3333, foi de-

cidido utilizá-lo como *proxy* reverso entre os clientes na *Internet* e o uWSGI, que de fato atende à PEP3333 e pode se comunicar com aplicações *Web* escritas em *Python Flask*. O uWSGI é capaz de se comunicar rapidamente com o *Nginx* por meio do protocolo de baixo nível *uwsgi* (em uma escolha infeliz de nomenclatura, o protocolo e o servidor *Web* possuem o mesmo nome – apenas com diferenças de caixa alta e baixa).

2.5.3 Rust Hyper

O *Hyper* é uma biblioteca da linguagem *Rust* que implementa o protocolo HTTP de maneira correta, em ambas as versões 1.1 e 2.0. A biblioteca é relativamente de "baixo-nível", voltada para ser usada como ponto de partida para a criação de outras bibliotecas e aplicações. O *Hyper* é construído em torno de um *design* assíncrono, sendo integrável facilmente com o pacote *Tokio*, outra biblioteca de relativo baixo-nível voltada para a criação de aplicações assíncronas em *Rust*.

2.5.4 Rust Actix-Web

O *Actix-Web* é uma biblioteca puramente escrita em *Rust*, de código-aberto, e completa para a criação de aplicações *Web*, já com servidor integrado. O nome *Actix* originou-se no fato de que o servidor implementado era baseado no modelo de atores, detalhado na Seção 2.1.

Em junho de 2019, com o lançamento da primeira versão estável do *Actix-Web* (1.0.0), a comunidade mantenedora confirmava a decisão de alterar completamente a arquitetura do servidor *Web* do modelo de atores para um modelo de serviços, baseado em três abstrações de *softwares* em um modelo voltado para a programação de servidores eficientes e modulares, com alta concorrência: *composable futures*, *services* e *filters* (ERIKSEN, 2013) – é a mesma arquitetura utilizada pelos servidores do *Twitter*[10]. Os *composable futures* são resultados de operações assíncronas, e são representadas por um valor "futuro" atrelado a uma operação específica; os *services* são basicamente funções assíncronas com uma API homogênea e simétrica (clientes e servidores são representados pela mesma abstração); e *filters* – que também são funções – encapsulam aos *services* características agnósticas às aplicações, como *timeouts* e *retries* (ERIKSEN, 2013).

Aplicações desenvolvidas com *Actix-Web* expõem um servidor *Web* HTTP integrado em um executável nativo, que pode ser implantado por trás de outro servidor *Web* como o *Nginx* ou o WSGI. Também é possível expô-lo diretamente à *Internet*. O *Actix-Web* oferece suporte nativo aos protocolos HTTP versão 1.1 e 2.0, incluindo os

[10] *Twitter* é uma rede social e um servidor para *microblogging*, que permite aos usuários enviar e receber atualizações pessoais de outros contatos, por meio da página *Web* do serviço ou SMS.

protocolos de segurança criptográfica *Secure Socket Layer (SSL)* e seu sucessor, o *Transport Layer Security (TLS)*.

No Código 2 é apresentada uma aplicação *Web* mínima escrita em linguagem *Rust* com a bibilioteca *Actix-Web*.

Código 2 – Aplicação Web mínima escrita com Rust Actix-Web.

```
use actix_web::{web, App, HttpServer, Responder};

fn index(info: web::Path<(u32, String)>) -> impl Responder {
    format!("Hello {}! id:{}", info.1, info.0)
}

fn main() -> std::io::Result<()> {
    HttpServer::new(
        || App::new().service(
              web::resource("/{id}/{name}/index.html").to(index)))
        .bind("127.0.0.1:8080")?
        .run()
}
```

Em *benchmarkings* realizados pela *TechEmpower* em julho de 2019, logo após o lançamento da versão estável do servidor *Rust Actix-Web* 1.0.0, o mesmo se sobressaiu em primeiro colocado em cinco dos seis testes comparativos realizados. Os resultados para o *benchmarking fortunes* são apresentados na Tabela 2.

Tabela 2 – Resultados dos testes comparativos de desempenho de servidores *Web* feitos pela TechEmpower em julho de 2019.

Rank	**Framework**	**Linguagem**	**Pontuação**
1	actix-core	Rust	702,165 (100.0%)
2	actix-pg	Rust	632,672 (90.1%)
3	h20	C	465,058 (65.0%)
4	atreugo-prefork-quicktemplate	Go	435,874 (62.1%)
5	vertx-postgres	Java	403,232 (57.4%)
6	ulib-postgres	C++	359,874 (51.3%)
7	fasthttp-postgresql-prefork	Go	352,914 (50.3%)
8	greenlightning	Java	341,347 (48.6%)
9	cpoll-cppsp-raw	C++	326,149 (46.4%)
10	fasthttp-postgresql	Go	324,149 (46.2%)

Fonte: Página oficial da TechEmpower `https://tinyurl.com/w8voors` (acessado em: 13/01/2020).

O teste *fortunes* é o mais interessante dos testes realizados pela *TechEmpower*,

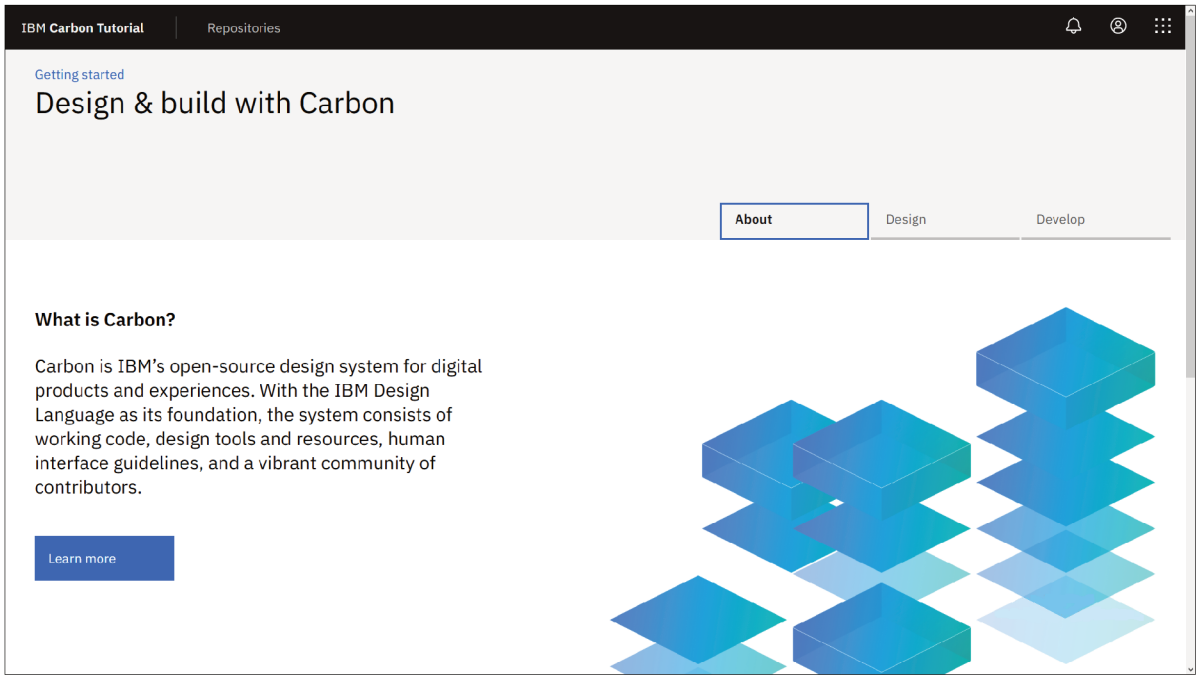
pois envolve consultas em um banco de dados e várias operações comuns em aplicações *Web* reais, em vez de tarefas simples e já otimizadas em inúmeras bibliotecas, como o simples envio de uma estrutura JSON via HTTP. É notável que uma biblioteca extremamente recente tenha conseguido alcançar resultados impressionantes sem o apoio financeiro de gigantes da tecnologia ou fundações como a *Apache*. Entretanto, embora o *Actix-Web* tenha apresentado resultados excelentes em testes comparativos, ainda resta o "teste em campo" da biblioteca, em grandes projetos reais.

2.5.5 IBM Carbon Design System

A IBM possui um sistema de *design Web* unificado, com o objetivo de padronizar o visual das páginas *Web* dos mais de 2000 produtos oferecidos pela corporação. Esse sistema recebe o nome de *IBM Carbon Design System*[11], e é um conjunto completo de diretrizes e padrões visuais, de experiência do usuário.

Na Figura 3 é apresentada uma amostra de página *Web* criada segundo as diretrizes do *IBM Carbon Design System*.

Figura 3 – Amostra de página *Web* criada com o sistema unificado de design da IBM.



Fonte: Arquivo pessoal.

Paralelamente às diretrizes visuais e de experiência do usuário, é fornecido um pacote de código-fonte completo para ajudar a acelerar o processo de desenvolvimento de interfaces *Web*. Os pacotes são blocos de código *Javascript*, HTML e CSS prontos

[11] Página Web disponível em `https://www.carbondesignsystem.com/` (acessível em: 18/01/2020).

para serem usados na construção de componentes como botões, tabelas, barras de rolagem, etc.

Neste trabalho, o autor preparou um arquivo CSS miniaturizado manualmente, com *design* praticamente idêntico ao do *Carbon Design System*. Essa decisão foi tomada principalmente devido ao grande tamanho do arquivo CSS padrão do sistema – não existe suporte para servidores *Web* escritos em *Rust*, e nem em *Python* – apenas em *Node.js*[12], que não será utilizado neste trabalho.

2.6 VIRTUALIZAÇÃO DE HARDWARE

A virtualização de *hardware* é basicamente a capacidade de simular – por meio de *software*, *hardware* especializado, ou ambos – componentes ou máquinas inteiras. Na perspectiva do *software* que é executado em uma máquina virtualizada, ele está sendo executado diretamente em uma máquina física genuína. A tecnologia surgiu nos anos 60, a partir do sistema experimental IBM M44/44X, precursor dos sistemas CP-40 da IBM, que implementavam múltiplos sistemas operacionais em um mesmo *mainframe* e colocavam à prova o conceito de memória virtual, que era ainda apenas uma área de pesquisa na época (BITNER; GREENLEE, 2012). A partir do CP-40, nasceram os primeiros hipervisores, controladores de *software* responsáveis por distribuírem os recursos de *hardware* e abstraírem essa camada subjacente para os diferentes sistemas operacionais hospedados na mesma máquina.

A principal vantagem da virtualização de *hardware* está na economia de custos com capital e trabalho, sobretudo devido ao fato de que por meio da virtualização, múltiplos servidores podem ser consolidados em um único grande computador físico, reduzindo a quantidade necessária de componentes onerosos como CPUs[13] e discos rígidos. Além dos custos relacionados à fabricação ou aquisição de vários computadores, custos com energia também são reduzidos (BUYYA; BROBERG; GOSCINSKI, 2010). Ademais, máquinas virtuais também podem ser facilmente copiadas e instaladas em outras máquinas físicas, e são mais facilmente gerenciadas remotamente.

A virtualização de *hardware* provoca uma mudança de perspectiva: os recursos computacionais não são mais tratados como recursos físicos separados, mas sim como recursos “lógicos” (SMITH; NAIR, 2005). Usando virtualização, é possível consolidar processadores, armazenamento, e bandas de rede em um ambiente virtual no qual os recursos de *hardware* são utilizados de forma otimizada e com mais flexibilidade. Tecnicamente, a virtualização é a criação de substitutos virtuais para os recursos reais – substitutos que possuem as mesmas funções e interfaces externas que os recursos

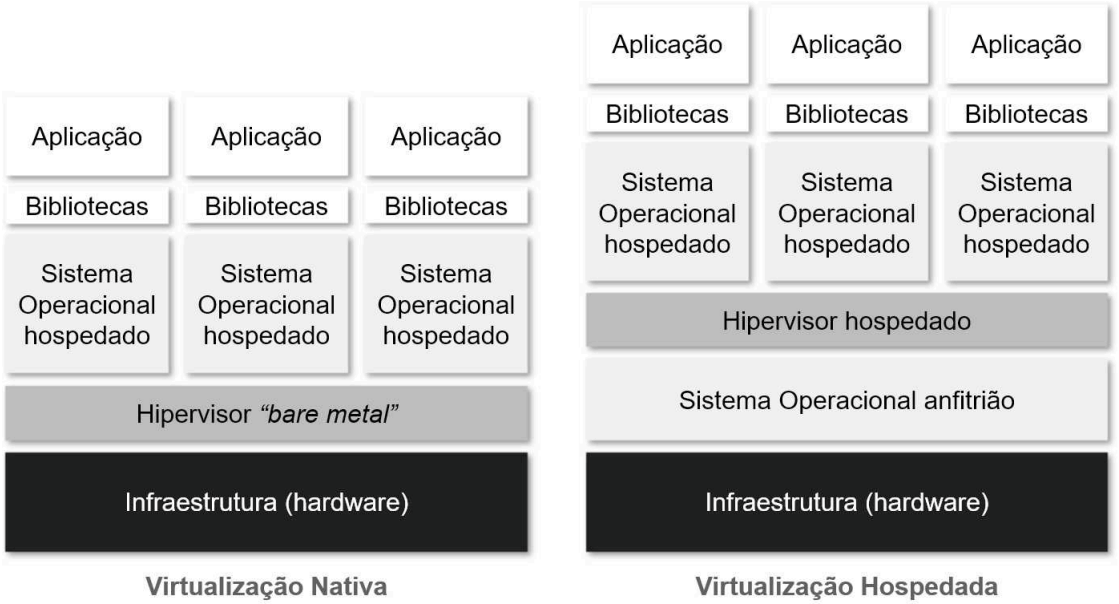
[12] *Node.js* é um interpretador de *JavaScript* assíncrono com código-aberto orientado a eventos, criado por Ryan Dahl em 2009 e focado em programação de aplicações *Web*.

[13] “Unidade Central de Processamento”, cujo acrônimo provém da língua inglesa: *Central Processing Unit*, é também comumente chamado simplesmente de processador.

reais, mas diferem em tamanho, desempenho e custo.

Genericamente, existem duas arquiteturas principais de virtualização de hardware: a virtualização nativa – também conhecida popularmente como *bare-metal* – e a virtualização hospedada (GOLDBERG, 1973). Essas duas arquiteturas são esquematizadas na Figura 4.

Figura 4 – Arquiteturas de virtualização de hardware.



Fonte: Arquivo pessoal.

A virtualização de *hardware* é uma das grandes tecnologias que possibilitaram a difusão do paradigma da computação em nuvem, tornando possível o provisionamento de infraestrutura computacional de maneira automatizada (SMITH; NAIR, 2005).

2.7 VIRTUALIZAÇÃO DE SISTEMAS OPERACIONAIS

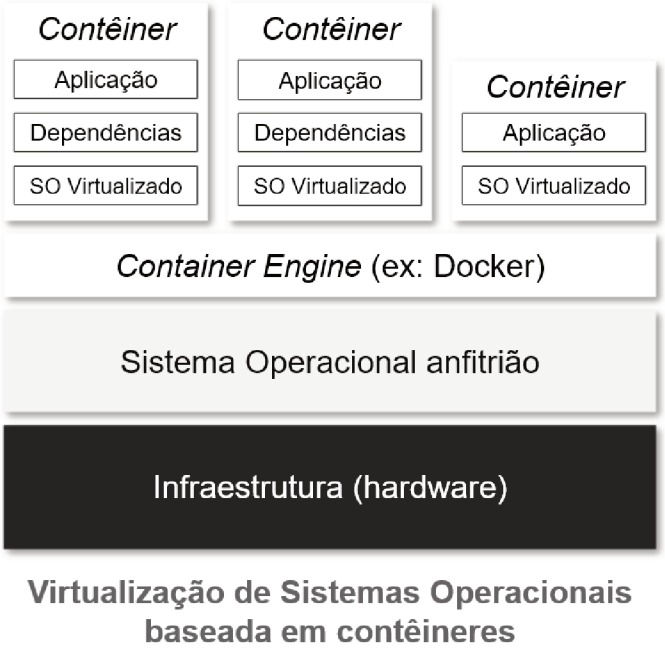
De modo análogo à virtualização de *hardware*, a virtualização em nível de sistemas operacionais possibilita que múltiplos sistemas operacionais – ou "ambientes de usuário" – sejam executados em um mesmo *kernel*. Esses ambientes de usuário são popularmente chamados de contêineres, e do ponto de vista do *software* encapsulado, uma aplicação executada em um contêiner aparenta estar sendo executada diretamente em um sistema operacional real, isolado. Aplicações executadas em contêineres não conseguem acessar recursos de *hardware* subjacentes, apenas os recursos designados ao seu contêiner específico.

A virtualização em nível de sistema operacional é comumente utilizada em aplicações onde ela é útil para alocar de maneira segura infraestrutura computacional entre uma grande quantidade de usuários mutuamente não-confiáveis. Outros típicos

cenários de uso incluem a separação de vários programas em contêineres diferentes, melhorando a segurança, proporcionando independência de *hardware*, e adicionando recursos de gerenciamento extras (BUYYA; BROBERG; GOSCINSKI, 2010).

Na Figura 5, temos um esquemático das camadas da arquitetura de virtualização de sistemas operacionais baseada em contêineres.

Figura 5 – Arquitetura de virtualização de sistemas operacionais via containers.



Fonte: Arquivo pessoal.

A virtualização em nível de sistemas operacionais apresenta uma grande vantagem sobre a virtualização completa de *hardware* apresentada anteriormente, pois os programas executados dentro das partições isoladas, ou contêineres, utilizam a interface de chamadas de sistema diretamente, e toda a camada de um ambiente de simulação completo intermediário é desnecessária, permitindo um desempenho superior às aplicações executadas em máquinas virtuais tradicionais.

É possível notar na Figura 5 que o hipervisor e os sistemas operacionais existentes na arquitetura de virtualização de *hardware* hospedada são substituídos por uma *container engine*, ou *daemon*. Essa eliminação de camadas intermediárias é o principal fator na melhora de desempenho dos contêineres em relação às máquinas virtuais tradicionais (BUYYA; BROBERG; GOSCINSKI, 2010).

2.7.1 A plataforma de contêineres Docker

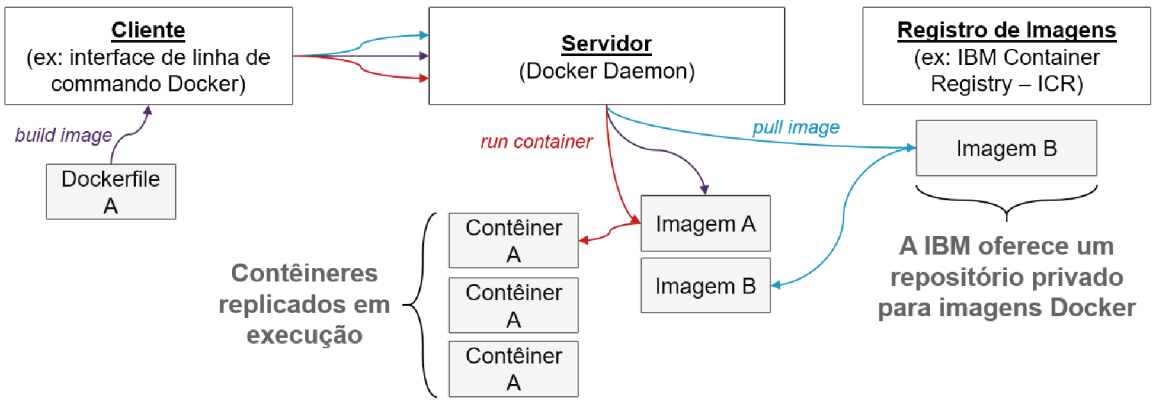
O projeto Docker foi fundado em 2010 e encubado como uma *startup* em 2011. A plataforma foi criada por Solomon Hykes e Sebastien Pahl em 2011, e o Docker

foi lançado oficialmente em março de 2013. O Docker utiliza o *Boot Filesystem (bootfs)* – um componente do *kernel Linux* – para encapsular processos em ambientes virtuais configuráveis de execução. Em termos simples, o Docker é uma plataforma para a virtualização de sistemas operacionais. Inicialmente baseada no *Linux Containers (LXC)*, a plataforma foi lançada em 2013 e rapidamente ganhou popularidade. A Red Hat em 2013 firmou parceria com a recém-nascida organização para o desenvolvimento de suporte nativo aos contêineres Docker no *Fedora Linux* e no *Red Hat Enterprise Linux (RHEL)*, além do desenvolvimento colaborativo da plataforma *Red Hat OpenShift (RHOS)* – detalhada na Seção 2.8.2 – para orquestração de aplicações conteinerizadas.

A arquitetura de funcionamento do Docker é baseada no paradigma cliente-servidor, onde um cliente utiliza uma CLI[14] (comando *docker*) para se comunicar com o *Docker Daemon (dockerd)*, responsável por gerenciar os diversos tipos de artefatos do Docker, como imagens, contêineres e serviços. De maneira simples, uma imagem contém toda a informação necessária para execução de uma aplicação (sistema operacional, binários estáticos, ambientes de execução e códigos-fontes) e são usadas como molde para a instanciação de contêineres.

Na Figura 6 temos um esquemático básico da arquitetura de funcionamento do Docker e as interações básicas entre seus principais componentes.

Figura 6 – Arquitetura da plataforma Docker.



Fonte: Arquivo pessoal.

O cliente Docker interage com o servidor através de uma interface de linha de comando (CLI), capaz de enviar comandos ao *Docker Daemon* por meio de uma API REST. Uma imagem Docker pode ser construída a partir de um arquivo sem extensão nomeado como *Dockerfile*, que contém instruções para a construção da imagem. O

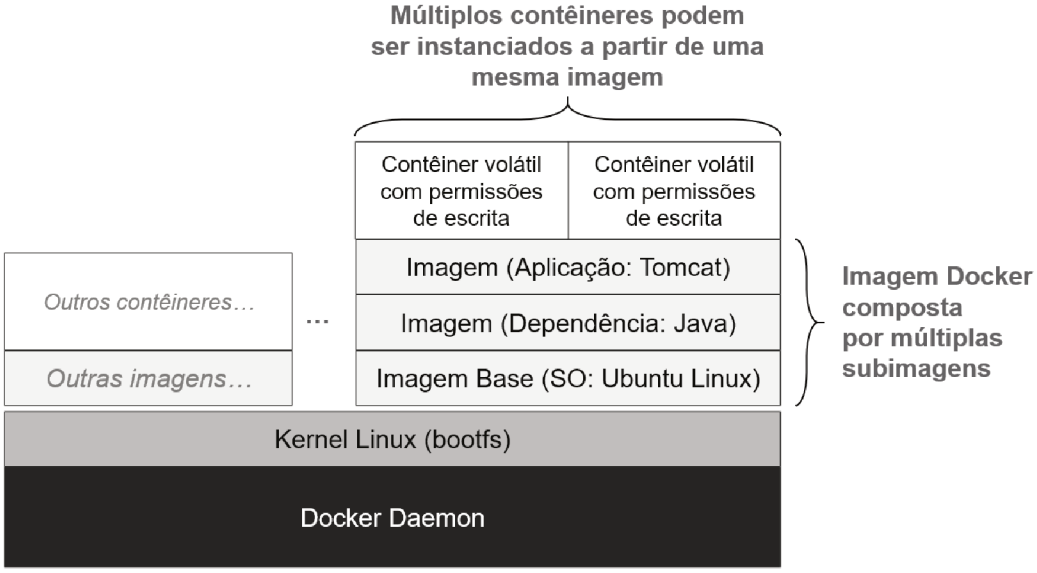
[14] "Interface de Linha de Comando", cujo acrônimo CLI provém da língua inglesa: *Command-Line Interface*.

contêiner, por sua vez, é uma entidade volátil que contém um *software* a ser executado e é gerenciado pelo *Docker Daemon*.

As imagens Docker, quando são construídas, são armazenadas em um repositório chamado geralmente de *Image Registry* ou *Container Registry*. Atualmente o Docker conta com um repositório comunitário conhecido como *Docker Hub*[15], que contém imagens base e oficiais de diversos sistemas operacionais e aplicações populares, além de contribuições da própria comunidade. Um registro privado geralmente é configurado para armazenar imagens que não devem ser expostas ao público.

As imagens Docker podem ser compostas por múltiplas imagens (e geralmente este é o caso). Na Figura 7 é esquematizada de maneira abstraída a estrutura de uma imagem Docker composta, com um contêiner construído sobre a mesma e executado pelo *Docker Daemon*.

Figura 7 – Estrutura exemplo de uma imagem Docker.



Fonte: Arquivo pessoal.

Conforme citado anteriormente, o contêiner é um artefato volátil e, durante o seu funcionamento, o gerenciamento de arquivos não permite a mudança nas informações relacionadas à imagem. Em outras palavras, as imagens, uma vez construídas com o comando *docker build*, são completamente estáticas. Essa estrutura torna possível o uso de uma única imagem para a instanciação de múltiplas cópias do mesmo contêiner, que poderão ser parados, reiniciados ou destruídos, e inclusive possuírem estados diferentes após a inicialização.

Atualmente existe um rol de ferramentas criadas especificamente para a administração de grandes aplicações conteinerizadas.

[15] Disponível em `https://hub.docker.com/` (acessado em: 08/01/2020).

2.8 GERENCIAMENTO DE APLICAÇÕES CONTEINERIZADAS

A possibilidade de escalonar múltiplos servidores Docker executando em máquinas diferentes gera inúmeras aplicações vantajosas. Por exemplo, um *cluster* capaz de ser gerenciado por um único cliente, pode implementar um mecanismo de replicação elástica das aplicações por meio de *scripts*, permitindo, por exemplo, que servidores *Web* multipliquem-se para atender uma certa carga temporária, ou que estejam sempre disponíveis caso parte das máquinas do *cluster* falharem. Diversas ferramentas robustas que vieram a ser denominadas "orquestradores" de contêineres foram criadas para gerenciar aplicações conteinerizadas. A mais famosa, usada em larga escala na indústria, é o Kubernetes.

2.8.1 Kubernetes

O Kubernetes nasceu como um projeto de código-aberto mantido pela Google em 2014, e atualmente está sob tutela da Linux Foundation[16]. A ferramenta é um sistema de orquestração de contêineres, utilizada para a automação de todos os processos de implantação de aplicações, desde a instanciação de contêineres, dimensionamento dinâmico, e a gestão das aplicações. A palavra "kubernetes" é proveniente da língua grega, e significa timoneiro.

Aplicações orquestradas pelo Kubernetes comumente são orientadas ao paradigma de microserviços, de maneira que cada serviço é implementado por uma imagem Docker replicada em diversos contêineres, conforme o que for determinado pelo operador do sistema – nota-se que o Kubernetes suporta várias tecnologias de virtualização de sistemas operacionais, e não apenas o Docker. O Kubernetes fornece meios para o gerenciamento do ciclo de vida completo destas aplicações conteinerizadas, permitindo que o usuário execute atualizações de maneira incremental sem derrubar o sistema, e também a replicação automática de novos contêineres em caso de falhas. Em termos simples, as interfaces do Kubernetes permitem que o usuário administrador defina o estado desejado da sua aplicação, e a ferramenta irá automaticamente construir ou destruir novos contêineres conforme o necessário para manter este estado pré-definido. Isso permite que o sistema seja previsível, escalável, e altamente disponível (LUKŠA, 2017).

A arquitetura interna de funcionamento do Kubernetes pode ser dividida em duas partes: o plano de controle, e o plano de nós "trabalhadores" (*worker nodes*) do *cluster* Kubernetes. Os nós são as máquinas hospedeiras onde contêineres são executados, podendo ser máquinas reais ou virtualizadas. Já o plano de controle do Kubernetes, refere-se aos componentes internos, ou nós mestres (*master nodes*). Um

[16] Fundada em 2007 pela fusão do *Open Source Development Labs (OSDL)* e o *Free Standards Group (FSG)*, a *Linux Foundation* patrocina o trabalho do criador do *Linux*, Linus Torvalds, e é suportada por diversas empresas e desenvolvedores de código-aberto do mundo todo.

nó mestre é responsável por gerenciar o *cluster* Kubernetes, e ele engloba os seguintes componentes: o *kube-apiserver*, responsável por toda comunicação interna entre o nó mestre e os nós trabalhadores do *cluster*; o *kube-scheduler*, responsável por definir onde iniciar (em qual nó trabalhador) as novas aplicações de acordo com uma política definida; o *kube-controller-manager*, que irá periodicamente comparar o estado atual do *cluster* Kubernetes com um estado de referência, e forçar alterações para manter o sistema no estado determinado; e, por fim, temos um mecanismo para armazenamento de pares de valores (*key-value store*) para guardar todas as informações sobre o estado do *cluster* – no Kubernetes, é utilizado por padrão o Etcd[17], que é um banco de dados distribuído para retenção de pares de valores, e serve para armazenar as informações de maneira organizada hierarquicamente como diretórios tradicionais, realizando a sincronização de seus nós através do protocolo Raft[18].

Uma outra parte importante de um *cluster* Kubernetes, é a Interface de Rede de Contêineres (ou CNI, da língua inglesa: *Container Network Interface*). A CNI é uma definição de biblioteca de *software* criada pela Linux Foundation, e visa o estabelecimento de um padrão para a criação de *software* para gerenciamento de redes virtualizadas voltadas para a comunicação entre contêineres Linux. O Kubernetes oferece por padrão a sua própria implementação da interface, chamada *kubenet*, entretanto existem diversas alternativas com recursos e funcionalidades extras. O repositório oficial do projeto CNI[19] lista várias alternativas.

Em relação aos componentes do Kubernetes que são executados nos nós trabalhadores, temos três: o *kubelet*, que é um agente responsável por comunicar o status e gerenciar os contêineres em *pods* (sem tradução adequada na língua portuguesa, os *pods* representam uma unidade gerenciável no Kubernetes); o *kube-proxy*, que é um *proxy* de rede responsável por implementar o conceito de serviços do Kubernetes (basicamente forçar regras de comunicação via rede aos nós trabalhadores); e, por fim, um ambiente de execução de contêineres, como o Docker. Ambos os *kubelets* e *kube-proxies* de cada nó trabalhador comunicam-se com o *kube-apiserver* do nó mestre.

O Kubernetes é capaz de escalar automaticamente as aplicações implantadas no *cluster* por meio do balanceamento de novos contêineres em diferentes *pods*, baseando-se nas especificações e requisitos definidos por um usuário administrador, geralmente através de arquivos de extensão YAML com instruções declarativas – essa escalabilidade automática é geralmente referenciada por meio do termo *load-balancing*, da língua inglesa. O administrador do *cluster* é capaz de enviar comandos diretamente

[17] O Etcd é um sistema de armazenamento de pares de valores distribuído. Mais informações sobre a tecnologia podem ser encontradas em `https://etcd.io/` (acessado em: 09/01/2020).

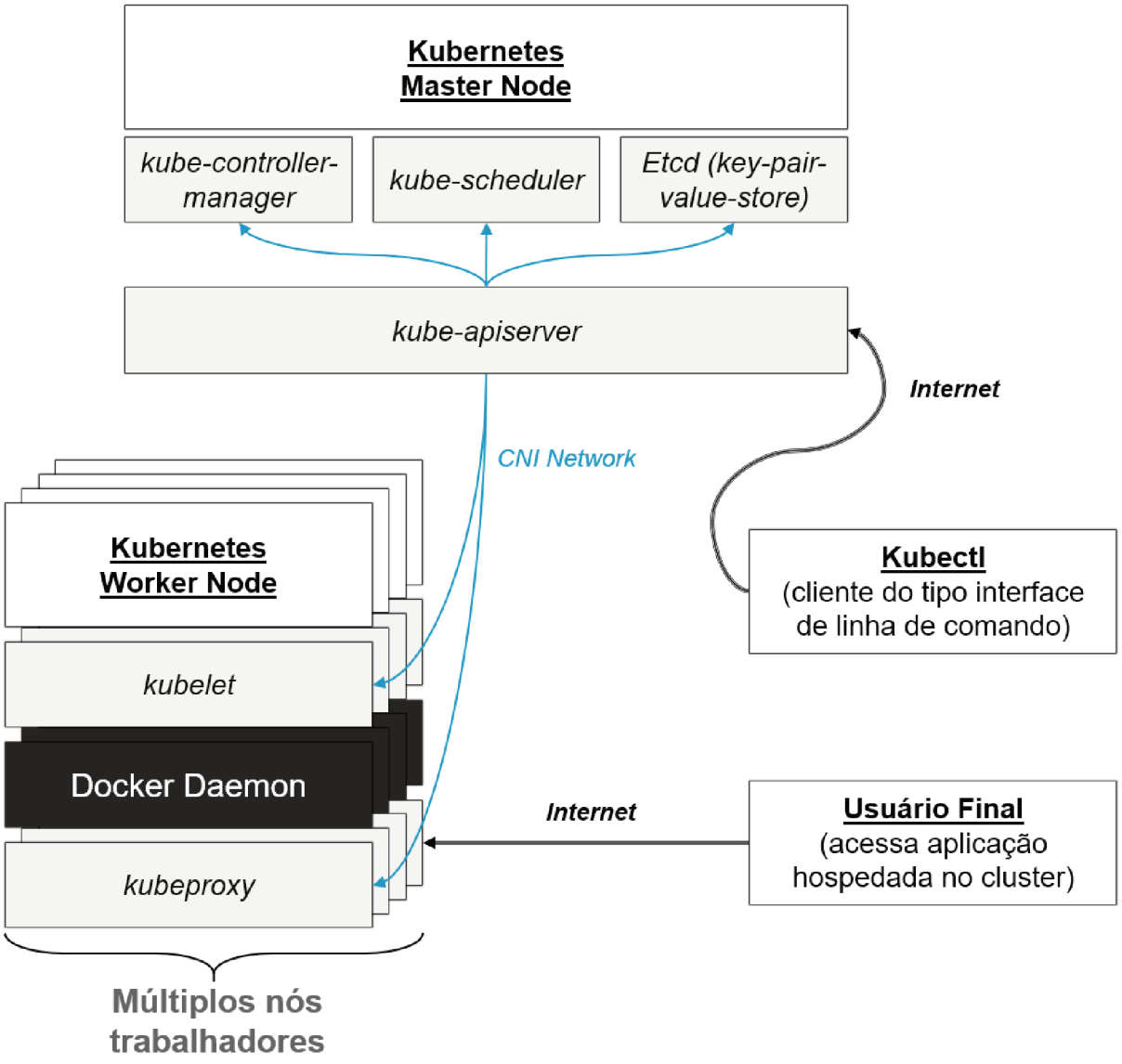
[18] O Raft é um protocolo de consenso para sistemas distribuídos equivalente ao protocolo Paxos, dividido em subtarefas mais simples. Mais informações sobre o protocolo podem ser encontradas em `https://raft.github.io/` (acessado em: 09/01/2020).

[19] Acessível em: `https://github.com/containernetworking/cni` (acessado em: 09/01/2020).

para o *kube-apiserver* do nó mestre por meio de uma interface de linha de comando do Kubernetes (*kubectl*).

Na Figura 8 temos um esquemático dos principais componentes da arquitetura do Kubernetes e suas interligações.

Figura 8 – Arquitetura de um cluster Kubernetes.



Fonte: Arquivo pessoal.

2.8.2 Red Hat OpenShift

O OpenShift (comumente referenciado pelo acrônimo RHOS) é uma plataforma para o desenvolvimento e gerenciamento de aplicações conteinerizadas, voltada para o mercado corporativo. O sistema foi desenvolvido pela Red Hat, e sua última versão (v4) é baseada em contêineres CRI-O (alternativa ao Docker criada pela Red Hat) or-

questrados com a ferramenta Kubernetes executada em um sistema operacional Red Hat Enterprise Linux (RHEL), focando em segurança corporativa. Historicamente o Kubernetes sempre teve problemas com atualizações do Docker que alteravam detalhes que eram assumidos pela arquitetura do Kubernetes. Para sanar esses problemas, a Red Hat desenvolveu do zero uma plataforma para virtualização em nível de sistema operacional, completamente em conformidade com o padrão da interface do Kubernetes para as *Container Engines*. Diferentemente do Docker, o CRI-O foi concebido puramente para funcionar em conjunto com o Kubernetes. O CRI-O suporta vários formatos de imagem, incluindo o formato de imagem Docker existente.

O OpenShift foi desenvolvido com base em uma arquitetura de microserviços, onde todos os seus componentes internos são unidades desacopladas que trabalham em conjunto. Essa arquitetura permite que cada componente da plataforma OpenShift seja completamente personalizável pelo usuário administrador, sendo uma característica muito desejável por empresas com grandes projetos de *software*. Além da possibilidade de customização, a Red Hat adicionou diversas ferramentas de DevOps[20] ao OpenShift, facilitando práticas para integração entre as equipes de desenvolvimento, administração e controle de qualidade de *software*, além da adoção de processos automatizados para produção rápida e segura de aplicações e serviços – principalmente por essas razões, a plataforma OpenShift tornou-se o padrão corporativo de implementação de aplicações em contêineres.

2.9 COMPUTAÇÃO EM NUVEM

O paradigma da computação em nuvem é, em termos simples, a extensão do modelo de utilidades públicas para a computação. Quando conectamos um eletrodoméstico em uma tomada, normalmente não nos preocupamos sobre como a energia elétrica é gerada ou como ela chega até a nossa casa – simplesmente utilizamos o serviço, que é prontamente disponível através de um soquete na parede. A computação em nuvem é a aplicação de maneira análoga desse conceito para a indústria de tecnologia da informação.

Com o advento de tecnologias de virtualização de *hardware*, computadores podem ser construídos a partir de componentes distribuídos e acessados via *Internet*: processamento, armazenamento, dados, e recursos de *software*. O público passou a ter acesso a infraestruturas computacionais que anteriormente possuíam custos proibitivos de instalação e manutenção. Também de maneira análoga ao exemplo da energia elétrica, os serviços de computação em nuvem são precificados com base em unidades de consumo, ou em esquemas de precificação mais diversos, dependendo

[20] O termo DevOps deriva da junção das palavras “desenvolvimento” e “operações”, sendo uma prática de engenharia de *software* que possui o intuito de unificar o desenvolvimento com todo o resto do ciclo de vida do *software*.

do tipo de serviço oferecido.

A literatura acadêmica define, no geral, o conceito de computação em nuvem como um paradigma da tecnologia da informação que permite o acesso onipresente a grupos de recursos compartilhados de sistemas configuráveis, que podem ser rapidamente provisionados com um esforço mínimo de gerenciamento através da *Internet*, semelhante às utilidades públicas (BUYYA; BROBERG; GOSCINSKI, 2010). Segundo (MELL; GRANCE, 2011), o paradigma da computação em nuvem é definido como um modelo *pay-per-use* que permite o acesso conveniente e sob demanda, a um grupo de recursos de computação configuráveis (redes, servidores, armazenamento, aplicações e demais serviços) de maneira rápida e com esforços e interações mínimas por parte do provedor. Diversas organizações técnicas e acadêmicas tentaram criar uma definição completa do conceito, entretanto, por tratar-se de um paradigma relativamente atual e em evolução, definições discrepantes não são incomuns.

Serviços de computação em nuvem fornecidos por terceiros permitem então que organizações e desenvolvedores foquem seus recursos em aspectos mais importantes dos negócios, como a qualidade de seus produtos digitais, em vez de desperdiçarem capital humano e financeiro na construção e manutenção de infraestrutura. Consequentemente, o uso de serviços de computação em nuvem permite o desenvolvimento e lançamento mais rápido de aplicações, minimizando os custos de entrada e manutenção, aprimorando a administração das aplicações e serviços por meio de ferramentas que também são fornecidas via *Internet*.

Os provedores de infraestrutura de computação em nuvem também fornecem métricas para os serviços usados, que são essenciais para as malhas de realimentação utilizadas na computação autônoma, permitindo que os serviços sejam escalonáveis sob demanda e executem recuperação automática. Portanto, a principal vantagem da computação em nuvem é possibilitar que os usuários beneficiem-se de uma vasta quantidade de tecnologias sem a necessidade de conhecerem a fundo ou possuírem conhecimento específico sobre cada uma delas. O paradigma é no geral procurado por empresas e organizações devido ao desejo de reduzir custos, e acaba auxiliando os usuários a concentrarem seus esforços na essência de seus negócios e evitarem obstáculos tecnológicos.

O conceito de computação em nuvem é ligado diretamente à área de atuação do autor desta monografia, e alguns dos serviços utilizados no desenvolvimento do sistema de *software* proposto neste projeto são ofertados através da plataforma de computação em nuvem da IBM.

2.9.1 Modelos de serviços

Os provedores de computação em nuvem no geral, incluindo a IBM, oferecem seus serviços seguindo o modelo SOA[21], filosofia de design de *software* na qual serviços são provisionados por meio de protocolos de comunicação via *Internet*. Os serviços são desenvolvidos com base em três diferentes modelos principais, padronizados pelo *National Institute of Standards and Technology (NIST)*[22] como: "Infraestrutura como um Serviço" – *Infrastructure as a Service (IaaS)*, "Plataforma como um Serviço" – *Platform as a Service (PaaS)*; e "*Software* como um Serviço" – *Software as a Service (SaaS)*. Também existem algumas novas especificações feitas por outros acadêmicos como "Base de Dados como um Serviço", ou *Database as a Service (DBaaS)*, mas essas são menos utilizadas. Esses modelos oferecem um alto nível de abstração e são comumente retratados como camadas em uma pilha, destacando quais camadas são responsabilidade do cliente e quais são do provedor de computação em nuvem. Na Figura 9 temos a representação das camadas de tecnologia de uma aplicação computacional para cada modelo de serviço, destacando o que é responsabilidade do provedor de computação em nuvem, e o que permanece responsabilidade do consumidor.

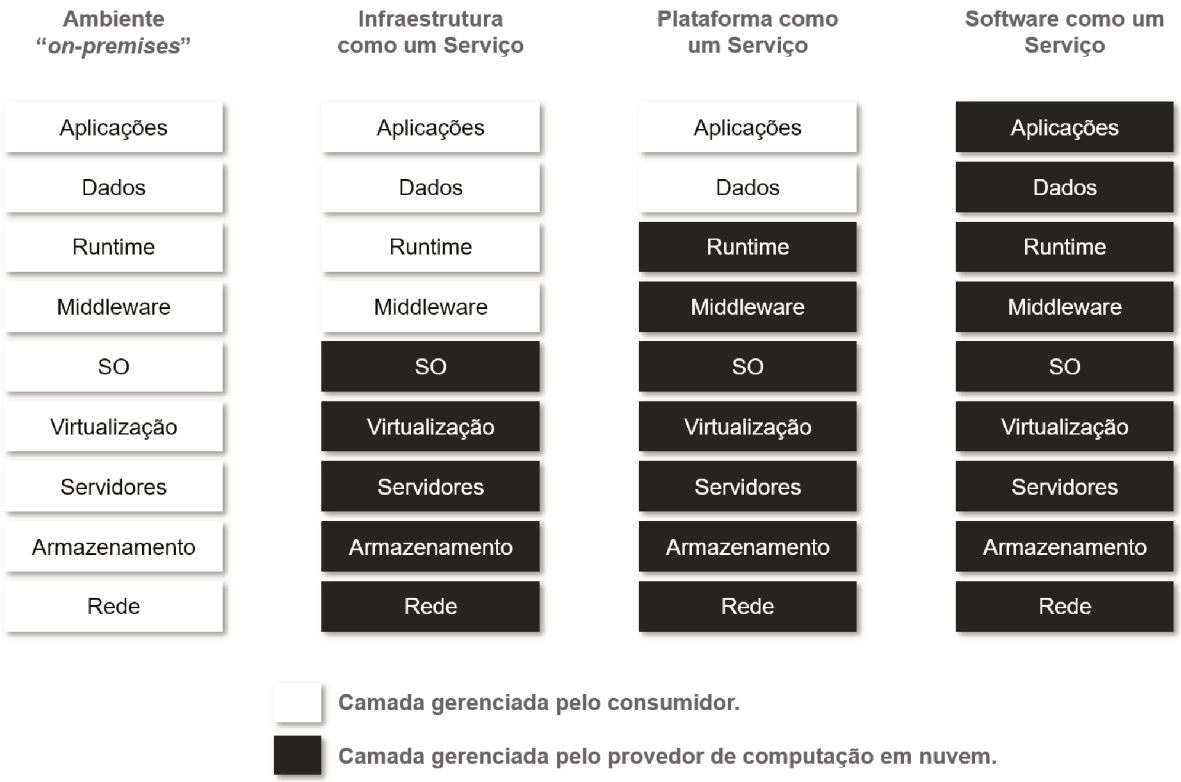
Infraestrutura como um Serviço refere-se a serviços *online* que fornecem APIs de alto nível usadas para referenciar vários detalhes de baixo nível de infraestrutura da rede subjacente, como recursos de computação física, localização, particionamento de dados, dimensionamento, segurança, *backup* etc. No modelo IaaS o consumidor pode implantar e executar *software* arbitrário, que pode incluir sistemas operacionais e aplicativos. O consumidor não gerencia ou controla a infraestrutura da nuvem subjacente, mas possui controle sobre sistemas operacionais, armazenamento, e aplicações implantadas e, possivelmente, controle limitado de componentes de rede selecionados (por exemplo, *firewalls* de *host*). Em suma, ofertas do tipo IaaS proveem recursos computacionais e o público alvo destes serviços geralmente é composto por engenheiros e administradores de sistemas.

No modelo de Plataforma como um Serviço, a capacidade fornecida ao consumidor é de implantar aplicações na infraestrutura da nuvem criadas usando linguagens de programação, bibliotecas, serviços e ferramentas suportadas pelo provedor. O consumidor não gerencia ou controla a infraestrutura da nuvem subjacente, incluindo rede, servidores, sistemas operacionais ou armazenamento, mas tem controle sobre as aplicações implantadas e, possivelmente, as configurações para o ambiente de hospedagem de aplicativos. Os fornecedores de nuvem então fornecem uma plataforma de computação, geralmente incluindo sistema operacional, ambiente de execução de

[21] "Arquitetura Orientada a Serviços", cujo acrônimo SOA provém da língua inglesa: *Service Oriented Architecture*.

[22] Anteriormente conhecido como *The National Bureau of Standards*, o NIST é uma agência governamental não-regulatória da administração de tecnologia do Departamento de Comércio dos Estados Unidos.

Figura 9 – Modelos de serviço comuns na computação em nuvem.



Fonte: Arquivo pessoal.

linguagem de programação, banco de dados e servidor *Web*. Os desenvolvedores de aplicativos podem desenvolver e executar suas soluções em uma plataforma em nuvem sem o custo e a complexidade de comprar e gerenciar as camadas subjacentes de *hardware* e *software*. Em suma, ofertas do tipo PaaS proveem plataformas de desenvolvimento e o público alvo desses serviços geralmente é composto por desenvolvedores de *software*.

Por último, no modelo de *Software* como um Serviço, a capacidade fornecida ao consumidor é a de usar aplicativos em execução na infraestrutura de nuvem do provedor. As aplicações são acessíveis a partir de vários dispositivos clientes através de uma interface como um navegador (por exemplo, *e-mail* baseado na *Web*). O consumidor não gerencia ou controla a infraestrutura de nuvem subjacente, incluindo rede, servidores, sistemas operacionais, armazenamento ou mesmo recursos de aplicativos individuais, com a possível exceção de configurações específicas do usuário. O modelo de contratação e pagamento para aplicações SaaS tipicamente é uma inscrição mensal ou uma taxa anual fixa por usuário. Em suma, ofertas seguindo o modelo SaaS proveem aplicações completas e o público alvo desses serviços geralmente é composto por usuários finais e empresas.

2.9.2 IBM Cloud

A *IBM Cloud*, plataforma de computação em nuvem da IBM, emergiu a partir da união dos serviços de mainframes com as tecnologias de virtualização de *hardware*. A IBM foi pioneira na tecnologia de virtualização, com o desenvolvimento das primeiras máquinas virtuais nos anos 60. O primeiro hipervisor desenvolvido foi o CP-67, usado para teste e desenvolvimento de *software*, e que permitia o compartilhamento de recursos de memória entre várias outras máquinas virtuais. Com o particionamento da capacidade de processamento de um mainframe em várias máquinas virtuais separadas, múltiplas aplicações podiam ser processadas paralelamente, tornando o *hardware* extremamente eficiente em comparação com um mainframe tradicional. Em 2007 a IBM introduziu a tecnologia de *Live Migration*, que refere-se ao processo de mover uma máquina virtual em execução para outra máquina física, sem desconectar uma aplicação cliente usando os serviços da máquina virtual. Posteriormente, a IBM procurou implementar uma padronização e automatizar as suas tecnologias, visando chegar a par com a proliferação de dados produzidos por *hardware* e centros de dados cada vez mais eficientes. Essa combinação de virtualização, padronização e automatização levou ao desenvolvimento da *IBM Cloud* como é conhecida atualmente.

Na Figura 10 é apresentada uma fotografia do interior de um dos centros de dados da IBM, localizado na cidade de Dallas no estado do Texas.

Figura 10 – Fotografia do interior de um dos centros de dados da IBM em Dallas.

Fonte: Twitter oficial da IBM (`https://twitter.com/ibm/status/981757445642047488`).

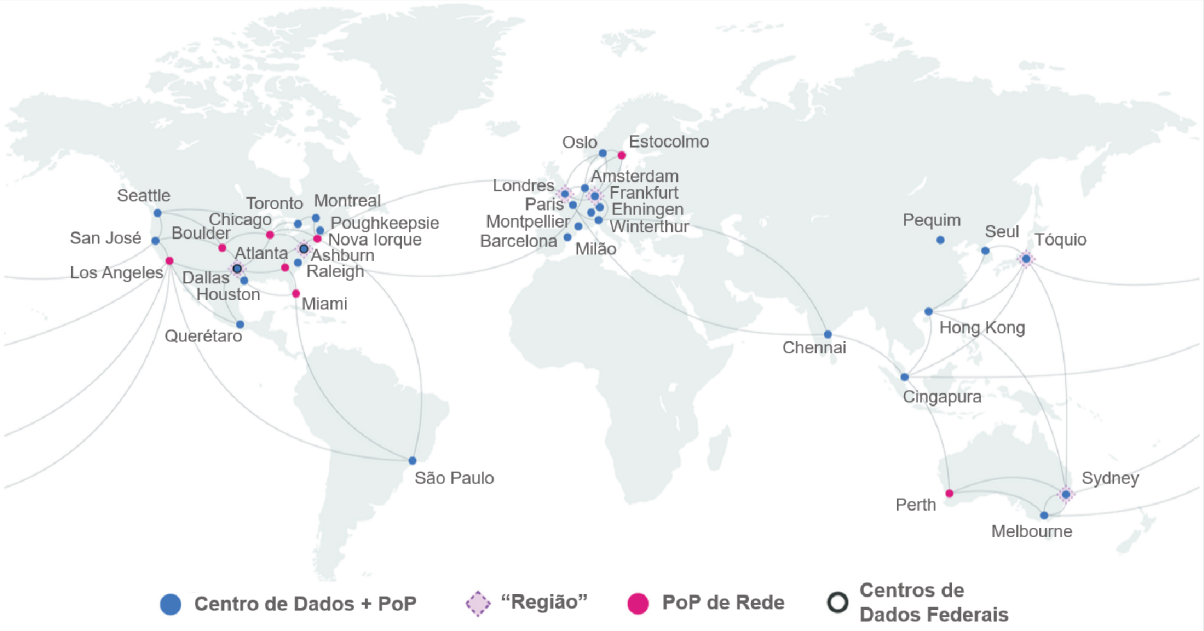
Hoje a *IBM Cloud* possui dezenas de centros de dados pelo mundo, com vários

pontos de presença de rede[23] (PoPs) conectados globalmente pela *Internet*. Quando um servidor da *IBM Cloud* é acessado, a rede é projetada de maneira a rotear a requisição da forma mais rápida possível, com o auxílio dos PoPs – quando um usuário solicita dados de um servidor *IBM Cloud*, esses dados viajam para o PoP de rede mais próximo, onde são entregues a outro provedor para transportar os dados pela distância restante.

A rede global da *IBM Cloud* é dividida em "regiões", que por sua vez são subdivididas em países e depois por zonas (centros de dados). Uma região, no contexto da *IBM Cloud*, é um grupo de centros de dados fisicamente separados (dentro de uma ou mais zonas de disponibilidade), com infraestruturas elétricas e de rede independentes umas das outras. Dessa forma, as regiões são projetadas para remover pontos únicos de falha compartilhados com outras regiões, e garantir baixas latências entre as suas zonas englobadas.

Na Figura 11 são mostradas as cidades com presença de PoPs e centros de dados da *IBM Cloud* pelo mundo.

Figura 11 – Localização geográfica dos centros de dados e pontos de presença de rede da IBM Cloud.



Fonte: Arquivo pessoal.

A *IBM Cloud* adota os modelos de serviço tradicionais da computação em nuvem, descritos na Seção 2.9.1. Os serviços provisionados sob essas arquiteturas são oferecidos por meio de modelos públicos, privados ou híbridos de implantação. Basicamente, a capacidade fornecida aos clientes é a de acessar aplicações dentro

[23] Mais conhecidos pelo termo de língua inglesa: *Network Points of Presence*.

da infraestrutura da IBM remotamente. O consumidor não administra ou controla a infraestrutura subjacente incluindo as redes, os servidores, os sistemas operacionais ou o armazenamento de dados, mas possui controle sobre as aplicações implantadas e sobre as configurações do ambiente de hospedagem.

A partir do provisionamento de infraestrutura computacional e componentes de *software*, aplicações podem ser executadas na nuvem e acessadas por qualquer cliente. Além disso, a IBM também fornece soluções próprias que consistem em um grande número de colaborações, ferramentas de produção de *software*, análise de texto, ciência de dados, reconhecimento de imagem e inteligência artificial aplicada (ofertas PaaS e SaaS). Recentemente a IBM também adicionou ao rol de serviços o IBM Q, um conjunto de APIs e ferramentas (como a biblioteca Python *qiskit*) para o desenvolvimento de algoritmos e circuitos quânticos, que podem ser executados remotamente nos computadores quânticos comerciais da IBM.

A *IBM Cloud* provê acesso a armazenamento, rede, servidores, e diversos recursos computacionais em um esquema de precificação *pay-as-you-go* – semelhante às utilidades públicas. Qualquer pessoa ou organização pode contratar os recursos por meio da página *Web*[24] da *IBM Cloud*. São oferecidas máquinas *bare-metal*, máquinas virtuais, *clusters* de contêineres orquestrados com Kubernetes – *IBM Kubernetes Service (IKS)*, *clusters* Red Hat OpenShift, e até uma plataforma para a criação e execução de funções sem servidor (*serverless functions*), baseada no projeto de código-aberto Apache OpenWhisk[25]. No contexto de escopo deste trabalho, as ofertas de infraestrutura computacional da *IBM Cloud* são importantes, pois são utilizadas tanto no sistema piloto, quanto na nova solução proposta pelo autor.

2.9.3 Computação Serverless

A computação sem servidor é um paradigma baseado no uso de tecnologias de computação em nuvem para oferecer infraestrutura computacional verdadeiramente sob-demanda. Em outras palavras, o consumidor apenas paga pela infraestrutura efetivamente utilizada (quando código está sendo executado), e não quando as máquinas estão ociosas. Esse paradigma se torna possível com o uso da tecnologia de conteinerização, que provisiona de maneira automatizada uma aplicação capaz de executar código fornecido pelo consumidor – tudo automaticamente exposto via API HTTP. O consumidor é cobrado então pelo número de chamadas realizadas ao seu código hospedado e também pelo tempo de execução. Quando nenhuma requisição está sendo realizada, os contêineres são destruídos, e o consumidor não paga por infraestrutura ociosa. Posteriormente, quando uma nova requisição eventualmente for realizada, o

[24] Acessível em `https://cloud.ibm.com/` (acessado em: 11/01/2020).

[25] O projeto Apache OpenWhisk é uma plataforma de código-aberto para a execução de aplicações baseada no paradigma da computação sem servidor. Página oficial do projeto: `https://openwhisk.apache.org/`. (acessado em: 11/01/2020).

provedor de computação em nuvem instancia um novo contêiner para execução do código do consumidor. Esse tipo de oferta é comumente denominado como de modelo "Função como um Serviço", ou *Function as a Service (FaaS)*.

Essa arquitetura proporciona uma enorme economia de capital, pois o consumidor não precisa alocar recursos e mão-de-obra nas camadas mais baixas, como gerenciamento de sistema operacional, máquinas, etc. Somente o código-fonte é necessário, e todo o resto é gerenciado pelo provedor de computação em nuvem – nem mesmo a imagem com o ambiente de execução precisa ser fornecido, já que, no caso da *IBM Cloud*, várias imagens preparadas e otimizadas já são oferecidas para a execução de código em dezenas de linguagens de programação. O desenvolvimento de aplicações *multi-threaded* ou concorrentes também é completamente simplificado, já que múltiplas cópias do código são executadas simultaneamente, tudo gerenciado de maneira automática pelo provedor de computação em nuvem. A oferta de computação *serverless* da *IBM Cloud* ainda permite que o consumidor integre de maneira extremamente simplificada suas funções, sejam com outras funções sem servidor ou com aplicações externas. O usuário também pode programar gatilhos temporizados, ou baseados em eventos para a ativação automática de seu código.

Todavia, esse paradigma extremamente atual enfrenta alguns possíveis problemas tecnológicos. Primeiro, o processo de instanciação de um novo contêiner leva um tempo consideravelmente longo em relação a simples execução do código em um servidor já preparado (como máquina virtual, ou contêiner já em execução). Dependendo do tamanho da imagem fornecida pelo consumidor ou do algoritmo implementado, a "partida fria" de um novo contêiner pelo provedor de nuvem pode levar um tempo potencialmente longo, inviabilizando algumas aplicações. Somado a isso, todos os grandes provedores de computação em nuvem no mercado impõem limites de tempo de execução nas funções *serverless*. Devido às restrições da tecnologia, o uso de funções sem servidor é geralmente limitado para tarefas simples e sem requisitos rigorosos de desempenho.

2.9.4 IBM Watson

Alguns dos serviços mais famosos disponibilizados por meio da *IBM Cloud* são as APIs e *frameworks* que levam o nome de Watson, em homenagem ao fundador da IBM, Thomas J. Watson (1874-1956). Esses *frameworks* e APIs foram desenvolvidos a partir de diversos projetos de pesquisa da IBM na área de inteligência artificial, como processamento de linguagem natural, análise semântica de texto, modelagem preditiva, reconhecimento de imagem e detecção de objetos.

Historicamente, o Watson nasceu de um projeto de criação de um sistema de respostas para perguntas de domínio aberto, isto é, perguntas que instigam o respondente a fornecer respostas usando suas próprias palavras. Sua primeira grande

aparição ao público foi em Fevereiro de 2011, quando participou do programa de perguntas e respostas britânico *Jeopardy!*, vencendo Ken Jennins e Brad Rutter, os dois maiores campeões da competição até então.

Na Figura 12 é apresentada uma fotografia do evento, com os placares finais dos competidores.

Figura 12 – IBM Watson contra Ken Jennings e Brad Rutter no programa *Jeopardy!* transmitido em 2011.

Fonte: Arquivo pessoal.

O programa de perguntas e respostas (quiz) é baseado em história, literatura, cultura e ciências, todavia, de maneira inversa aos quizes tradicionais, são apresentadas as respostas das perguntas e os concorrentes devem formular a pergunta correspondente a cada uma delas. A competição envolve a tomada de decisão ousada em poucos segundos pelos participantes, e é tradicionalmente conhecida por utilizar artifícios de linguagem para confundir os competidores. Dois anos antes da competição, os sistemas automatizados mais avançados de perguntas e respostas do mundo eram capazes de formular respostas apenas para perguntas extremamente simples e claras como: “Qual é a capital da Rússia?”, e ainda levavam minutos para retornar as respostas. No *Jeopardy!*, os participantes devem responder em segundos, o que tornava a possibilidade de um computador vencer humanos na competição minúscula, se não inexistente. Entretanto, o sistema desenvolvido pela IBM foi capaz de vencer os maiores campeões humanos apenas dois anos após o início de sua concepção.

Em depoimento após a competição, Ken Jennings, um dos capeões históricos do programa, diz que o Watson parecia “raciocinar” de maneira semelhante a um humano. É transcrito abaixo o depoimento original em língua inglesa.

> "*The computer's techniques for unravelling Jeopardy! clues sounded just like mine. That machine zeroes in on keywords in a clue then combs its memory (in Watson's case, a 15-terabyte databank of human knowledge) for clusters of associations with those words. It rigorously checks the top hits against all the contextual information it can muster: the category name; the kind of answer being sought; the time, place, and gender hinted at in the clue; and so on. And when it feels 'sure' enough, it decides to buzz. This is all an instant, intuitive process for a human Jeopardy! player, but I felt convinced that under the hood my brain was doing more or less the same thing.*" – Ken Jennings, em depoimento à revista americana *Slate*[26].

Entretanto, na época, a grande quebra de paradigma para o sucesso de Watson sobre os campeões humanos no jogo de perguntas e respostas não foi na criação de algoritmos complexos de processamento de linguagem, e sim na possibilidade de execução de algoritmos já conhecidos em um período de tempo suficiente pequeno. Para que o Watson se tornasse competitivo contra humanos, a IBM preparou um *hardware* extremamente poderoso, baseado nos sistemas POWER7, capaz de processar de maneira paralela múltiplas tarefas complexas que eram executadas simultaneamente em *threads* individuais. O Watson em 2011 funcionava em um *cluster* de 90 servidores IBM POWER750 Express, com quatro soquetes de CPUs IBM POWER7 em cada servidor (cada CPU IBM POWER7 possui oito núcleos com quatro *threads* cada, resultando em 128 *threads* por servidor, e 11520 *threads* no total). Dessa maneira, o sistema empregava paralelismo em massa para acelerar a velocidade de resposta do Watson, que executava milhares de algoritmos simultaneamente para encontrar várias respostas e analisar a mais adequada em segundos. No total, o *cluster* onde o Watson era executado era capaz de operar em mais de 80 Teraflops (80 trilhões de operações por segundo). Além do poderio em processamento, toda a base de dados do Watson – para a competição, era a coletânea completa de artigos da Wikipédia[27] – foi armazenada diretamente em memória RAM, pois o acesso de dados armazenados em discos rígidos tornaria os tempos de resposta bem mais lentos.

Com a vitória de Watson no *Jeopardy!*, o que era um problema considerado impossível de ser solucionado por um computador, tornou-se possível. Aproximadamente uma década desde a aparição no programa, o Watson deixou de ser apenas um sistema de respostas para perguntas de domínio aberto, e hoje acumula diversos algoritmos para realização de múltiplas tarefas "cognitivas", como enxergar, escutar, ler, conversar, interpretar, aprender, e recomendar. Com o surgimento deste tipo de sistema informatizado, capaz de compreender dados não-estruturados e interpretar texto em linguagem natural em segundos, várias indústrias vêm sendo completamente transformadas. Na indústria de atendimento ao cliente, o Watson é utilizado para proporcionar uma interação personalizada e direta 24 horas por dia com os clientes. A

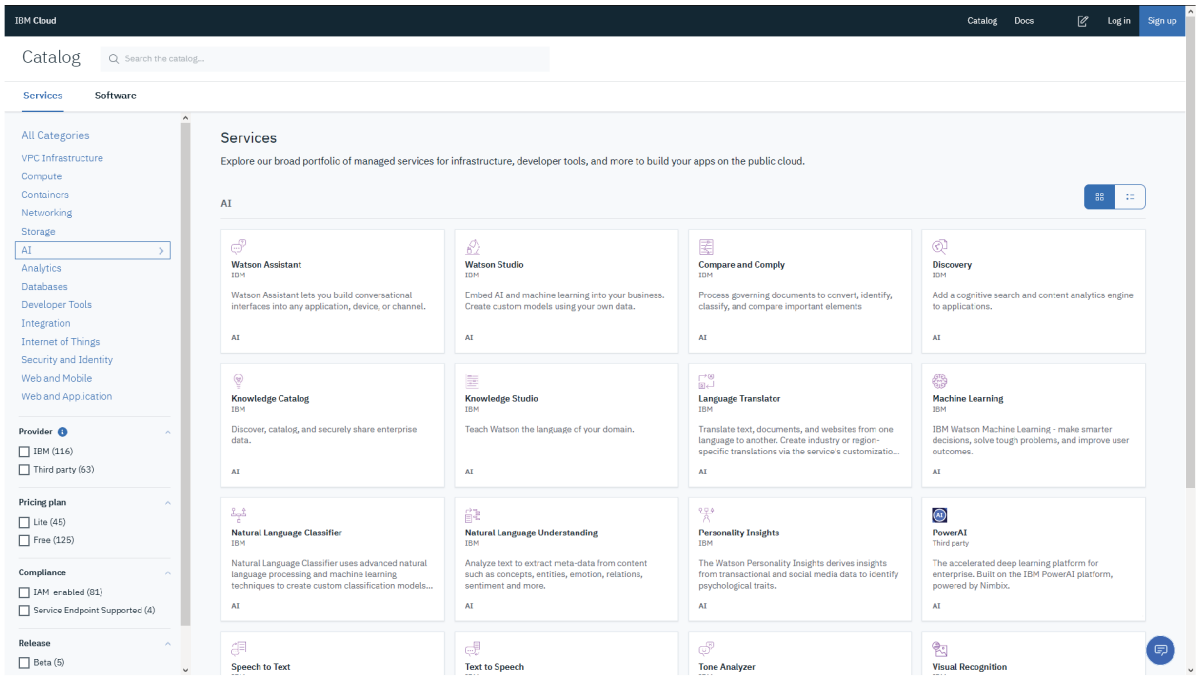
[26] Disponível em `https://tinyurl.com/was8psb` (acessado em: 11/01/2020).

[27] A Wikipédia é uma enciclopédia eletrônica mantida como projeto livre. Acessível em `https://www.wikipedia.org/` (acessado em: 11/01/2020).

Autodesk, uma das maiores clientes da IBM, é uma companhia global de 3D *Computer Aided Design (CAD)*, e trata cerca de 5 milhões de inquéritos de clientes por ano. O tempo de resolução de um inquérito, que anteriormente era de 38 horas, passou a ser de 5 minutos com a aplicação do Watson – segundo os relatos da própria Autodesk[28]. Essa diferença de tempo afeta diretamente a satisfação do cliente, e também reduz custos operacionais. As aplicações são vastas em muitas outras indústrias, como o setor bancário, o setor de seguros, e a indústria de telecomunicações.

O Watson atualmente é comercializado na forma de APIs de diversos serviços que podem ser instanciados por meio do página *Web* da *IBM Cloud*[29] – apresentada na Figura 13.

Figura 13 – Catálogo mostrando alguns serviços do Watson na página *Web* da IBM Cloud.



Fonte: Arquivo pessoal.

Esse modelo de negócio permite que virtualmente qualquer organização, independente de tamanho, seja capaz de implementar soluções avançadas de inteligência artificial.

De maneira mais específica, o Watson engloba os seguintes serviços principais (sendo algumas ofertas do tipo SaaS e outras do tipo PaaS):

a) O *Watson Assistant*, que é uma oferta PaaS completa para a criação de assistentes virtuais, onde o usuário define fluxos e intenções de conversa com

[28] Disponíveis em `https://www.ibm.com/watson/autodesk/` (acessado em: 10/01/2020).
[29] Catálogo de serviços do Watson disponível em `https://cloud.ibm.com/catalog?category=ai` (acessado em: 10/01/2020).

o auxilio de uma interface gráfica, e o assistente virtual (também chamado de *chatbot*) é exposto automaticamente via API REST, onde a requisição transporta frases de texto como entrada e o Watson retorna frases de texto com respostas pré-definidas pelo usuário.

b) O *Watson Tone Analyzer*, que é uma oferta SaaS de análise linguística para detectar entonações emocionais e de idiomas em texto escrito. O serviço é exposto via API, pronto para ser consumido. Texto é dado como entrada, e uma estrutura JSON é retornada para o cliente com os atributos de entonação, e os trechos onde eles foram identificados.

c) O *Watson Natural Language Understanding (WNLU)*, que é uma oferta SaaS para análise de texto e extração de metadados de conteúdos como conceitos, entidades, palavras-chave, categorias, sentimentos, emoções, relacionamentos e funções semânticas (diversos idiomas são suportados, incluindo a língua portuguesa). O serviço é exposto via API, pronto para ser consumido, entretanto existe a opção do usuário modificar o modelo padrão oferecido utilizando o *Watson Knowledge Studio*.

d) O *Watson Knowledge Studio (WKS)* é uma oferta PaaS completa para a criação de modelos extratores de metadados textuais personalizados, para serem expostos através do serviço *Watson Natural Language Understanding*. Esse serviço, por ser tratado em detalhes neste trabalho, é detalhado mais profundamente na Seção 2.9.5.

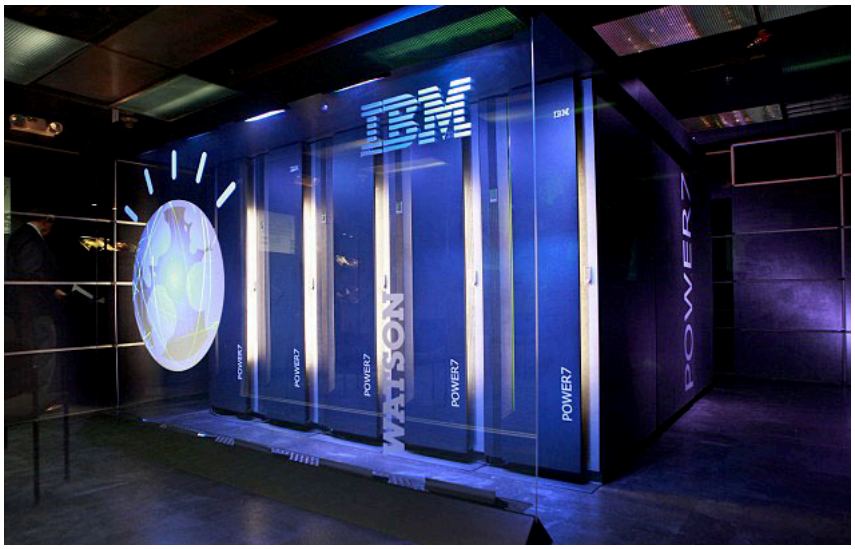
e) O *Watson Visual Recognition (WVR)*, que é uma oferta PaaS para a criação de modelos classificadores de imagem, e também voltados para a detecção de múltiplos objetos. É basicamente uma interface gráfica que auxilia o usuário no treinamento do modelo, e o expõe via API REST automaticamente para ser integrado com outras aplicações.

f) O *Watson Natural Language Classifier (WNLC)*, é uma oferta PaaS para a criação de modelos classificadores de texto. O usuário provê os conjuntos de treino e a ferramenta cria um modelo e o expõe automaticamente via API REST.

g) O *Watson Studio* é uma oferta PaaS completa para o gerenciamento e execução de projetos de ciência de dados completos, com ferramentas de suporte para todas as etapas de um projeto do tipo, desde a administração e tratamento de dados, até a implantação e exposição do modelo criado via API. Além disso, o Watson Studio permite que o usuário crie modelos preditivos diretamente com código, ou opcionalmente usando as interfaces gráficas da ferramenta.

Os serviços do Watson que são ofertas SaaS, quando instanciados por um consumidor, são materializados na forma de contêineres, que automaticamente expõem uma API com credenciais e chaves de acesso geradas para que o usuário utilize o *software*. A aplicação da tecnologia de virtualização permite que os serviços do Watson sejam completamente escalonáveis e à prova de desastres, devido às múltiplas zonas de disponibilidade espalhadas pelos diversos centros de dados da IBM localizados em regiões geográficas diferentes. Quanto às ofertas PaaS, elas são todas baseadas em aplicações *Web*, cujo acesso é destravado após a compra dos serviços (muitas das ofertas são livres de custo).

Figura 14 – Visão externa do cluster de servidores POWER750 onde era executado o Watson em 2011.



Fonte: Arquivo pessoal.

2.9.5 Estudo de caso: Watson Knowledge Studio

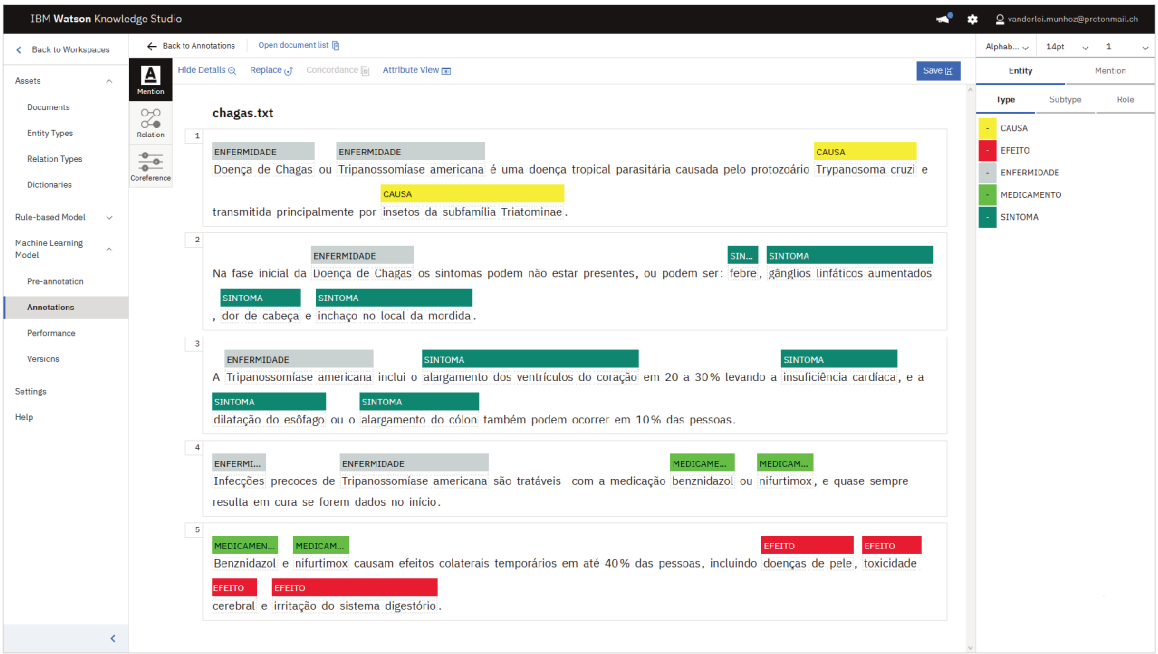
O Watson Knowledge Studio (WKS) é uma oferta PaaS da *IBM Cloud*, que fornece ao usuário a capacidade de extrair de maneira automatizada entidades e relações em informação textual não-estruturada. O serviço é disponibilizado para análise de texto direta em vários idiomas, como inglês, alemão, árabe, mandarim, italiano, japonês, coreano, espanhol, francês, holandês, e português. A plataforma fornece um ambiente de ponta-a-ponta para o treinamento de modelos estatísticos baseados em aprendizado de máquina. O usuário realiza a subida de documentos com texto para treino do modelo, e o WKS fornece ferramentas para auxiliar o processo de anotação de texto manual, realizado por um humano – que será descrito a seguir.

Para que o Watson seja capaz de reconhecer entidades e relações textuais, primeiramente um modelo customizado deve ser treinado por um humano (geralmente

um especialista de uma indústria com linguajar específico), e a ideia da plataforma é facilitar o processo de criação de modelo personalizado, baseado em inteligência artificial, onde o usuário treina o Watson por meio de exemplos. Em outras palavras, o modelo é criado com base em algoritmos de aprendizado de máquina supervisionados. De maneira simplificada, uma rede neural base e proprietária para cada idioma é mantida pelas equipes de pesquisa da IBM, e o modelo customizado pelo usuário com a ajuda do WKS é basicamente uma camada final (*layer*) adicionada à essa rede neural base proprietária. Dessa forma, o usuário ensina os termos de linguagem de seu domínio, área ou indústria, sem precisar de conhecimentos avançados de inteligência artificial e processamento de linguagem natural. Também não é necessário escrever código algum, pois o WKS fornece uma interface gráfica para a anotação dos documentos, elaboração de regras de anotação e uso de dicionários (que podem auxiliar no processo de treinamento).

Basicamente existem dois tipos de artefatos desejados: “entidades” e “relações”. Uma entidade é uma palavra ou múltiplas palavras que representam um item de relevância para o modelo de anotação. Na Figura 15 é apresentada a interface gráfica do WKS, com alguns exemplos de entidades anotadas em um texto sobre doenças parasitárias.

Figura 15 – Interface gráfica do WKS e anotação de entidades textuais.



Fonte: Arquivo pessoal.

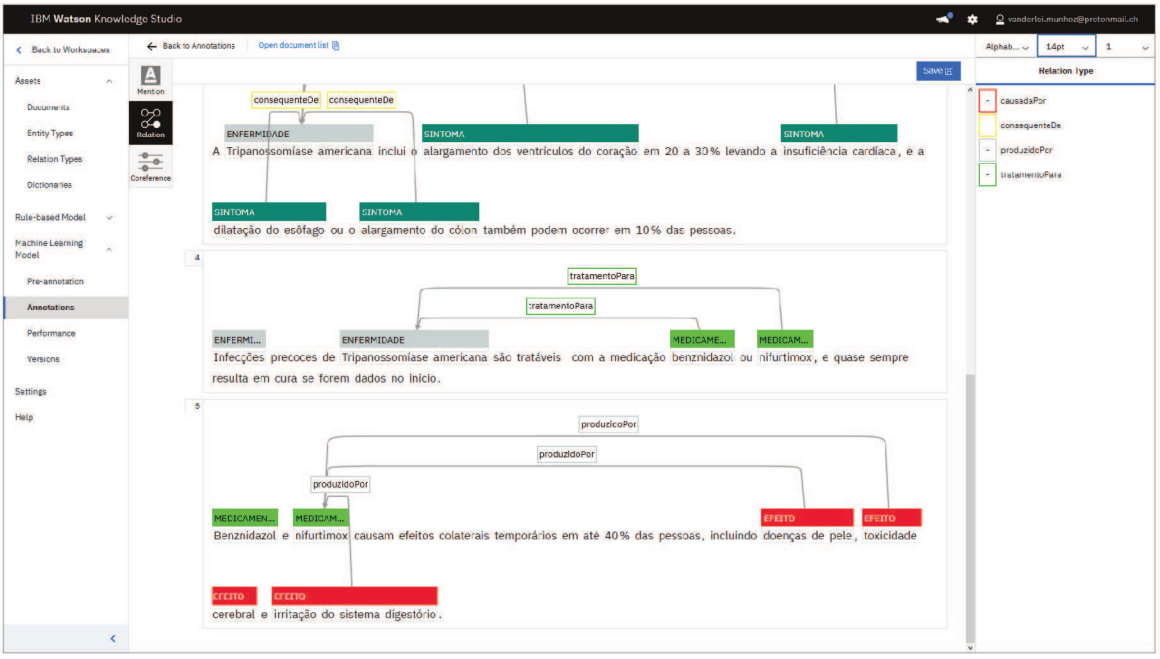
No exemplo mostrado na Figura 15, cinco entidades são relevantes: “CAUSA”, “EFEITO”, “ENFERMIDADE”, “MEDICAMENTO”, e “SINTOMA”. As palavras são destacadas manualmente por uma pessoa especialista no assunto tratado – no caso,

doença de Chagas. Após a marcação de todas as menções de entidades, o usuário deve indicar os tipos e classes de menções: tipos servem para diferenciar menções baseadas em partes da fala (nome, nominal, pronome, ou nenhum dos três); já as classes servem para especificar se uma menção é genérica (ex: "um livro"), negada (ex: "nenhum livro"), ou específica (ex: "o livro"). A anotação de tipos e classes de entidades serve para refinar o modelo anotador – elas não são obrigatórias para o treinamento. Após a anotação manual de todas as entidades relevantes, o usuário pode também especificar menções que são correferências (menções a uma mesma entidade específica, como por exemplo o presidente da França).

Após a anotação das entidades, se for desejado, o usuário pode definir também relações entre as entidades declaradas. Relações são definidas como um artefado que liga duas entidades de maneira ordenada (ex: ENTIDADE A está relacionada à ENTIDADE B por meio da RELAÇÃO AB), e o usuário da plataforma WKS é responsável por definir todas as entidades e relações que o modelo deve identificar – o modelo só é capaz de identificar o que for diretamente definido pelo usuário. No Watson Knowledge Studio, o conjunto completo com as definições dos tipos de entidades e tipos de relações é denominado *type system*.

Na Figura 16 são apresentados alguns exemplos de relações declaradas entre as entidades anotadas, continuando o exemplo de texto médico sobre doença de Chagas.

Figura 16 – Interface gráfica do WKS e anotação de relações textuais.



Fonte: Arquivo pessoal.

Recapitulando, no processo de anotação manual, o usuário utiliza a interface

gráfica do WKS para indicar todas as ocorrências de entidades e seus tipos. Após todas as entidades serem anotadas, o usuário deve indicar quais são correferências, isto é, múltiplas entidades que referem-se ao mesmo indivíduo, ou a um mesmo objeto específico. Após a anotação das correferências, o usuário deve indicar quais menções de entidades são genéricas ou específicas. Ao fim deste processo, o usuário passa para a anotação das relações entre as entidades. Após todas as relações estarem indicadas, o documento anotado pode ser utilizado para o treino do modelo de compreensão de linguagem natural. O WKS oferece diversas ferramentas para facilitar a anotação automática, e o usuário pode utilizar até mesmo o próprio modelo treinado previamente por ele para pré-anotar novos documentos.

Em um processo iterativo, de melhoras graduais, o usuário repete o processo de anotação com novos documentos até alcançar um sistema com uma taxa de acerto satisfatória para sua aplicação. Após a construção do modelo, o mesmo pode ser automaticamente exposto para ser consumido por qualquer outra aplicação via API REST HTTP.

2.10 KAFKA: PROCESSAMENTO DE FLUXOS DE DADOS

O projeto Apache Kafka é uma plataforma de *software* de código-aberto mantida pela *Apache Software Foundation* e criada em 2011, originalmente pelo LinkedIn[30]. O objetivo principal do projeto é a criação de um sistema voltado para o processamento de fluxos de dados em tempo real, otimizado para realizar operações de baixa latência. O Kafka é comumente utilizado como alternativa aos chamados *Enterprise Message Systems* (EMS). O Apache Kafka é capaz de realizar três tarefas essenciais: produzir e consumir fluxos de dados, como uma fila ou EMS; armazenar fluxos de dados de maneira tolerante a falhas e de maneira durável; e processar fluxos de dados conforme eles entram no sistema. Essas características tornam o Kafka uma solução robusta para sistemas que envolvem a troca de mensagens entre diversos componentes de *software*, e onde o processamento de dados em tempo real também é desejado.

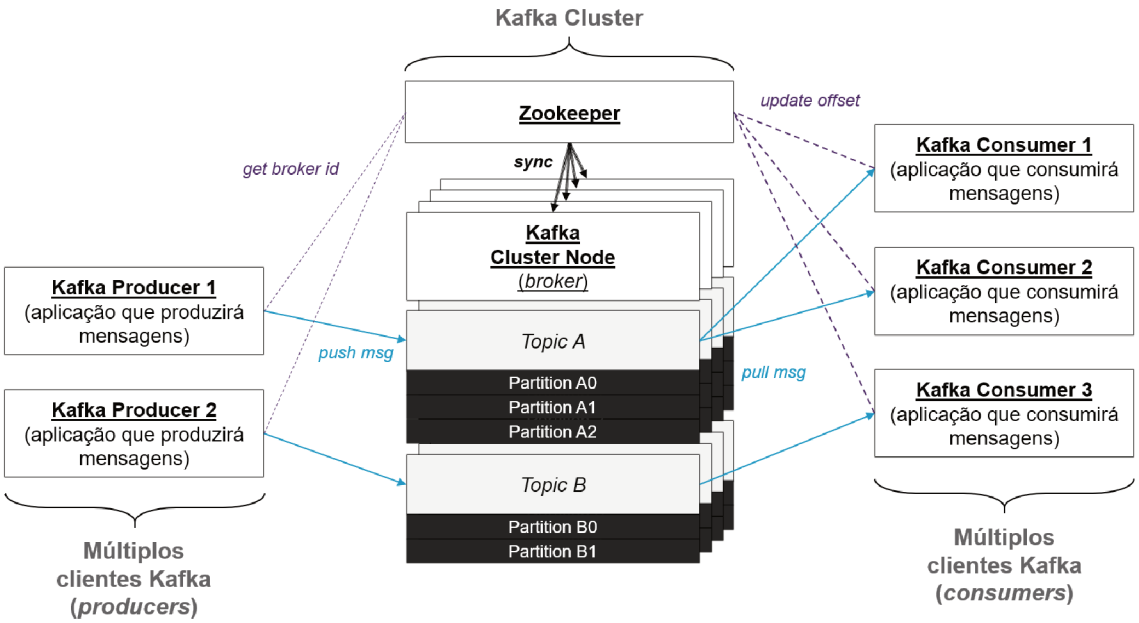
O Kafka pode ser executado em um ou múltiplos servidores – geralmente ele é executado em múltiplas máquinas diferentes. Cada nó em um *cluster* Kafka é um servidor *stateless*, também chamado de *broker*, que é replicado de maneira a escalar o sistema. O Kafka utiliza o *software* Zookeeper para rastrear o status do *cluster*, devido à homogeneidade dos nós. O Zookeeper é um *software* também desenvolvido pela Apache Software Foundation, com foco na centralização de serviços distribuídos. Ele é utilizado para rastrear e armazenar configurações e o estado dos *brokers* (servidores Kafka). O Zookeeper age como um serviço compartilhado de configuração do sistema, e também realiza a eleição de um *broker* líder, que é responsabilizado por gerenciar o

[30] O LinkedIn é uma rede social voltada para negócios, e em 2015 possuía mais de 347 milhões de usuários em todo o mundo.

estado das partições e réplicas das mesmas, além de realizar tarefas administrativas. No caso de falha de um *broker* líder, o Zookeeper realiza a eleição de um novo líder de maneira automática, garantindo que sempre exista apenas um *broker* líder a todo instante e todos os nós estejam sincronizados quanto a isso.

Na Figura 17 é esquematizada a arquitetura de um *cluster* Kafka, mostrando como cada cliente e os servidores (*brokers*) do *cluster* Kafka se comunicam, incluindo o papel do Zookeeper.

Figura 17 – Arquitetura do Apache Kafka.



Fonte: Arquivo pessoal.

As mensagens (chamadas usualmente de *records*) são armazenadas em uma estrutura chamada tópico (*topic*). Cada *record* é composto por uma *key*, *value*, e *timestamp*. Os tópicos, por sua vez, são particionados, e as partições são replicadas e distribuídas entre diferentes *brokers* garantindo tolerância a falhas e resiliência aos dados. Uma partição é basicamente uma sequência ordenada de mensagens.

Existem basicamente quatro tipos de clientes Kafka – que produzem ou consomem mensagens – cada um baseado em uma interface de aplicação com um objetivo diferente. Para clientes do tipo *connector*, a API do Kafka permite a conexão de tópicos do *cluster* a bancos de dados, permitindo a captura de todas alterações que ocorrerem em uma tabela, por exemplo. Para clientes do tipo *producer* ou *consumer*, a API do Kafka permite que essas aplicações respectivamente publiquem ou consumam mensagens de um tópico. Para clientes do tipo *stream processors*, a API do Kafka permite a criação de código capaz de transformar um fluxo de mensagens de um tópico e escrever os resultados em outro tópico, em tempo real, efetivamente transformando

um fluxo de entrada em um fluxo de saída de mensagens. Toda a comunicação entre os clientes Kafka e os *brokers* do *cluster* são realizadas via protocolo TCP binário (exceto quando é utilizada a API REST administrativa, ou a API REST para produção de mensagens, pois ambas realizam comunicação via HTTP).

A Apache Software Foundation provê clientes Kafka oficiais escritos em linguagem Java, entretanto existem dezenas de clientes escritos em outras linguagens e mantidos pela comunidade de desenvolvedores. Outro detalhe importante, é que devido aos servidores Kafka serem *stateless*, é responsabilidade do cliente rastrear as mensagens que foram consumidas por ele. Isso significa que um cliente pode retroceder deliberadamente e reconsumir mensagens antigas, o que viola o comportamento normal de uma fila – entretanto essa é uma característica essencial para muitos tipos de aplicações consumidoras de mensagens.

2.11 SCYLLA: BANCO DE DADOS DISTRIBUÍDO DE TEMPO-REAL

O Scylla é um projeto de código-aberto de banco de dados distribuído e NoSQL escrito em linguagem *C++*. Lançado em 2015, o projeto nasceu baseado no Apache Cassandra (lançado em 2008), outro banco de dados NoSQL distribuído de código-aberto, mantido pela Apache Software Foundation. O principal objetivo do Scylla era superar as fraquezas da JVM[31] – já que o Apache Cassandra é escrito em Java – e fornecer desempenho superior. Os criadores do Scylla alegam que o banco de dados é capaz de tratar um número de operações de 10 à 50 vezes superior, em diversos testes realizados (SCYLLADB INC., 2018). A *Memory Solutions Labs*, divisão de pesquisa da Samsung, realizou testes comparativos entre os dois bancos de dados distribuídos, e conferiu grande parte das alegações sobre o Scylla (REZAEI; GUZ; BALAKRISHNAN, 2017). A Organização Europeia para Pesquisa Nuclear, mais conhecida como CERN, utilizou o Scylla para armazenar as informações do projeto ALICE (*A Large Ion Collider Experiment*), destinado ao estudo do plasma de partículas subatômicas. O projeto ALICE precisava armazenar dados em uma taxa de 2.5 GBytes/s, bem superior ao comum para experimentos do tipo (CERN, 2019).

O Scylla utiliza uma abordagem *shared-nothing* para tratar os dados distribuídos, isto é, cada nó de um *cluster* Scylla não pode acessar memória de maneira independente. Desta forma, os nós nunca tentam atualizar o mesmo registro simultaneamente. A abordagem *shared-nothing* elimina pontos de falha, permitindo também que o *cluster* continue operando apesar de falhas em nós individuais (STONEBRAKER, 1986). A abordagem *shared-nothing* do Scylla é aprimorada por meio da replicação de dados que são muito acessados, porém pouco alterados, permitindo que mais requisições sejam tratadas em um único nó do *cluster*.

[31] A *Java Virtual Machine* aparta do sistema operacional a execução de código Java, visando a portabilidade de código para diferentes sistemas operacionais.

Os *clusters* Scylla adotam uma topologia de rede do tipo anel, onde – como citado previamente – todos os nós são homogêneos (dada a abordagem *shared-nothing*). Os dados são armazenados em *keyspaces*, que são basicamente um conjunto de tabelas com atributos que definem como os dados são replicados em cada nó, de modo análogo à uma base de dados no mundo SQL tradicional. É no *keyspace* que parâmetros como o fator de replicação (RF) são configurados. As criações de tabelas nos *keyspaces* do Scylla, são feitas por meio da *Cassandra Query Language* (CQL). As tabelas são armazenadas em partições do Scylla, que são basicamente subconjuntos de dados armazenados e replicados nos nós.

O Scylla provê alto desempenho principalmente ao uso de um motor dedicado para computação assíncrona chamado Seastar[32], que é basicamente uma biblioteca *C++* que permite que o Scylla acesse diretamente a memória RAM, e de maneira assíncrona, por meio de *Direct Memory Access (DMA)*[33]. As técnicas de otimização utilizadas pelo Scylla e a biblioteca *C++* Seastar fogem do escopo deste trabalho, entretanto podem ser encontrados detalhes e outros textos sobre o assunto na página oficial do projeto Scylla[34].

Concluindo o Capítulo 2, todas as tecnologias e conceitos apresentados até então são aplicados diretamente na construção do sistema proposto pelo autor e em um sistema protótipo já existente, descrito nos próximos Capítulos.

[32] Página oficial em http://seastar.io/ (acessado em: 13/01/2020).

[33] Direct memory access (DMA) é uma característica de computadores modernos que permite que certos componentes de hardware acessem a memória RAM do sistema de maneira independente da CPU.

[34] Acessível em `https://www.scylladb.com/2017/10/05/io-access-methods-scylla/` (acessado em 13/01/2020).

3 PROBLEMÁTICA DO PROJETO

Conforme a prévia apresentada na Seção 1.3, a problemática atacada neste trabalho é centrada no desenvolvimento de um sistema para avaliar automaticamente soluções de *software* submetidas em competições de programação da IBM, na qual diversas ferramentas e serviços de computação em nuvem da corporação podem ser utilizados.

O detalhamento minucioso dos desafios não se faz necessário para o escopo deste trabalho, dado que as próximas competições certamente não utilizarão desafios idênticos aos das competições passadas. Entretanto, é descrito superficialmente na Seção 3.1, um dos desafios de uma das competições anteriores, que é baseado na ferramenta *Watson Knowledge Studio* (WKS) – detalhada na Seção 2.9.5 do capítulo de fundamentação técnica e teórica. A descrição desse desafio servirá apenas para facilitar o entendimento do funcionamento do sistema de pontuação, e alguns pontos críticos a serem trabalhados.

Devido ao fato de muitos dos problemas atacados neste trabalho serem atrelados ao funcionamento de algumas tecnologias utilizadas no antigo sistema de pontuação automática, é recomendada a leitura prévia do Capítulo 2, sobretudo a Seção 2.9.3, caso o leitor não seja familiarizado com o conceito de computação *serverless*[1]. Na Seção 3.2 é descrito de maneira detalhada o suficiente o sistema piloto e os principais problemas que levam o autor desta monografia a propor uma nova arquitetura.

3.1 OS DESAFIOS DA COMPETIÇÃO: ESTUDO DE CASO

Na competição piloto, um dos vários desafios baseava-se na ferramenta *Watson Knowledge Studio* da *IBM Cloud*. Essa ferramenta fornece um arcabouço conceitual para a construção de modelos de anotação textual personalizados, capazes de identificarem entidades e relações em palavras e frases de texto em linguagem natural. Em outras palavras, texto não-estruturado é dado como entrada para uma função “caixa preta” gerada com a ferramenta e exposta via API, e um JSON[2] com uma lista de entidades definidas pelo usuário e trechos de texto correspondentes é retornado como saída. O desafio era baseado na criação de um modelo anotador capaz de identificar algumas entidades pré-definidas. O participante iria então usar o WKS para criar esse modelo, e submeter as credenciais de acesso da API para que o sistema avaliador pontuasse sua solução.

[1] Conhecido na língua portuguesa como “computação sem servidor”, a computação *serverless* é um paradigma de execução de código através do uso de serviços de computação em nuvem, onde os recursos são alocados dinamicamente e apenas o que é de fato utilizado é cobrado (não existe um “servidor” fixo contratado).

[2] JSON, um acrônimo de *JavaScript Object Notation*, é um formato compacto de troca de dados simples e rápida entre sistemas.

Esse desafio era baseado em soluções tecnológicas para o sistema judiciário brasileiro. Nos processos de resolução de conflitos, as fases de mediação e conciliação concluem com a elaboração de um termo de audiência do acordo a ser realizado – documentos esses que não são padronizados e possuem complexidade variada. No sistema jurídico brasileiro, cada termo de audiência de acordo é lido e interpretado por um juiz de carreira, que analisa cada caso e ordena a execução dos acordos entre os envolvidos. O desafio em questão baseava-se no desenvolvimento de um modelo anotador de texto capaz de extrair diretamente os elementos importantes de cada documento, como título do conflito, tipo de acordo, termos do acordo, e nome das partes envolvidas. O propósito da solução criada seria auxiliar um juiz a ordenar a execução dos termos acordados, acelerando o processo de homologação.

Além do desenvolvimento do modelo anotador com a ferramenta WKS da *IBM Cloud*, existiam outras etapas no desafio, como a correção de parte do código-fonte fornecido e a implantação de uma aplicação *Web*, na qual o participante era capaz de enviar sua solução para ser pontuada. O algoritmo pontuador apenas avaliava a qualidade do modelo anotador de texto criado – as outras etapas eram meras barreiras para a submissão da solução.

3.2 O SISTEMA PILOTO IMPLEMENTADO

O sistema piloto pode ser dividido em quatro grandes componentes: 1 – a página *Web* do evento, onde os participantes encontravam informações sobre a competição, formulário de inscrição, e *rankings* em tempo real; 2 – a camada de retenção de dados, que armazenava as informações em duas tabelas (inscritos e submissões dos desafios); 3 – um grande número de funções *serverless* que realizavam todas as tarefas de integração entre as páginas *Web* e os bancos de dados, além de implementarem os algoritmos de pontuação de cada desafio; e 4 – uma página *Web* no estilo painel, que era utilizada internamente pela equipe organizadora para acompanhar métricas e estatísticas sobre a competição. Com exceção da principal página *Web* do evento que foi delegada a uma empresa terceira, todos os outros componentes eram hospedados na própria *IBM Cloud*.

3.2.1 O gerenciamento de inscrições

Todo o sistema responsável por realizar o processo de inscrição de novos participantes (incluindo mecanismos de comunicação via *e-mail*), foi desenvolvido por uma empresa prestadora de serviços. A equipe da IBM responsável pela competição tomou a decisão de terceirizar o desenvolvimento dessa parte do sistema para que os poucos desenvolvedores disponíveis para a preparação do evento se dedicassem no *design* dos desafios e nos algoritmos pontuadores.

Foram criadas pela prestadora de serviços uma página *Web* para o evento, assim como sistema de cadastro na competição (o usuário não possuía *login* e senha apenas cedia algumas informações pessoais e confirmava sua inscrição através de um *link* de confirmação enviado por *e-mail*). Na página *Web* do evento também era possível ver os *rankings* de cada desafio, e um *ranking* geral. Para a integração dessa aplicação *Web* com a camada de dados do sistema, a IBM providenciou para os desenvolvedores da empresa terceirizada a API de algumas funções *serverless*: uma que era capaz somente de inscrever novos participantes na competição, e outra que retornava um *ranking* com os 100 melhores participantes em cada desafio, que eram apresentados na página oficial do evento.

No contexto da problemática atacada neste trabalho, o principal objetivo em relação a este componente é o seu desenvolvimento e implantação por completo na *IBM Cloud*, dispensando a contratação dos serviços de terceiros. Esse componente possui requisitos críticos de desempenho, como funcionamento 24/7 sem quedas de serviço independentemente da quantidade de usuários acessando-o. Os requisitos de segurança também são rigorosos, e importantíssimos – ataques do tipo DDoS[3] são uma ameaça séria para um futuro evento de grande porte, e mecanismos de combate contra esses tipos de ataque devem ser implementados no novo sistema.

Ademais, a IBM também é obrigada pela legislação federal brasileira a implementar diversos mecanismos que garantam a privacidade e o controle sobre os dados pessoais dos participantes. Como a intenção é expandir o próximo evento para outros países, a ideia é construir o novo sistema de maneira a utilizar o mínimo possível de informações pessoais dos participantes, e armazenar qualquer tipo de dado de maneira totalmente anonimizada e segura de intrusos. Com mecanismos adicionais de controle, os participantes poderão facilmente requisitar a remoção completa de suas informações de qualquer banco de dados ou mídia de retenção da IBM. A implementação desses mecanismos simples descritos até aqui já é suficiente para cumprir as leis de privacidade mais rigorosas da atualidade, e casos específicos poderão ser tratados futuramente.

No sistema piloto, essa página *Web* era implementada em sistema *Wordpress*[4].

3.2.2 O tratamento e a pontuação das soluções submetidas

O participante, após concluir sua solução (por exemplo seu modelo anotador de texto, continuando o caso apresentado na Seção 3.1), realizava a submissão por meio de uma requisição baseada em HTTP para uma função *serverless* da *IBM Cloud*

[3] Tipo de ataque cibernético que tenta tornar os recursos de um sistema indisponíveis para os seus utilizadores, por meio de invalidação por sobrecarga. O acrônimo provém do termo em língua inglesa: *Distributed Denial of Service*.

[4] *Wordpress* é um sistema livre e aberto de gestão de conteúdo para Internet, baseado na linguagem de programação PHP e lançado em 2003.

– essa requisição era realizada através de uma aplicação *Web*, executada opcionalmente na máquina do próprio participante, ou na *IBM Cloud* (aplicação escrita em *Node.js*). Na requisição, o participante enviava as credenciais de acesso da API onde o modelo anotador de texto era exposto, e também o seu CPF[5] (para fins de identificação). Com essas informações, uma chamada sem texto de entrada era realizada para o modelo do participante apenas para verificação inicial de que as credenciais fornecidas eram válidas, e o modelo era capaz de ser avaliado. Caso a resposta da requisição indicasse uma API de modelo funcionando corretamente, outra função *serverless* era engatilhada para escrever os dados da submissão em uma tabela de um banco de dados *PostgreSQL*[6]. Caso a resposta da requisição indicasse uma falha na API do modelo submetido para avaliação, um erro era retornado para o participante e a solução não era submetida. Também era verificado se o CPF do usuário era cadastrado na competição.

Monitorando a tabela do banco de dados *PostgreSQL*, outra função *serverless* era engatilhada sempre que uma atualização era realizada na tabela – isto é, quando uma nova solução era submetida. Essa função iniciava o processo de pontuação, que consistia em diversas chamadas HTTP para o modelo de linguagem natural criado pelo participante com o WKS. O algoritmo de pontuação comparava então os trechos de texto anotados pelo modelo do participante com pares de *strings* e nomes de entidades que formavam um gabarito. Também eram realizadas pelas funções *serverless* envolvidas na pontuação das soluções, diversas chamadas HTTP para a escrita de *logs* em um banco de dados *MongoDB*[7] não-relacional, hospedado na *IBM Cloud*.

Na Figura 18 é apresentado um esquemático da arquitetura do sistema piloto – simplificado para o estudo de caso do desafio de linguagem natural. O fluxo de dados no sistema é representado por flechas azuis, enquanto que a escrita de *logs* no banco de dados *Mongo* é representada por flechas cinzas. Alguns fluxos de informação e componentes (como por exemplo a comunicação de função *serverless* para checagem de CPF no banco de dados) foram omitidos do esquemático para fins de simplificação.

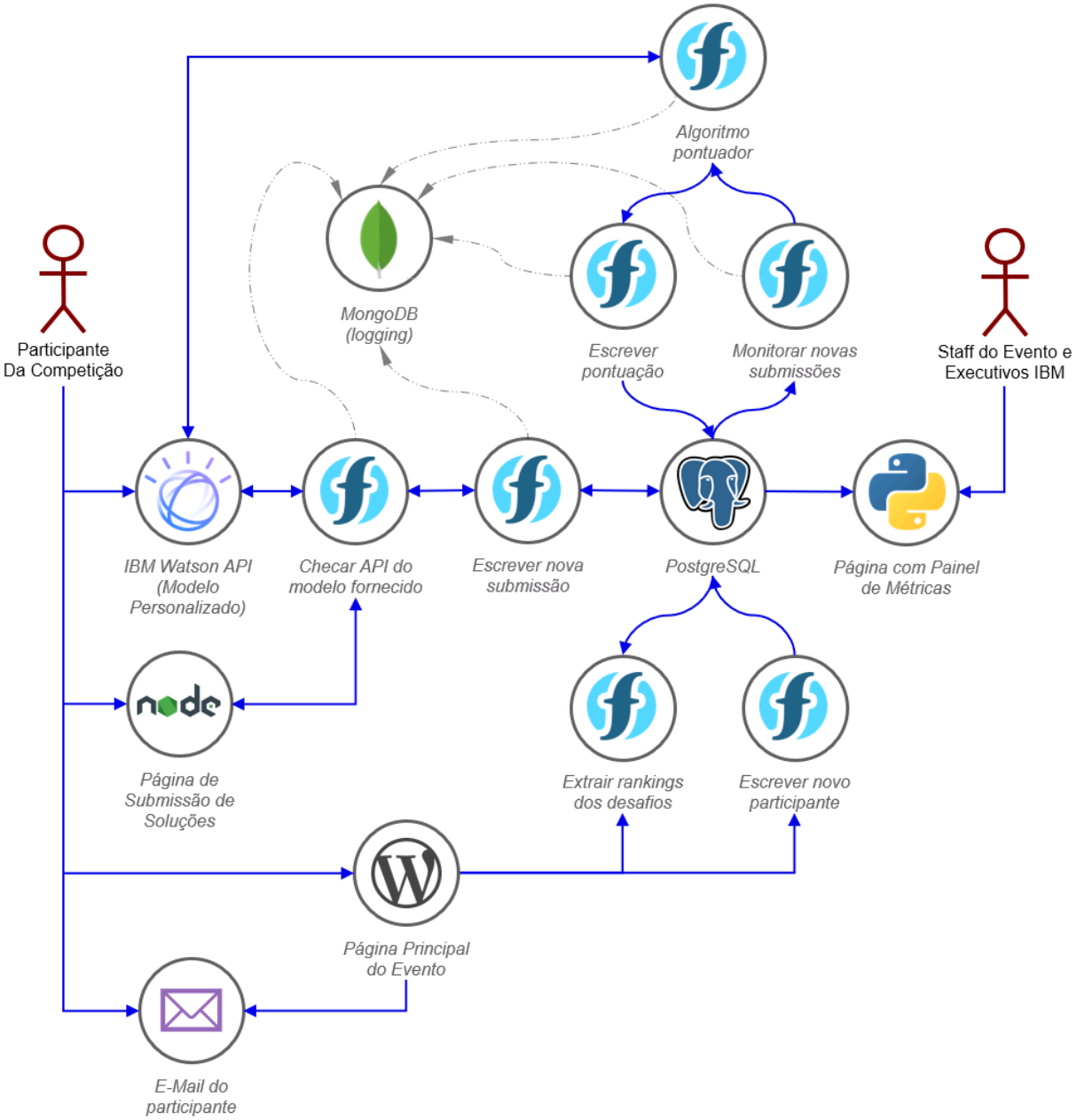
No contexto da problemática atacada neste trabalho, o uso de diversas funções *serverless* encadeadas, embora facilite em muito o desenvolvimento das aplicações e reduza drasticamente os custos de implantação, acaba acarretando em sérios problemas de latência. Isso se deve principalmente ao fato de que cada chamada de função *serverless* é de fato uma requisição HTTP – a grande quantidade de funções encadeadas forçava a propagação de várias requisições. Somado a isso, a função *serverless* que implementava o algoritmo pontuador executava também várias requisições HTTP

[5] Cadastro de Pessoa Física. Documento emitido pela Receita Federal do Brasil para a identificação dos contribuintes.

[6] O *PostgreSQL* é um dos sistemas gerenciadores de bancos de dados relacionais mais populares da atualidade, desenvolvido como projeto de código-aberto e lançado em 1996.

[7] O *MongoDB* é um sistema de gerenciamento de banco de dados não-relacional de código-aberto, lançado em 2009.

Figura 18 – Arquitetura simplificada do sistema piloto.



Fonte: Arquivo pessoal.

para as APIs dos serviços da *IBM Cloud* usados na solução dos desafios, e a escrita de *logs* em um banco de dados MongoDB também era realizada via chamada de API HTTP. No geral, eram envolvidas mais de 30 requisições na submissão e pontuação de um único desafio.

As altas latências causavam outro problema, mais sério: as funções *serverless*, por padrão, possuem um tempo limite de expiração – isto é, caso nenhuma resposta tenha sido retornada até o instante de expiração (calculado a partir da inicialização da função), é forçada uma resposta com código de erro do tipo *timeout*, e toda a

informação é perdida. Isso ocorria com uma frequência preocupante no sistema piloto, sobretudo devido ao fato de que as funções *serverless* eram encadeadas – uma era o gatilho da outra – e as respostas dependiam de uma sequência de requisições aninhadas. Para sanar esse problema, as soluções submetidas que provocavam esse erro precisavam ser pontuadas com a execução manual de um *script* "monolítico" de pontuação, que envolvia apenas chamadas HTTP para os serviços da *IBM Cloud*, e não possuía prazo de expiração na execução, e nem geração de *logs*.

Observando as consequências, fica aparente que o uso de funções *serverless* encadeadas é inadequado para a aplicação desejada. Entretanto, deve-se ser considerado o contexto de desenvolvimento do sistema piloto, que foi construído por vários programadores e com diferentes linguagens. A arquitetura do sistema piloto foi escolhida principalmente devido à fácil integrabilidade e ao curto prazo para desenvolvimento e entrega do produto final.

3.2.3 A aplicação Web para análise de métricas

O módulo de análise de métricas do sistema piloto era implementado como uma aplicação *Web* no estilo painel, ou *dashboard*. O painel de métricas possuía sub-páginas para cada desafio, com tabelas apresentando o *ranking* e informações sobre todos os participantes e soluções submetidas. A aplicação era implementada nas linguagens *Python* e *Javascript*, e não possuía requisitos rigorosos de desempenho (apenas poucas pessoas internas à IBM acessavam essa aplicação). Eram geradas visualizações interativas dos dados da competição, como gráficos de séries de tempo de número de inscritos, submissões realizadas, e também número de inscrições por período de tempo, permitindo que a equipe averiguasse a eficácia dos meios de divulgação utilizados durante o evento.

No contexto da problemática atacada neste trabalho, diferentemente dos outros componentes, o painel de análise de métricas necessita de apenas alguns ajustes, como alterações no *design* e adaptação para acesso adequado em sistemas móveis.

No próximo Capítulo é descrito todo o processo de planejamento e linha de raciocínio do autor para a criação do sistema proposto neste trabalho.

4 PLANEJAMENTO

O objetivo deste capítulo é formalizar as especificações do sistema de *software* desenvolvido pelo autor, detalhando os requisitos de desempenho, funcionalidades, interfaces, e métodos de desenvolvimento. Também são detalhados os ambientes de operação do sistema final.

Naturalmente, o novo sistema herdou grande parte dos requisitos do sistema protótipo já existente. A maioria destes requisitos havia sido levantada a partir de discussões com os membros da equipe organizadora do evento piloto. Além destes requisitos, outros foram levantados conforme o evento piloto foi sendo executado, a pedido de lideranças internas e também pela própria equipe organizadora, conforme se tornava aparente a necessidade de alguma nova funcionalidade.

Toda a nova arquitetura de sistema proposta neste trabalho, apresentada na Seção 4.3.2, foi definida pelo autor baseando-se nos requisitos e especificações que foram formalizados durante a etapa de planejamento. Vale ressaltar que os requisitos e definições apresentadas neste capítulo estão em sua forma final, e passaram por um processo iterativo de avaliação e modificações conforme este trabalho, e o trabalho prévio com o sistema piloto, foram sendo executados.

4.1 ESCOPO E VISÃO DO PROJETO

4.1.1 Objetivos de negócio e critérios de sucesso

O objetivo de negócio principal é proporcionar uma experiência excelente aos participantes, que devem ser capazes de submeter suas soluções e receber um relatório de avaliação detalhado no menor tempo possível. O sistema deve ser completamente isolado e as interfaces expostas (aplicações *Web*) devem ser desacopladas do banco de dados, de maneira a proporcionar uma maior segurança e maior modularização da aplicação. Todos os dados sensíveis de participantes, se for o caso, devem ser anonimizados e criptografados. O serviço de avaliação e o portal do evento (aplicação *Web* principal), devem estar disponíveis 24 horas por dia de maneira ininterrupta.

4.1.2 Visão da solução

A solução deverá ser modular e robusta quanto a falhas e quedas de serviço. Na etapa de projeto já é previamente pensada a implantação em contêineres replicados em um *cluster* multirregião. O sistema utilizará um barramento de dados para desacoplar o banco de dados das aplicações expostas aos participantes. O *software* que irá executar a pontuação das submissões será desenvolvido em linguagem de programação *Rust*, visando primariamente segurança e alto desempenho.

4.1.3 Escopo e limitações

Dadas as restrições de tempo para a execução do projeto de fim de curso, o autor optou por focar na construção do *back-end* do sistema, deixando de lado os aspectos visuais de *design* das aplicações que serão acessadas pelos participantes. O sistema será pensado de maneira a facilmente integrar novos algoritmos de validação, conforme forem definidos os desafios das competições futuras. Para este trabalho, iremos considerar um desafio fictício bem semelhante a um desafio que foi executado no evento piloto, baseado na ferramenta *Watson Knowledge Studio*, detalhada na Seção 2.9.5.

4.2 FORMALIZAÇÃO DOS CASOS DE USO

Os casos de uso do sistema foram inicialmente formalizados pelo autor no documento *Use Cases Document*, disponível no **??**. Como citado anteriormente, os casos de uso foram levantados principalmente a partir de discussões com a equipe organizadora do evento, e também a partir de pedidos de lideranças internas na empresa e experiência própria do autor com o sistema piloto.

Foram identificados três atores primários: participantes da competição, usuários especiais, e administradores do sistema. Os usuários especiais são executivos ou gerentes da IBM que deverão possuir acesso às métricas e estatísticas sobre o andamento do evento, mas sem permissões administrativas. Os administradores de sistema herdam os casos de uso dos usuários especiais.

Para cada tipo de ator foram levantados os seguintes casos de uso centrais, apresentados na Tabela 3.

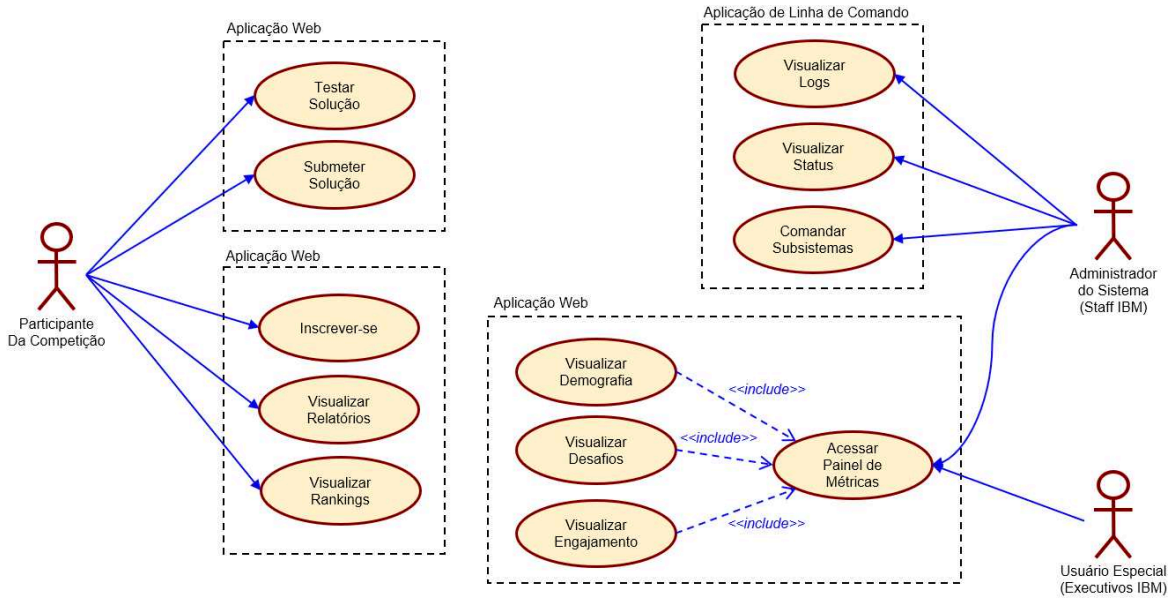
Tabela 3 – Casos de uso do sistema.

Atores primários	**Caso de uso**
Participante da Competição	1. Inscrever-se na competição 2. Testar solução criada 3. Submeter solução para pontuação automática 4. Visualizar relatório de avaliação das soluções pontuadas 5. Visualizar *rankings* da competição
Usuário Especial e Administrador do Sistema	6. Acessar painel de métricas 7. Visualizar métricas demográficas dos participantes 8. Visualizar métricas sobre as soluções de cada desafio 9. Visualizar métricas sobre o engajamento do evento
Administrador do Sistema	10. Visualizar logs dos componentes do sistema 11. Visualizar status dos componentes do sistema 12. Comandar componentes do sistema

Fonte: Arquivo pessoal.

Na Figura 19 são apresentados os casos de uso no formato de diagrama.

Figura 19 – Diagrama de casos de uso do sistema.



Fonte: Arquivo pessoal.

A princípio (quase que certamente pelas regras a serem futuramente definidas para os eventos), administradores e usuários especiais não podem participar da competição, ou seja, não herdam os casos de uso relacionados ao ator Participante. Entretanto, não é prevista a existência de nenhum mecanismo de *software a priori* que impeça um usuário com credenciais de administrador ou usuário especial de inscrever-se e efetivamente participar da competição.

4.3 ESPECIFICAÇÃO DOS REQUISITOS DE SOFTWARE

Os requisitos de *software* do sistema foram formalizados pelo autor no documento *Software Requirements Specification*, disponível no **??**. Alguns requisitos que seriam herdados do sistema piloto foram considerados irrelevantes e omitidos neste novo sistema, enquanto outros aspectos foram tratados e novos requisitos adicionados na especificação, baseando-se na experiência própria do autor com o sistema piloto, orientação dos supervisores, e novas sessões de discussão com a antiga equipe de desenvolvimento.

Primeiramente, são apresentados os requisitos não-funcionais do novo sistema, ambiente de operação e as restrições de implementação. Em seguida, é apresentada a arquitetura proposta para o sistema, incluindo detalhes tecnológicos de implementação. Por último, são tabelados os requisitos funcionais de cada subsistema proposto.

4.3.1 Requisitos não-funcionais

Os requisitos não-funcionais são divididos em três grupos: desempenho, segurança e atributos de qualidade de *software*, apresentados em seguida.

4.3.1.1 Requisitos de desempenho

Como supracitado neste trabalho, os requisitos de desempenho são um dos mais críticos a serem tratados. Foram definidos intervalos de tempos máximos para diversas respostas do sistema, entretanto a intenção é reduzi-los ao máximo possível – os valores definidos servem apenas para criar uma métrica de avaliação, a fim de conferir se o sistema atende suficientemente aos requisitos de desempenho. As métricas foram definidas a partir de discussões com os outros desenvolvedores envolvidos no projeto piloto.

Basicamente, existem dois grandes requisitos de desempenho: o sistema deve suportar um grande número de usuários (participantes) simultâneos, acessando as páginas *Web* e também submetendo soluções para avaliação; e as soluções devem ser pontuadas rapidamente para proporcionar uma ótima experiência aos participantes. A partir dos dados de uso das aplicações coletadas no evento piloto, sabemos que os participantes acessam bastante a página de visualização de *rankings*. Além disso, no evento piloto não era permitida a ressubmissão das soluções (o que reduziu bastante a carga sobre os algoritimos pontuadores) – entretanto, deseja-se projetar o novo sistema com a possibilidade de ressubmissão ilimitada das soluções, o que pode ampliar muito a carga sobre o sistema.

No Quadro 1 são apresentados os requisitos de desempenho do sistema.

Quadro 1 – Requisitos de desempenho do sistema.

ID/Número	**Descrição**
PE-1	Todas as páginas *Web* geradas pelo sistema devem ser totalmente baixadas em, no máximo, 5 segundos, considerando uma conexão de 1 Mbps.
PE-2	O algoritmo de pontuação e geração de relatório de avaliação de uma submissão deve ser executado completamente em no máximo 30 segundos.
PE-3	Todas as páginas *Web* geradas pelo sistema devem permanecer com serviços disponíveis 24 horas por dia *(zero downtime deployment)*.
PE-4	Todos os componentes do sistema devem ser tolerantes a falhas, e uma queda em um nó de serviço não deve comprometer o funcionamento do sistema como um todo, que deve regenerar-se automaticamente.

Fonte: Arquivo pessoal.

4.3.1.2 Requisitos de segurança

Os requisitos de segurança referem-se principalmente aos níveis de acesso de cada recurso, e comprometimento com as regulamentações sobre dados privados e sensíveis envolvidos no projeto.

No Quadro 2 são listados os requisitos de segurança do sistema.

Quadro 2 – Requisitos de segurança do sistema.

ID/Número	**Descrição**
SE-1	Os usuários devem inserir uma senha antes de obterem acesso à página *Web* com informações sensíveis do evento (painel de métricas).
SE-2	Funções administrativas do sistema devem pedir confirmação de maneira explícita antes de realizarem qualquer ação deletéria ou de desligamento.
SE-3	Todos os componentes do sistema deverão implementar mecanismos de *logging* automático para a criação de um histórico rastreável de eventos ocorridos.
SE-4	Toda informação externa que entrar no sistema deve ser criptografada.
SE-5	Toda informação preenchida em formulários deve passar por um *proxy* que irá processá-la por tentativas de ataque (por exemplo, ataques do tipo *SQL injection*).

Fonte: Arquivo pessoal.

4.3.1.3 Atributos de qualidade de software

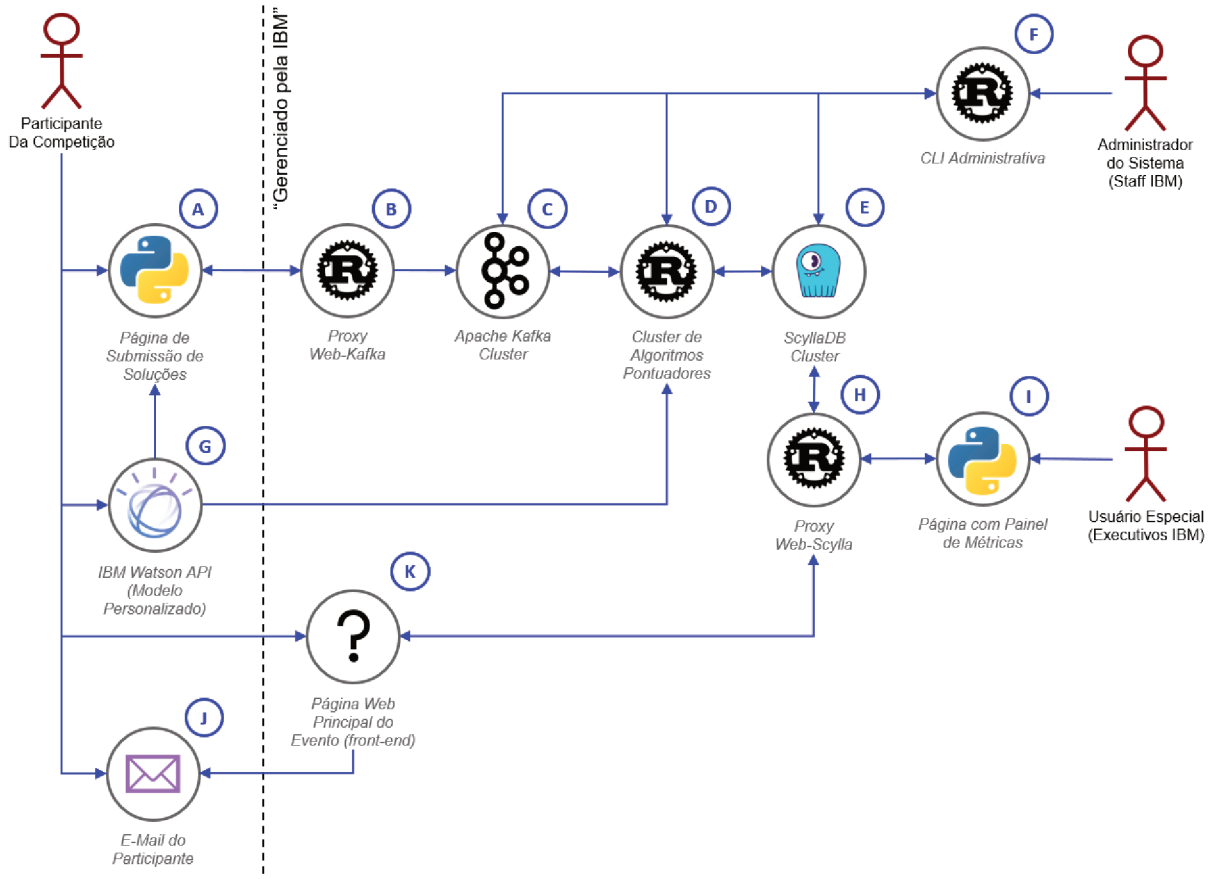
Quanto a atributos de qualidade de *software*, é desejada a interoperabilidade com múltiplos ambientes de implantação como contêineres, máquinas virtuais e servidores *bare-metal*. O *software* deve ser o mais simples possível, possibilitando que novas funcionalidades sejam desenvolvidas por cima do sistema base facilmente, sem a necessidade de muitas alterações pelas futuras equipes de desenvolvimento.

4.3.2 Descrição geral do novo sistema proposto

Dados os casos de uso detalhados na Seção 4.2 e os requisitos não-funcionais detalhados respectivamente na Seção 4.3.1, foi planejado pelo autor um sistema distribuído de módulos replicados. Essa arquitetura, se implantada corretamente, irá garantir o atendimento dos requisitos de disponibilidade de serviço ininterrupto e tolerância a falhas.

Na Figura 20 são esquematizados os módulos do sistema proposto pelo autor e os principais fluxos de dados entre cada componente.

Figura 20 – Arquitetura do sistema proposto.



Fonte: Arquivo pessoal.

No que diz respeito a cada componente, de acordo com cada um dos caracteres de identificação indicados na Figura 20, temos:

a) O componente A será uma aplicação *Web* com código-fonte exposto para o participante. Essa aplicação *Web*, uma vez executada, fornece um interface para teste das soluções desenvolvidas. Também é possível submeter a solução para avaliação, por meio de um clique de botão.

b) O componente B é um *Web proxy* que tratará as submissões enviadas pelo participante, já que o código-fonte do *back-end* da aplicação *Web* de submissão será exposto (e passível de adulteração). Esse componente irá realizar todos os aspectos de validação e segurança dos dados que entrarão no sistema.

c) O componente C irá funcionar como barramento de dados, no esquema de fila, recebendo mensagens do *Web proxy*, e que serão consumidas por múltiplas aplicações consumidoras (representadas pelo componente D). A tecnologia utilizada aqui é o *Apache Kafka*, cujo detalhamento técnico foi apresentado na Seção 2.10.

d) O componente D será uma aplicação cujo objetivo será o consumo de mensagens do barramento de dados (componente C), e a execução do algoritmo de pontuação das soluções. É neste componente onde as computações mais intensivas serão realizadas, incluindo múltiplas chamadas de API para cada modelo cognitivo criado com os *frameworks* da *IBM Cloud* (componente H). Essa aplicação irá gerar um relatório de avaliação, e por fim escreverá todos os dados de maneira permanente no banco de dados (componente E).

e) O componente E será a camada permanente de dados do sistema, onde *logs*, relatórios de avaliação, e informações sobre os participantes serão armazenadas. A tecnologia escolhida para a implementação do banco de dados é o *ScyllaDB*, descrito em detalhes na Seção 2.11.

f) O componente F será uma interface de linha de comando com poderes e credenciais administrativas aos componentes C, D, e E. Administradores do sistema deverão ser capazes de analisarem *logs* e situações em tempo real dos componentes do sistema.

g) O componente G representa de modo genérico os *frameworks* da *IBM Cloud* utilizados na resolução dos desafios. Cada *framework* irá requerer um algoritmo de pontuação diferente. Neste trabalho, como o foco é o funcionamento correto e o desempenho do sistema, foi escolhido apenas um dos *frameworks*, cujo processo de validação era mais custoso computacionalmente e apresentou o maior número de problemas no evento piloto – o *Watson Knowledge Studio*, que é o caso de estudo e é descrito em detalhes na Seção 2.9.5.

h) O componente H será um *proxy Web* para tratar requisições ao banco de dados Scylla. Toda camada de segurança de acesso aos dados será implementada neste componente, que será acessado pelas demais aplicações *Web* que desejarem consultar informações sobre o evento.

i) O componente I será uma aplicação *Web* do tipo *dashboard*, onde usuários especiais serão capazes de analisarem dados sensíveis sobre o andamento do evento, como demografia dos participantes e frequências de submissão, inscrição, e linhas temporais. Em outras palavras, todas as métricas e informações sobre o evento armazenadas no banco de dados serão apresentadas de maneira gráfica e intuitiva por meio dessa aplicação.

j) O componente J representa a caixa de *e-mails* do participante, onde será enviado o endereço de confirmação após a realização da primeira etapa de inscrição no evento, através da página *Web* principal do evento (componente K).

k) Por último, o componente K será a interface *Web* exposta ao público, que poderá realizar inscrição e visualizar os *rankings* da competição, assim como informações gerais sobre o evento. A interface, ou *front-end* dessa aplicação foge do escopo deste trabalho, pois será delegada para equipes de design da IBM. Esse componente irá ser integrado com o *proxy Web* desenvolvido (componente H).

4.3.2.1 Ambiente de operação

O ambiente de operação do sistema final, em uma eventual competição futura, ainda não é definido. Entretanto, sabe-se que todo o sistema de *software* será hospedado em centros de dados da *IBM Cloud*. O autor provisionou uma certa infraestrutura computacional que julgou adequada para a aplicação, entretanto o sistema proposto é projetado de maneira que essa infraestrutura subjacente pode ser escalonada sem a necessidade de alterações no sistema.

Foram provisionadas três máquinas *bare-metal* limpas, através dos serviços de computação em nuvem da empresa. Cada máquina foi provisionada em uma região diferente – mais especificamente Dallas, Washington DC, e Frankfurt – a fim de garantir *zero-downtime deployment* e tolerância a falhas por meio de replicação em diferentes regiões geográficas. Essas máquinas *bare-metal* serão responsáveis por hospedar os componentes C, D, e E. Mais especificamente, o barramento de dados *(cluster Kafka)*, os pontuadores automáticos, e o banco de dados distribuído *(cluster Scylla)*.

Os componentes B, H, e I serão implantados com tecnologia de conteinerização, orquestrados por meio da ferramenta *OpenShift* (descrita em detalhes na Seção 2.8.2). Um *cluster Red Hat OpenShift* pode ser provisionado de maneira automática na *IBM Cloud*, entretanto o autor preferiu instanciar máquinas virtuais e instalar a ferramenta e suas dependências manualmente, para mero fim de aprendizado. Os contêineres de isolamento de cada aplicação serão criados a partir de imagens armazenadas em um repositório privado, o *IBM Container Registry*. Os componentes conteinerizados operarão sobre imagens criadas completamente pelo autor, e descritas em detalhes no Capítulo 5. Essas imagens apenas utilizarão como base as imagens oficiais do *CentOS*[1] e do *Alpine Linux*[2].

O componente A, irá operar nas máquinas pessoais dos participantes, ou na própria *IBM Cloud* caso o participante opte por realizar a implantação da aplicação na nuvem.

Dessa forma, os componentes *stateful* (*Apache Kafka*, *cluster* de pontuadores, e o *ScyllaDB*) serão executados diretamente em máquinas reais, enquanto que os

[1] O *CentOS*, abreviação de *Community ENTerprise Operating System*, é uma distribuição *Linux* de classe corporativa derivada de códigos fonte gratuitamente distribuídos pela *Red Hat Enterprise Linux* e mantida pela organização *CentOS Project*.

[2] O Alpine é uma distribuição Linux mínima e independente, focada em segurança.

componentes *stateless* serão executados em ambiente virtualizado.

4.3.2.2 Restrições de design e implementação

Dentre as restrições de *design* e implementação para o projeto, temos:

a) Atender às conformidades estabelecidas pelas políticas corporativas de privacidade da IBM, que são amparadas pela Regulação Geral de Proteção de Dados Europeia (GDPR) e pela Lei Geral de Proteção de Dados Pessoais Brasileira (LGPDP); e

b) O *hardware* para o projeto, quando funcionando em ambiente de produção, deverá ser provisionado totalmente através da *IBM Cloud*.

4.3.2.3 Pressupostos e dependências de software

Como o ambiente de operação será baseado em contêineres, cuidados deverão ser tomados para que nenhuma dependência fique atrelada ou aberta a atualizações externas. Para tal, todas bibliotecas e dependências de *software* utilizadas no projeto serão compiladas estaticamente nos executáveis, assim como as versões do *OpenShift*, *Apache Kafka* e *ScyllaDB*. Em outras palavras, as imagens a partir das quais os contêineres serão instanciados serão completamente imutáveis, e não existirão referências a *software* externo que porventura possa ser atualizado e causar problemas no sistema.

Na Tabela 4 é apresentada uma relação de versionamento das dependências de *software* que farão parte do sistema.

Tabela 4 – Dependências de *software* pressupostas para a primeira implementação completa do sistema.

Nome	**Versão**	**Descrição**
Red Hat Enterprise Linux CoreOS	2303.3.0 (stable)	Sistema Operacional
CentOS	8.1.1911 (stable)	Sistema Operacional
Alpine Linux	3.11.2 (stable)	Sistema Operacional
Python	3.8.0 (stable)	Ambiente de execução
Java SE (JDK)	1.8.0 (stable)	Ambiente de execução
Rustc	1.40.0 (stable)	Compilador (Rust)
Apache Kafka	2.4.0 (stable)	Pacote de *software* completo (Java)
Apache Zookeeper	3.5.6 (stable)	Pacote de *software* completo (Java)
Scylla	3.2.0 (stable)	Pacote de *software* completo (C++)
Red Hat OpenShift	3.5 (stable)	Pacote de *software* completo (Go)

Fonte: Arquivo pessoal.

4.3.3 Requisitos de interfaces externas

4.3.3.1 Interfaces de usuário

No que referem-se às interfaces expostas aos usuários comuns (participantes), temos:

a) Visualização de página principal *(landing page)* do evento;

b) Formulário de dados para teste de solução criada;

c) Formulário de dados para submissão de solução para pontuação automática;

d) Formulário de dados para inscrição no evento;

e) Visualização dos *rankings* da competição;

f) Visualização dos relatórios de pontuação das soluções submetidas.

No que referem-se às interfaces expostas aos usuários especiais e administradores, temos:

a) Visualização de estatísticas demográficas acerca dos participantes do evento, mais especificamente sobre a distribuição geográfica, gênero, e grau de conhecimento de programação e tecnologia da informação dos participantes;

b) Visualização de métricas acerca da participação e engajamento no evento, como números de acessos, inscrições, e submissões;

c) Visualização de métricas detalhadas acerca de cada desafio e soluções correspondentes.

No que referem-se às interfaces expostas somente aos administradores do sistema, temos:

a) Interface de linha de comando para envio de comandos administrativos para os componentes internos do sistema;

b) Visualização de status e *logs* dos componentes internos do sistema.

Todas as visualizações listadas anteriormente serão implementadas via aplicações *Web*.

4.3.3.2 Interfaces de software

No que refere-se às interfaces de *software*, as seguintes serão necessárias para a integração dos módulos do sistema:

a) Interface de comunicação entre aplicação *Web* (componente A) e *proxy Web-Kafka* (componente B);

b) Interface de comunicação entre *proxy Web-Kafka* (componente B) e barramento de mensagens (componente C);

c) Interface de comunicação entre barramento de mensagens (componente C) e aplicações de pontuação (componente D);

d) Interface de comunicação entre aplicações de pontuação (componente D) e partições do banco de dados *ScyllaDB* (componente E);

e) Interface de comunicação entre aplicações de pontuação (componente D) e *frameworks* da *IBM Cloud* (componente G);

f) Interface de comunicação entre *proxy Web-Scylla* (componente H) e banco de dados *Scylla* (componente E);

g) Interface de comunicação entre aplicações *Web* (componentes I e K) e *proxy Web-Scylla* (componente H).

4.3.3.3 Interfaces de comunicação

A comunicação entre todos os componentes do sistema será realizada utilizando-se o protocolo HTTP, exceto no caso da comunicação entre o *Web proxy* (componente B) e o Kafka, e entre o Kafka e o algoritmo pontuador (componente D), que serão baseadas no protocolo TCP binário para maior desempenho.

Toda comunicação HTTP que passar pelos componentes na "borda" do sistema (componentes B, G, I e K) será criptografada simetricamente fazendo-se uso do protocolo TLS.

4.3.4 Requisitos Funcionais

Como se trata de um sistema distribuído de múltiplos componentes de *software* diferentes, é mais simples a separação dos requisitos funcionais em diferentes módulos.

Nesta Seção são organizados os requisitos de acordo com os seguintes módulos (ou recursos de sistema) principais:

a) Módulo de tratamento de inscrições;

b) Módulo de visualização de *rankings*;

c) Módulo de teste de soluções;

d) Módulo de submissão de soluções;

e) Módulo de pontuação automática;

f) Módulo de visualização de métricas (painel de análise); e

g) Módulo de interface administrativa.

4.3.4.1 Módulo de inscrições

Este recurso de sistema será primariamente responsável pelo processamento e verificação dos dados informados pelo participante no momento da inscrição (por exemplo, números de CPF reais), e também a verificação de duplicidade de inscrições (mesma pessoa se registrando mais de uma vez no evento). Em casos válidos de inscrição, as informações são armazenadas de maneira permanente na camada de dados do sistema. Um esquema de endereço eletrônico de confirmação enviado por *e-mail* é utilizado pelo sistema para finalizar as inscrições. Após uma conclusão do processo de inscrição, uma lista de chaves que serão usadas para autenticar a submissão das soluções criadas pelo participante deve ser gerada. Vale ressaltar que esse sistema de inscrição não funcionará como um sistema tradicional, onde o participante escolhe um nome de usuário e senha. Em outras palavras, não existirá um sistema de *login*.

As sequências de estímulos e respostas do módulo de inscrições são apresentadas na Tabela 5.

Tabela 5 – Estímulos e respostas do módulo de inscrições.

Estímulo	**Resposta**
Usuário solicita inscrever-se no evento.	Sistema apresenta formulário de inscrição.
Usuário submete pedido de inscrição.	Sistema processa pedido de inscrição e envia *e-mail* com endereço eletrônico de confirmação para o participante.
Usuário acessa o endereço eletrônico de confirmação.	Sistema realiza a confirmação da inscrição e informa chaves de submissão para o participante registrado.

Fonte: Arquivo pessoal.

O mecanismo existente para que o sistema saiba quem está submetendo cada solução será baseado na leitura de uma chave informada no ato da submissão. Dessa forma, as equipes responsáveis pelos eventos futuros poderão alterar as regras conforme o desejado, por meio da geração e consumo das chaves de submissão pelo módulo de pontuação (por exemplo, permitir múltiplas submissões por desafio, ou um certo número de tentativas, apenas alterando a lógica de consumo das chaves).

Os requisitos funcionais para o módulo de inscrições são apresentados no Quadro 3.

4.3.4.2 Módulo de rankings

Este módulo será responsável por apresentar abertamente ao público um *ranking* para cada desafio do evento, assim como um *ranking* geral da competição.

As sequências de estímulos e respostas do módulo de *rankings* é apresentada na Tabela 6.

Quadro 3 – Requisitos funcionais para o módulo de inscrições.

ID/Número	**Descrição**
RF1.1	O sistema deve capturar os seguintes dados do usuário no ato da inscrição: nome completo, número de CPF, data de aniversário, *timestamp* do ato de inscrição, gênero, nível de experiência com tecnologia da informação, e cidade de origem.
RF1.2	O sistema deve verificar se o CPF informado pelo usuário no ato da inscrição é um número de CPF válido.
RF1.3	O sistema deve verificar se o CPF informado pelo usuário no ato da inscrição já foi previamente utilizado em uma inscrição.
RF1.4	O sistema deve remover sintaxe (ex: SQL, HTML) e caracteres indesejados dos dados preenchidos pelo usuário no ato da inscrição.
RF1.5	O sistema deve criptografar todas as informações preenchidas pelo usuário no ato da inscrição.
RF1.6	O sistema deve gerar chaves únicas descartáveis para a realização de submissões das soluções dos desafios (a organização do evento pode decidir futuramente a quantidade gerada e a política de uso das chaves).
RF1.7	O sistema deve alertar o usuário em caso de erros no processo de inscrição.
RF1.8	O sistema deve enviar comunicado via *e-mail* com link de confirmação após o participante inscrever-se na competição.
RF1.9	O sistema deve alterar o status do participante de “inativo” para “ativo” após o acesso ao link de confirmação previamente gerado.
RF1.10	O sistema deve enviar comunicado via *e-mail* para o usuário com as chaves de submissão após validação da inscrição.
RF1.11	O sistema deve capturar *timestamps* dos pedidos e também das confirmações de inscrições, para futura análise.

Fonte: Arquivo pessoal.

Os requisitos funcionais para o módulo de *rankings* dos desafios são apresentados no Quadro 4.

4.3.4.3 Módulo de teste de soluções

Este recurso de sistema será primariamente responsável pelo teste das soluções criadas pelos participantes. O resultado dos testes dos modelos de anotação textual criados com o *Watson Knowledge Studio* serão apresentados em formato de tabela, de maneira simples de serem visualizados e compreendidos por um humano.

As sequências de estímulos e respostas do módulo de teste de soluções são

Tabela 6 – Estímulos e respostas do módulo de *rankings*.

Estímulo	**Resposta**
Usuário acessa página de *ranking*.	Sistema busca tabela de participantes no banco de dados do sistema e apresenta para o usuário uma tabela com os 100 primeiros colocados em determinado desafio.

Fonte: Arquivo pessoal.

Quadro 4 – Requisitos funcionais para o módulo de *rankings*.

ID/Número	**Descrição**
RF2.1	O sistema deve possuir um método do tipo *captcha* para evitar o uso de *Web scrappers* ou *crawlers* por agentes externos, antes de apresentar os *rankings* do evento.
RF2.2	O sistema deve realizar consultas ao banco de dados do sistema e construir as tabelas com os 100 primeiros colocados em cada desafio, e na competição geral.

Fonte: Arquivo pessoal.

apresentadas na Tabela 7.

Tabela 7 – Estímulos e respostas do módulo de teste de soluções.

Estímulo	**Resposta**
Usuário solicita realização de teste.	Sistema executa o teste e apresenta resultados.

Fonte: Arquivo pessoal.

Os requisitos funcionais para o módulo de teste de soluções (modelos de anotação textual criados com o WKS) são apresentados no Quadro 5.

4.3.4.4 Módulo de submissão de soluções

Este recurso de sistema será primeiramente responsável pelo processamento e envio das submissões de soluções criadas pelos participantes ao sistema de pontuação automática. No escopo deste trabalho, será desenvolvido apenas o módulo referente ao desafio de compreensão de linguagem natural, baseado na ferramenta *Watson Knowledge Studio* da IBM. Para realizar a submissão de uma solução, uma chave gerada previamente no ato de inscrição no evento deve ser fornecida pelo participante.

As sequências de estímulos e respostas do módulo de submissão de soluções são apresentadas na Tabela 8.

Os requisitos funcionais para o módulo de submissão de soluções são apresentados no Quadro 6.

Quadro 5 – Requisitos funcionais para o módulo de teste de soluções.

ID/Número	Descrição
RF3.1	O sistema deve capturar os seguintes dados do usuário no ato da realização de teste: chaves de acesso do Watson NLU instanciado pelo participante, *endpoint* URL da API do Watson NLU, e ID do modelo criado pelo participante.
RF3.2	O sistema deve remover sintaxe e caracteres indesejados (SQL, HTML) dos dados preenchidos pelo usuário no ato da realização de teste ou submissão.
RF3.3	O sistema deve realizar uma chamada de API para o modelo informado pelo usuário para fins de validação se a solução é possível de ser avaliada.
RF3.4	O sistema deve alertar o usuário em caso de erros no processo de execução de teste da solução (erros de comunicação com o modelo exposto pelo usuário).

Fonte: Arquivo pessoal.

Tabela 8 – Estímulos e respostas do módulo de submissões.

Estímulo	**Resposta**
Usuário submete solução.	Sistema processa submissão da solução e a adiciona à pilha de tarefas do módulo de pontuação. Um aviso de sucesso ou falha na submissão é apresentado.

Fonte: Arquivo pessoal.

4.3.4.5 Módulo de pontuação automática

Neste módulo também será tratado apenas um caso dos diversos tipos de desafios: o desafio de compreensão de linguagem natural. Esse desafio foi escolhido devido a maior complexidade do algoritmo de validação, que acarretava maior latência e ocorrência de problemas. O detalhamento da ferramenta WKS, usada para a criação de modelos anotadores de texto personalizados é apresentada na Seção 2.9.5, e sua leitura é recomendada para a compreensão total dos requisitos desse módulo e suas dependências.

As sequências de estímulos e respostas do módulo de pontuação das soluções é apresentada na Tabela 9.

Os requisitos funcionais para o módulo de pontuação automática das soluções são apresentados no Quadro 7.

4.3.4.6 Módulo de visualização de métricas

Este módulo será responsável por apresentar para os usuários especiais e administradores do sistema dados e estatísticas sobre o andamento do evento. Este módulo é basicamente o componente I apresentado na Figura 20.

Quadro 6 – Requisitos funcionais para o módulo de submissão de soluções.

ID/Número	Descrição
RF4.1	O sistema deve capturar os seguintes dados do usuário no ato da submissão de solução para pontuação: *apikey* do Watson NLU, *endpoint* URL, *model id*, *timestamp* do ato de submissão, e chave de submissão válida, gerada previamente no ato da inscrição.
RF4.2	O sistema deve verificar se a chave de submissão informada pelo usuário no ato da submissão já foi previamente utilizada, ou se continua válida.
RF4.3	O sistema deve remover sintaxe e caracteres indesejados (SQL, HTML) dos dados preenchidos pelo usuário no ato da realização da submissão.
RF4.4	O sistema deve criptografar todas as informações preenchidas pelo usuário no ato da submissão.
RF4.5	O sistema deve alertar o usuário em caso de erros no processo de execução da submissão de solução.

Fonte: Arquivo pessoal.

Tabela 9 – Estímulos e respostas do módulo de pontuação.

Estímulo	**Resposta**
Uma submissão é adicionada à pilha de tarefas pelo módulo de submissões.	Sistema inicia algoritmo de pontuação e geração de relatório. Ao final do processo o relatório é enviado para o autor da solução via *e-mail*, e os resultados são escritos no banco de dados do sistema.

Fonte: Arquivo pessoal.

As sequências de estímulos e respostas do módulo de visualização de métricas (painel de análise) é apresentada na Tabela 10.

Os requisitos funcionais para o módulo de painel de análise são apresentados a seguir, no Quadro 8.

4.3.4.7 Módulo de Interface administrativa

Este módulo será responsável por prover aos administradores do sistema a capacidade de controle e supervisão de alguns subsistemas (Kafka, *Scylla*, e algoritmos pontuadores). Na Figura 20, esse módulo é representado pelo componente F. Essa interface administrativa será uma aplicação de linha de comando acessada somente *on-premises* pelos administradores do sistema (em suas máquinas pessoais).

As sequências de estímulos e respostas da interface administrativa é apresentada na Tabela 11.

Os requisitos funcionais para o módulo de interface administrativa são apresentados a seguir, no Quadro 9.

Quadro 7 – Requisitos funcionais para o módulo de pontuação automática.

ID/Número	**Descrição**
RF5.1	O sistema deve verificar no banco de dados se as credenciais do participante ainda são válidas antes de iniciar o processo de pontuação.
RF5.2	O sistema deve realizar chamadas de API para os modelos anotadores de textos criados e expostos pelos participantes (utilizando as credenciais fornecidas no ato da submissão da solução pelo participante).
RF5.3	O sistema deve comparar todas as entidades e relações de texto encontradas pelos modelos de anotação textual criados pelos participantes com um gabarito pré-definido.
RF5.4	O sistema gera relatório no formato de texto sobre o resultado de cada chamada de API e comparação com gabarito pré-definido.
RF5.5	O sistema deve escrever o relatório final completo de um processo de pontuação no banco de dados do sistema.
RF5.6	O sistema deve enviar alerta e relatório de pontuação via *e-mail* para o participante, ao final do processo.

Fonte: Arquivo pessoal.

Tabela 10 – Estímulos e respostas do módulo de visualização de métricas.

Estímulo	**Resposta**
Usuário acessa página de painel de análise.	Sistema apresenta tela de autenticação.
Usuário se autentica no painel de análise	Sistema realiza consultas no banco de dados e constrói visualizações gerais sobre o evento (histórico de número total de submissões, frequência de submissões por tempo, número e composição demográfica de inscritos)
Usuário acessa seção de dados geográficos	Sistema realiza consultas no banco de dados e constrói mapa coroplético do Brasil baseado no número de inscritos por Estado e/ou município.
Usuário acessa seção de dados específicos de um desafio	Sistema realiza consultas no banco de dados e constrói tabelas detalhadas com os relatórios de avaliação de cada submissão.

Fonte: Arquivo pessoal.

4.4 MÉTODO DE DESENVOLVIMENTO APLICADO

O método de desenvolvimento de *software* escolhido para a construção do projeto é vagamente baseado no método "*cleanroom*", criado por Harlan Mills, Alan Hevner, e outros funcionários da IBM no final dos anos 80 (MILLS; HEVNER, 1987).

O principal foco do método *cleanroom* é entregar como produto final um *software* com alto nível de confiabilidade – o nome do método vem das "salas limpas",

Quadro 8 – Requisitos funcionais para o módulo de visualização de métricas.

ID/Número	Descrição
RF6.1	O sistema deve autenticar o usuário antes de oferecer acesso às informações do evento.
RF6.2	O sistema deve realizar consultas ao banco de dados e gerar as visualizações interativas (séries de tempo, histogramas e mapas coropléticos) automaticamente, em tempo real (sempre que houver atualização nos dados).
RF6.3	O sistema deve realizar consultas ao banco de dados e construir tabelas com detalhes sobre as submissões de maneira automática, em tempo real (sempre que houver atualização nos dados).

Fonte: Arquivo pessoal.

Tabela 11 – Estímulos e respostas da interface administrativa.

Estímulo	**Resposta**
Administrador requere lista de comandos disponíveis.	Sistema apresenta lista detalhada de comandos e opções disponíveis.
Administrador executa comando para leitura de *logs* de um subsistema.	Sistema apresenta todos os *logs* disponíveis para o subsistema definido como argumento no comando.
Administrador executa comando para ver status geral dos componentes do sistema.	Sistema apresenta lista com detalhes sobre o status do banco de dados, do barramento de dados e do *cluster* de algoritmos pontuadores.
Administrador executa comando para enviar comando diretamente para a API de um subsistema.	Sistema pede confirmação, e em caso positivo realiza o comando desejado.

Fonte: Arquivo pessoal.

ambientes controlados para manufatura de componentes onde a contaminação por partículas do ar não pode ocorrer, geralmente utilizados na indústria de semicondutores e laboratórios químicos. Os princípios essenciais do método *cleanroom* são baseados em: aplicação de métodos matemáticos formais na verificação de *software*; testes estatísticos; e desenvolvimento incremental (MILLS; HEVNER, 1987). Um modelo de referência de engenharia de *software* para o método *cleanroom* foi formalizado por (LINGER; TRAMMELL, 1996), e contém 14 processos e 20 produtos de *software* relacionados ao ciclo de desenvolvimento.

Apenas alguns aspectos do método foram aplicados no desenvolvimento do sistema tratado neste trabalho, sobretudo a aplicação de testes estatísticos e desenvolvimento incremental, e isso se deve ao fato de que certos aspectos do método *cleanroom*, como a adoção de métodos formais de análise de *software*, são inviáveis de serem aplicados em sistemas de alta complexidade, principalmente sistemas concorrentes e com múltiplas arquiteturas de implantação, como é o caso deste projeto e

Quadro 9 – Requisitos funcionais para a interface administrativa.

ID/Número	**Descrição**
RF7.1	O sistema deve possuir um método que retorne todos os comandos disponíveis e suas descrições.
RF7.2	O sistema deve ser capaz de comunicar-se com a API do *cluster Kafka* e enviar comandos diretamente a esse subsistema.
RF7.3	O sistema deve ser capaz de comunicar-se com a API do *cluster Scylla* e enviar comando diretamente a esse subsistema.
RF7.4	O sistema deve ser capaz de comunicar-se via SSH com as máquinas *bare-metal* onde o Kafka, o *Scylla* e os pontuadores automáticos estão hospedados.
RF7.5	O sistema deve exibir *prompt* de confirmação quando necessário (para qualquer tipo de comando deletério ou potencialmente perigoso).

Fonte: Arquivo pessoal.

praticamente qualquer outro projeto moderno, mesmo que de pequeno porte.

Sintetizando o método aplicado, foram primeiro formalizados todos os requisitos funcionais e não-funcionais do sistema – apresentados neste capitulo – assim como a definição do sistema completo apresentado na Seção 4.3. Após a definição da arquitetura do sistema, foram definidos os incrementos de *software* a serem realizados pelas equipes de desenvolvimento (na prática, apenas o autor deste trabalho). Após a escrita do código, o mesmo é testado já em ambiente de produção – ou seja, já no mesmo ambiente onde ele seria executado em produção final – e o próximo incremento de *software* só começa a ser desenvolvido após a consolidação do incremento anterior, validado por meio de testes automatizados desenvolvidos pelo autor com o uso de *scripts* personalizados para cada incremento de *software*, além da realização de testes práticos por uma equipe apartada do desenvolvimento.

Os incrementos de *software* de cada componente desenvolvido são tratados no Capítulo 5.

5 DESENVOLVIMENTO

Neste capítulo são detalhadas todas as atividades efetivamente realizadas para o desenvolvimento do sistema proposto e planejado no Capítulo 4. São detalhados os processos de implantação do sistema, assim como os detalhes importantes do código desenvolvido e processos de automação implementados.

O desenvolvimento ocorreu de maneira incremental, partindo dos módulos de *software* "prontos", como o provisionamento de infraestrutura na IBM Cloud e a instalação, configuração e teste do *Apache Kafka*, do banco de dados Scylla, e da plataforma *Red Hat OpenShift*. A partir desse ponto, foi desenvolvido pelo autor uma maneira para comunicação com esses *softwares* na linguagem de programação *Rust* (técnica baseada em *static linking* para bibliotecas de código *C*, descrita na Seção 5.6). Com a possibilidade de integração de aplicações escritas em *Rust* com esses pacotes de *software*, o próximo passo foi o desenvolvimento de uma biblioteca de *software* para comunicação com a API HTTP do Watson – essa biblioteca possuía apenas fins de organização do *software* desenvolvido, pois o *Rust* já possui métodos para a realização de chamadas HTTP (entretanto sem protocolo TLS, cujo suporte foi implementado pelo autor). Com essas três interfaces de *software* testadas, o componente de pontuação automática foi completamente desenvolvido. Posteriormente, o foco foi na construção e integração dos *Web proxies* em *Rust* para o Kafka e o Scylla. Com os *Web proxies* implantados no cluster OpenShift e testados, o processo de desenvolvimento do trabalho passou para a construção das aplicações *Web* do sistema em linguagem *Python*, que são três: a aplicação para teste e submissão de modelos anotadores de texto; a aplicação para tratamento de inscrições e rankings; e a aplicação do tipo painel para análise de métricas. Após a implantação das três aplicações *Web* no *cluster OpenShift* e integração com os *Web proxies* desenvolvidos, o sistema se materializava em sua primeira implementação completa, sobre a qual os testes comparativos de desempenho serão executados.

O desenvolvimento do trabalho ocorreu então de acordo com os seguintes passos incrementais:

1. Provisionamento de infraestrutura de produção na IBM Cloud (máquinas virtuais e servidores *bare-metal*);
2. Instalação e configuração de um *cluster Red Hat OpenShift* e suas dependências, com posterior teste;
3. Instalação e configuração de um *cluster* Kafka e suas dependências, com posterior teste;
4. Instalação e configuração de um *cluster* Scylla e suas dependências, com posterior teste;

5. Desenvolvimento e integração da aplicação de linha de comando administrativa (componente F segundo a referência na Figura 20);
6. Desenvolvimento e integração dos pontuadores automáticos (componente D segundo a referência na Figura 20);
7. Desenvolvimento e integração dos *Web proxies* (componentes B e H segundo a referência na Figura 20);
8. Desenvolvimento e integração do painel de métricas (componente I segundo a referência na Figura 20); e, por fim,
9. Desenvolvimento e integração da aplicação *Web* de teste e submissão de soluções (componente A segundo a referência na Figura 20).

5.1 CONFIGURAÇÃO DO AMBIENTE DE PRODUÇÃO

O ambiente de produção é a infraestrutura computacional na qual o sistema de *software* final irá ser executado, servindo aos usuários. Para tal, conforme o definido pelas especificações de requisitos de *software* – Seção 4.3, todo o *hardware* será provisionado na plataforma de computação em nuvem da empresa, a IBM Cloud. O autor deste trabalho possui acesso a uma conta interna compartilhada, a qual será utilizada para fins de desenvolvimento e teste do sistema. Em uma eventual competição futura, o sistema será portado exatamente como construído e implementado através de uma conta específica para o evento, onde apenas pessoas autorizadas terão acesso aos recursos e dados.

Para o ambiente de produção será necessário o provisionamento de máquinas *bare-metal* para a hospedagem do banco de dados distribuído Scylla e do sistema de troca de mensagens Kafka, além é claro dos pontuadores automáticos escritos em linguagem *Rust*. Alternativamente, tudo pode ser hospedado em ambiente virtualizado (simplificando tremendamente o trabalho de equipes de DevOps), e neste trabalho foram preparadas imagens Docker para tal. A escolha pelo uso de ambiente *bare-metal* se faz para estudo do ganho de desempenho e avaliação da relação custo-benefício entre os dois ambientes de implantação.

Máquinas virtuais foram escolhidas para a instalação do *cluster OpenShift*, que irá hospedar as aplicações *Web* do sistema. Uma alternativa às máquinas virtuais, é o provisionamento direto de um *cluster OpenShift* pré-configurado pela IBM Cloud (oferta PaaS). Ambos não causam diferença nenhuma no sistema final, considerando que as máquinas virtuais tenham a mesma capacidade de processamento em vCPUs[1] e memória que os nós do *cluster* pré-configurada. Vale notar que o *OpenShift* pré-configurado oferecido através da IBM Cloud pode ser implantado em máquinas virtuais

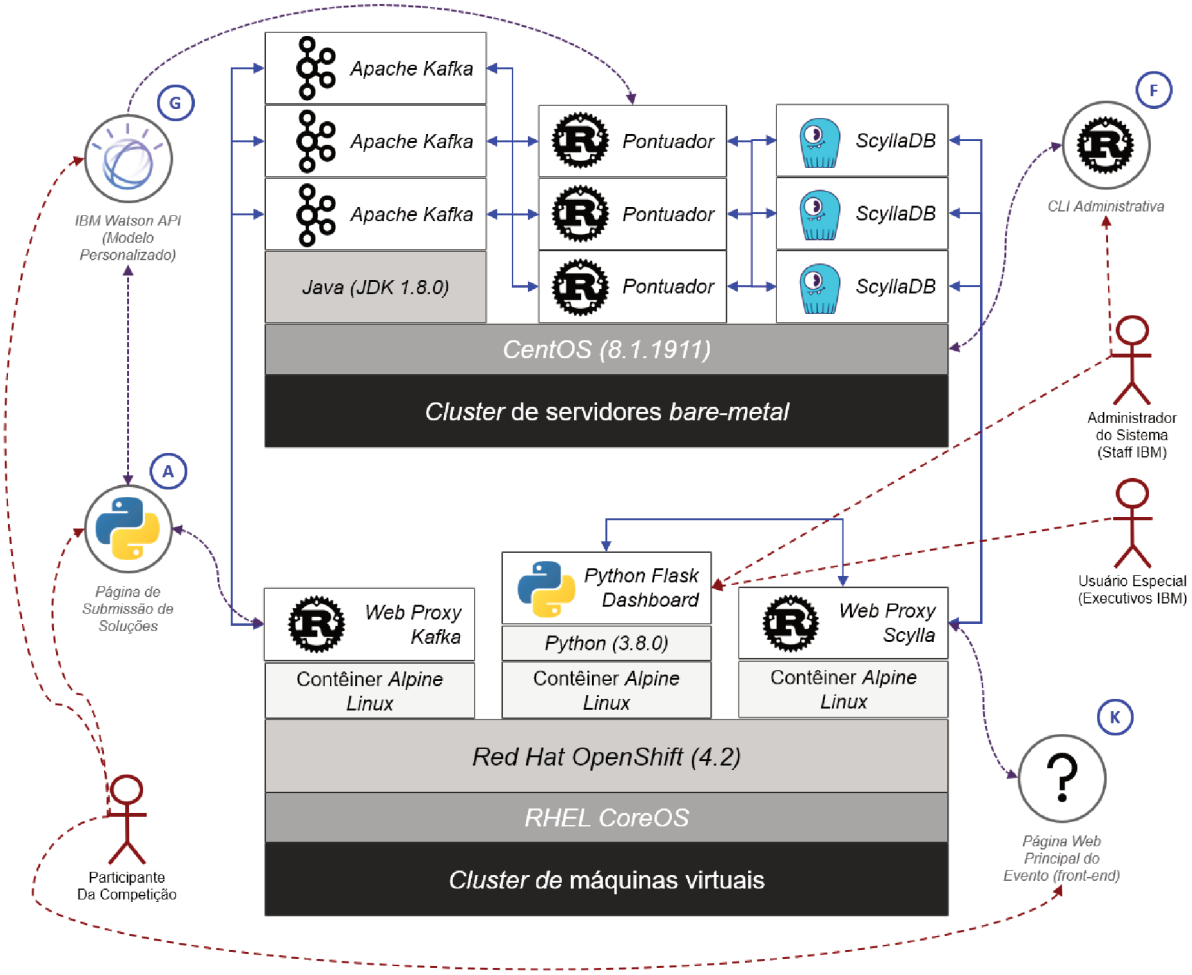
[1] Uma vCPU é um núcleo de processador que é designado a uma máquina virtual.

ou diretamente em máquinas *bare-metal* – entretanto o custo é bem maior no último caso.

Recapitulando, conforme especificado na Seção 4.3, o *cluster OpenShift* será utilizado para a hospedagem das aplicações *Web* do sistema, enquanto que o *cluster* de máquinas *bare-metal* será utilizado para a implantação dos componentes críticos (barramento e banco de dados), assim com os algoritmos de pontuação e geração de relatório das soluções analisadas. A aplicação de linha de comando administrativa será executada localmente, nas máquinas pessoais dos administradores do sistema.

Na Figura 21 é apresentada a arquitetura do sistema proposto, do ponto de vista de operação, ou infraestrutura.

Figura 21 – Arquitetura do sistema da perspectiva de infraestrutura.



Fonte: Arquivo pessoal.

5.1.1 Provisionamento de máquinas bare-metal

Conforme supracitado, os servidores *bare-metal* nos quais serão executados os componentes de retenção de dados e pontuação de submissões serão provisionados

através da IBM Cloud. As máquinas foram provisionadas em três regiões diferentes, a fim de atender ao requisito de *zero-downtime deployment*. Dessa forma, períodos de manutenção em uma região não irão desligar completamente o sistema, a menos que um problema generalizado ocorra simultaneamente nas três regiões, o que possui uma probabilidade de ocorrência negligenciável.

Na Tabela 12 são apresentadas as configurações e regiões das máquinas provisionadas.

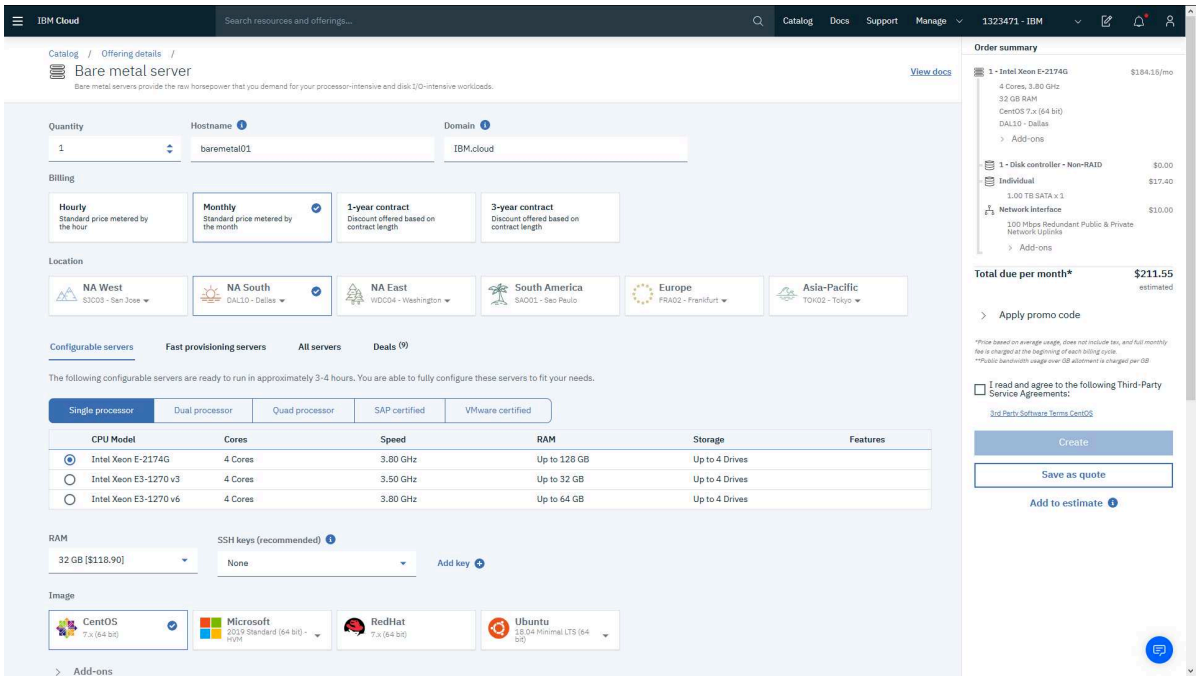
Tabela 12 – Servidores bare-metal provisionados para o sistema.

Localização	**CPU**	**Cores @ Freq.**	**RAM**	**Storage**	**Rede**
Dallas (DAL11)	Intel Xeon 4110	16 @ 2.10GHz	32GB	800GB SSD	1 Gbps
São Paulo (SAO01)	Intel Xeon 4110	16 @ 2.10GHz	32GB	800GB SSD	1 Gbps
Frankfurt (FRA02)	Intel Xeon 4110	16 @ 2.10GHz	32GB	800GB SSD	1 Gbps

Fonte: Arquivo pessoal.

O provisionamento de infraestrutura via IBM Cloud é simples, e pode ser feito via aplicação *Web*. Na Figura 22 temos a tela de provisionamento de servidores *bare-metal* da IBM Cloud.

Figura 22 – Provisionamento de servidores bare-metal na IBM Cloud.



Fonte: Arquivo pessoal.

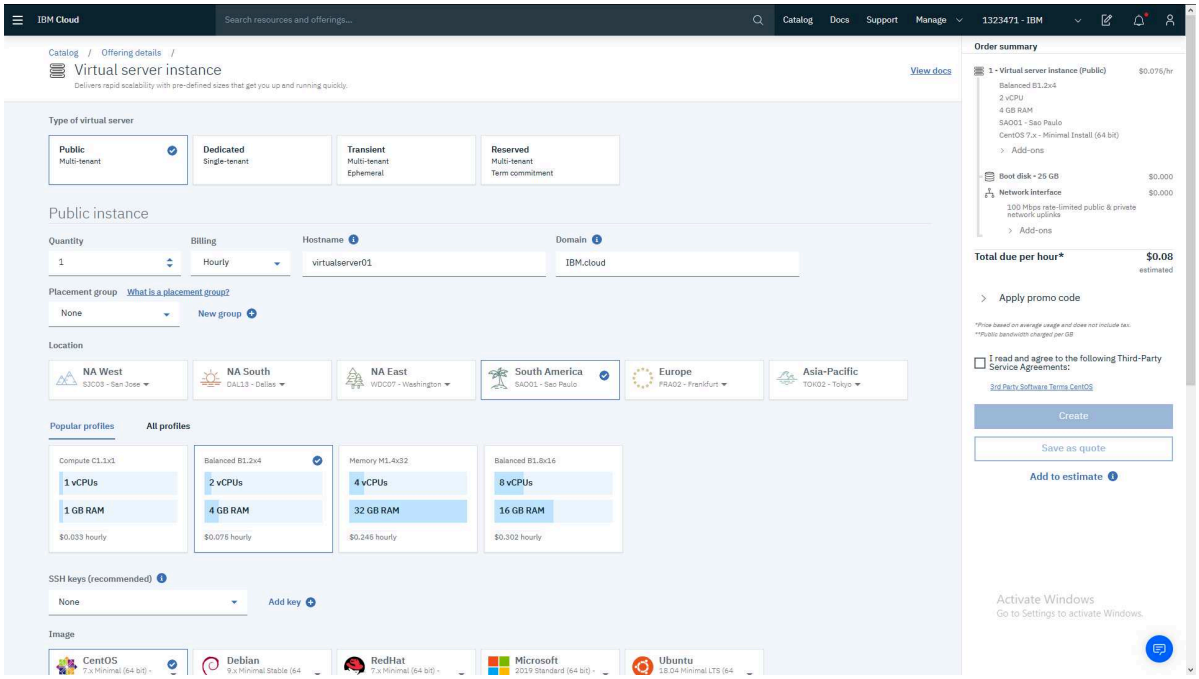
Os três servidores *bare-metal* serão utilizados para hospedar o *cluster* Kafka, e o *cluster* Scylla. Também serão executados neles os algoritmos de pontuação, que

farão proveito da arquitetura *multicore* das máquinas. Na Seção 5.2 é detalhado o processo de instalação do Kafka, e na Seção 5.3 é detalhado o processo de instalação do Scylla. Quanto aos algoritmos pontuadores, na Seção 5.6 é detalhado o processo de desenvolvimento e implantação dos *softwares* nos servidores *bare-metal* provisionados.

5.1.2 Provisionamento de máquinas virtuais

Da mesma forma que os servidores *bare-metal*, as máquinas virtuais também podem ser provisionadas via *console Web* da IBM Cloud. Na Figura 23 temos uma imagem da tela de provisionamento de máquinas virtuais.

Figura 23 – Provisionamento de máquinas virtuais na IBM Cloud.



Fonte: Arquivo pessoal.

Para as aplicações *Web*, conforme explicitado na Seção 4.3, será provisionado um *cluster OpenShift*. É possível utilizar o *OpenShift "as a service"* criado através da IBM Cloud, já pré-configurado. Entretanto, o autor deste trabalho optou por instalar o *OpenShift* diretamente em máquinas virtuais para fins de aprendizado. A configuração mínima para a criação de um *cluster OpenShift* é de 6 máquinas. Para este trabalho foram provisionadas 7 máquinas virtuais. As máquinas podem ser acessadas via protocolo criptográfico SSH, e todo o desenvolvimento e implantação do sistema é realizado de maneira remota.

Na Tabela 13 são apresentados detalhes sobre as sete máquinas virtuais provisionadas na IBM Cloud.

Tabela 13 – Máquinas virtuais provisionadas para o sistema.

Localização	**CPU**	**RAM**	**Storage**	**Rede**
Dallas (DAL11)	4 vCPUs	16GB	25GB SAN	1 Gbps
Dallas (DAL12)	4 vCPUs	16GB	25GB SAN	1 Gbps
Dallas (DAL13)	4 vCPUs	16GB	25GB SAN	1 Gbps
Washington D.C. (WDC01)	4 vCPUs	16GB	25GB SAN	1 Gbps
Washington D.C. (WDC07)	4 vCPUs	16GB	25GB SAN	1 Gbps
São Paulo (SAO01)	4 vCPUs	16GB	25GB SAN	1 Gbps
São Paulo (SAO01)	4 vCPUs	16GB	25GB SAN	1 Gbps

Fonte: Arquivo pessoal.

5.1.3 Instalação e configuração do *cluster* Red Hat OpenShift

O próximo passo na configuração do ambiente de produção do sistema, é a instalação e configuração de um *cluster OpenShift*, a plataforma corporativa de orquestração de contêineres construída em cima do *Kubernetes*, e desenvolvida pela *Red Hat*. É neste *cluster* que as aplicações *Web* do sistema serão executadas, de maneira replicada em três regiões geográficas diferentes (São Paulo, Washington DC, e Dallas).

Como citado anteriormente, para instalar um *cluster OpenShift* são necessárias, no mínimo, seis máquinas: uma máquina para *bootstrap*[2] temporário do *cluster*; três máquinas para formarem o plano de controle (nós mestres); e duas máquinas para execução de aplicações (nós trabalhadores). Os requisitos mínimos de *hardware* para cada máquina são: 4 vCPUs, 16GB RAM, e 120GB e armazenamento.

Após o provisionamento das máquinas e a instalação do sistema operacional RHCOS em todas elas, é necessário configurar a rede de comunicação (conexão via Internet) entre todas as máquinas do *cluster*. Algumas portas TCP (2379-2380, 6443, 9000-9999, 10249-10259, 10256) e UDP (4789, 6081, 9000-9999, 30000-32767) devem ser expostas em todas as máquinas para garantir a comunicação entre diversos componentes, como a API do *Kubernetes*, do banco de pares de valores Etcd, e o *OpenShift Software Defined Networking (openshift-sdn)*, uma abordagem de *software* para a definição de uma rede unificada para o *cluster*. Além disso, é necessária a instalação de dois balanceadores de carga de camada 4 (relativo à camada OSI)[3]: um para o funcionamento da API do *OpenShift*; e outro para que o *cluster* provenha acesso às aplicações implantadas aos usuários externos na Internet.

Ainda em relação aos requisitos de rede para o funcionamento do *OpenShift*,

[2] O *cluster OpenShift* necessita inicialmente de uma máquina para a implantação de nós mestres, e após a instalação inicial essa máquina pode ser removida se desejado.

[3] O balanceamento de carga da camada 4 usa as informações definidas na camada de transporte de rede para construir a política de distribuição de solicitações de clientes em um grupo de servidores.

alguns endereços DNS[4] precisam ser configurados nas máquinas do *cluster* – isso é realizado de maneira declarativa nos arquivos **zonefile.db** e **reverse.db**, e a ferramenta de instalação do *OpenShift* irá automaticamente configurar os endereços DNS nas máquinas. Essas configurações servem para padronizar o acesso de nós do *cluster* uns aos outros e também para que máquinas externas acessem o *cluster* – são necessárias rotas para a API do Kubernetes, para o Etcd, e uma rota para acesso através da Internet para clientes externos.

Ainda antes de iniciar a instalação da plataforma *OpenShift*, é necessária a geração e distribuição de chaves SSH para autenticação sem senha para todas as máquinas do *cluster*. Isso pode ser realizado via geração de arquivos com as chaves através da ferramenta *ssh-keygen*, e posterior distribução para cada máquina do *cluster* – essa tarefa pode ser feita manualmente ou com o auxílio de um simples *shell script*. Isso é necessário pois a ferramenta de instalação do *OpenShift* utiliza comunicação SSH para acionar ferramentas remotamente, e realizar o carregamento e instalação de pacotes e dependências de *software* da plataforma em todas as máquinas do *cluster*.

Após a configuração do SSH, já é possível instalar o *CoreOS* em todas as máquinas. Para tal, será necessário criar um arquivo de configuração do cluster, o arquivo **install-config.yaml**, que irá carregar configurações gerais. No Código 3 é apresentado o arquivo de configuração para instalação do *cluster*.

Código 3 – Arquivo install-config.yaml para configuração OpenShift.

```
1  apiVersion: v1
2  baseDomain: vms-master-dal13.IBM.cloud
3  compute: # Worker nodes configuration
4  - hyperthreading: Enabled
5    name: worker
6    replicas: 0
7  controlPlane: # Master nodes configuration
8  - hyperthreading: Enabled
9    name: master
10   replicas: 3
11 metadata:
12   name: vanderlei-icpcs-cluster
13 networking: # OpenShiftSDN configuration
14   clusterNetwork:
15   - cidr: 10.128.0.100/14
16     hostPrefix: 11
17   networkType: OpenShiftSDN
18   serviceNetwork:
19   - 172.30.0.0/16
```

[4] O Sistema de Nomes de Domínio, mais conhecido pela nomenclatura em Inglês *Domain Name System*, é um sistema hierárquico e distribuído de gestão de nomes para computadores, serviços ou qualquer máquina conectada à Internet ou a uma rede privada.

```
20 platform:
21   none: {}
22 pullSecret: '{"auths": ************}'
23 sshKey: '************'
```

Na linha 6 do Código 3 o número de nós do tipo trabalhador (réplicas) é definido como nulo, pois iremos configurar manualmente cada nó trabalhador no nosso *cluster*. No caso de infraestrutura provisionada automaticamente através da nuvem, a própria ferramenta de instalação do *OpenShift* é capaz de provisionar as máquinas de maneira automática. Na linha 15, a opção **cidr** identifica um bloco de endereços IP que poderão ser alocados para as *pods* (do *Kubernetes*), no caso de necessidade de acesso externo às mesmas. Nas linhas 22 e 23 são declaradas as chaves SSH e um **pullSecret**, utilizado pela ferramenta de instalação para autenticação com os servidores que hospedam pacotes e dependências do *OpenShift* – que não é um serviço gratuito, e deve ser destravado por meio de aquisição de licença.

Após o provisionamento das máquinas virtuais e a configuração de DNS, SSH e detalhes do *cluster* nos arquivos correspondentes, é utilizada a ferramenta da instalação do *OpenShift* para gerar um arquivo de configuração para a ferramenta *Ignition*[5]. A ferramenta *Ignition* é utilizada para configurar as máquinas virtuais e instalar o sistema operacional *Red Hat Enterprise Linux CoreOS (RHCOS)*[6] em todos os nós do *cluster* de maneira remota e automática. Também é necessária a geração de manifestos de configuração do *Kubernetes*. Tudo isso pode ser feito por meio da ferramenta de instalação *openshift-install*, uma aplicação de linha de comando.

Figura 24 – Estrutura de arquivos gerada pela ferramenta openshift-install.

```
.
├── auth
│   ├── kubeadmin-password
│   └── kubeconfig
├── bootstrap.ign
├── master.ign
├── metadata.json
└── worker.ign
```

Fonte: Arquivo pessoal.

Com os arquivos de configurações gerados, é iniciado o comando para a criação

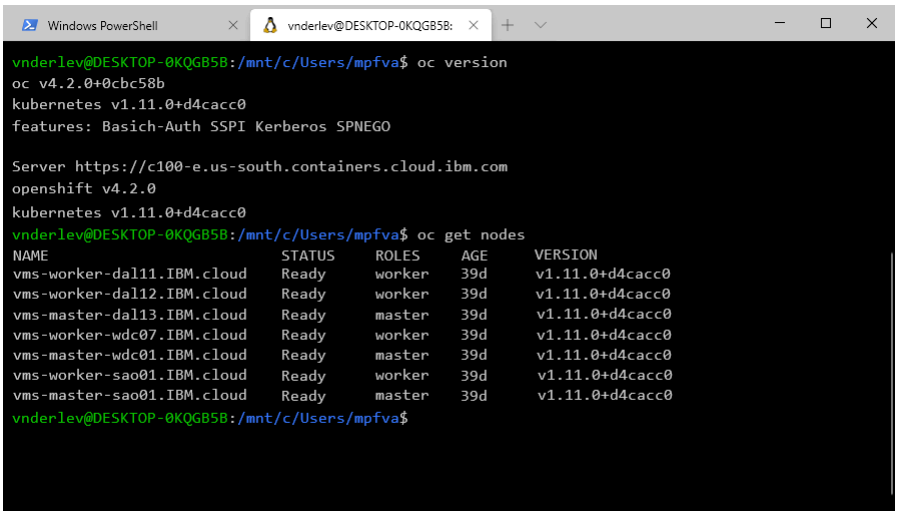
[5] Ignition é um novo utilitário de provisionamento projetado especificamente para gerenciamento de contêineres no *Red Hat Enterprise Linux CoreOS*. No nível mais básico, é uma ferramenta para manipular discos durante a inicialização antecipada.

[6] O *Red Hat Enterprise Linux CoreOS (RHCOS)* é um sistema operacional baseado no *Red Hat Enterprise Linux 8*, e é o único sistema com suporte à plataforma de contêineres *OpenShift* (após a versão 4.1).

do *cluster OpenShift* através da ferramenta *openshift-install*. Um processo de inicialização é realizado (e relativamente demorado), onde o *CoreOS* é primeiramente instalado em todas as máquinas, e posteriormente o *Kubernetes* e módulos do *OpenShift* são instalados e configurados automaticamente, com base das definições especificadas nos arquivos *Ignite* gerados.

Assim como no *Kubernetes*, para utilizar remotamente a interface de linha de comando do *OpenShift*, é necessário exportar para o sistema local o arquivo **KUBECONFIG** – gerado no processo de instalação do *cluster*. Na Figura 25 é mostrado o resultado de comandos simples para verificação dos nós, enviados ao *cluster* através da interface de linha de comando do *OpenShift* – ferramenta **oc**, que é um *wrapper* em torno da ferramenta **kubectl** do *Kubernetes*.

Figura 25 – Verificação de status do cluster OpenShift via CLI.



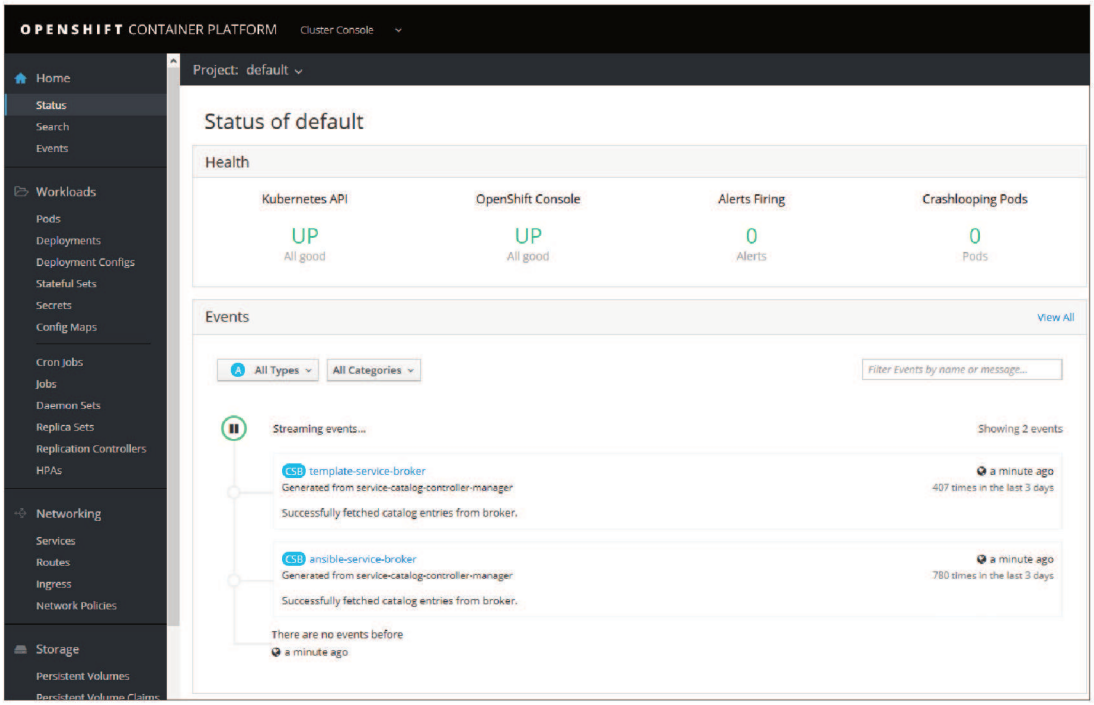
Fonte: Arquivo pessoal.

Além disso, também é possível utilizar o *Web* console do OpenShift – um dos grandes diferenciais da ferramenta, que permite a realização de alterações no número de nós do cluster e a criação de *deployments*, *pods*, *services*, e outros componentes do Kubernetes diretamente via interface gráfica, ou editor de arquivos de extensão **yaml** integrado. O *console Web* do *OpenShift* será utilizado pela equipe administrativa para monitorar a situação das aplicações *Web*, executadas em contêineres *stateless* facilmente replicáveis. Já a CLI administrativa, desenvolvida pelo autor deste trabalho, será utilizada para a realização de operações CRUD e monitoramento das camadas de dados – Kafka e Scylla – implantadas nas máquinas *bare-metal* provisionadas.

Na Figura 26 é apresentada uma captura de tela do *console Web* do OpenShift, mostrando o status de um projeto **default** (no *OpenShift*, um *project* é equivalente a um *namespace* no *Kubernetes*). Neste trabalho, todas as aplicações *Web* (seus contêineres) farão parte do projeto "*default*" para fins de simplicidade. Se desejado,

futuras equipes podem facilmente organizar cada uma das aplicações *Web* (*proxy Web* para acesso ao Kafka, *proxy Web* para acesso ao Scylla e *dashboard*) em projetos diferentes no *OpenShift*.

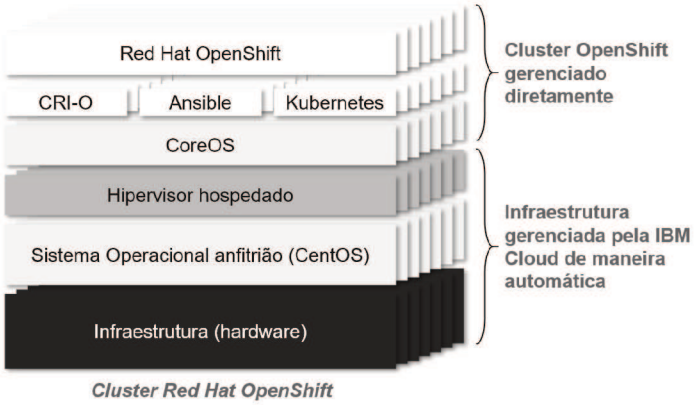
Figura 26 – Console *Web* do Red Hat OpenShift.



Fonte: Arquivo pessoal.

Na Figura 27 podemos conferir um esquemático de pilhas das sete máquinas virtuais e dependências instaladas, compondo o *cluster OpenShift* configurado.

Figura 27 – Camadas de *hardware* e *software* do *cluster* OpenShift.



Fonte: Arquivo pessoal.

O *cluster* OpenShift provisionado possui nós em três regiões geográficas, em

centros de dados diferentes da IBM. Do ponto de vista deste trabalho, todo o processo de instalação da plataforma *OpenShift* pode ser automatizado por meio de *shell scripts*. Um *script* foi preparado pelo autor, entretanto, no caso de um real evento, será certamente utilizada a oferta PaaS do *OpenShift* através da IBM Cloud, que é automaticamente configurada e pronta para utilização. O uso da oferta PaaS dispensa a necessidade de provisionamento de máquinas virtuais – tudo isso é feito automaticamente pela IBM Cloud.

5.2 INSTALAÇÃO E CONFIGURAÇÃO DO KAFKA

O barramento de mensagens distribuído, segundo a especificação do sistema desenvolvida na etapa de planejamento, será baseado no *Apache Kafka*. Como mostrado na Seção 2.10, será necessário instalar o *Zookeeper* para sincronização e manutenção dos nós no *cluster* Kafka. Tanto o Kafka como o *Zookeeper* podem ser instalados via contêineres facilmente no *cluster OpenShift* configurado. Entretanto, como a arquitetura do *Kubernetes*, e do próprio *OpenShift*, é voltada para aplicações *stateless*, a configuração de persistência de dados pode tornar-se demasiadamente complicada. Então, optou-se pela instalação padrão do *Apache Kafka* e do *Zookeeper* em múltiplas máquinas, diretamente sobre o sistema operacional.

O primeiro passo para instalação do Kafka – considerando que já foram provisionados três servidores *bare-metal* – é a instalação do *Java Development Kit 1.8 (JDK)*. Isso é feito facilmente via comando SSH para cada uma das máquinas, acionando o processo de instalação do JDK via ferramenta Yum[7]. Com o JDK 1.8 instalado, passa-se para a instalação do *Kafka 2.4.0* (última versão estável). Com o uso da ferramenta *Wget*[8], acionada via comandos SSH, arquivos binários oficiais do projeto Kafka, já com binários do *Zookeeper* inclusos, são baixados nas três máquinas.

Para que o barramento de mensagens não pare de funcionar no caso de queda de conexão ou falha em uma das três máquinas, o *Zookeeper* deve ser instalado como um *cluster* também – no jargão da ferramenta, será criado um *Zookeeper Ensemble*. Para configuração do *Zookeeper*, é editado o arquivo **zookeeper.properties**, carregado juntamente com o Kafka via ferramenta *Wget*. Neste arquivo, desejamos alterar os seguintes valores, apresentados no Quadro 10. Após o ajuste dos parâmetros desejados, para a inicializar o *Zookeeper* basta executar o shell script **zookeeper-server-start.sh** com o arquivo **zookeeper.properties** como argumento. Nas três máquinas *bare-metal* o *script* de inicialização do *Zookeeper* é preparado para ser executado automaticamente na inicialização.

[7] O *Yellowdog Updater, Modified* ou conhecido também como Yum é uma ferramenta utilizada para gerenciar a instalação e remoção de pacotes em distribuições *Linux*, que utilizam o sistema RPM de gerenciamento de pacotes.

[8] *Wget* é um programa de código-aberto que propicia o download de dados da *Web*. Seu nome deriva de *World Wide Web* e a palavra *get*. Ele suporta os protocolos a HTTP, HTTPS e FTP.

Quadro 10 – Configurações dos nós do cluster Zookeeper.

Configuração	**Valor**		
/var/zookeeper/myid	1	2	3
dataDir	*/home/zookeeper*		
maxClientCnxns	100		
clientPort	2181		
server.1	baremetal-dal11.IBM.cloud:2888:3888		
server.2	baremetal-sao01.IBM.cloud:2888:3888		
server.3	baremetal-fra02.IBM.cloud:2888:3888		

Fonte: Arquivo pessoal.

Com o *cluster* Zookeeper configurado, passamos para a configuração do *cluster* Kafka. O arquivo **server.properties** deve ser editado em cada uma das três máquinas *bare-metal*, a fim de se configurar as opções apresentadas no Quadro 11. Basicamente, o diretório onde os *logs* de cada nó Kafka são salvos são alterados de */tmp/kafka-logs* para */home/kafka-logs*; cada nó recebe um *broker.id* diferente e um *broker.rack* (parâmetro utilizado pelo mecanismo de tolerância à falhas do *Zookeeper*).

Quadro 11 – Configurações dos nós do *cluster* Kafka.

Configuração	**Valor**		
broker.id	0	1	2
broker.rack	RACK1	RACK1	RACK2
log.dirs	*/home/kafka-logs*		
offsets.topic.num.partitions	3		
offsets.topic.replication.factor	2		
min.insync.replicas	2		
default.replication.factor	2		
zookeeper.connect	baremetal-dal11.IBM.cloud:2181,baremetal-sao01.IBM.cloud:2181,baremetal-fra02.IBM.cloud:2181		

Fonte: Arquivo pessoal.

Os parâmetros de configuração definidos para o Kafka e *Zookeeper* são basicamente parâmetros para a implementação inicial do sistema, e eles são geralmente ajustados após o estudo das condições reais de operação. Para um *cluster* de três nós Kafka, é comum a divisão dos dados em três partições, com um fator de replicação duplo.

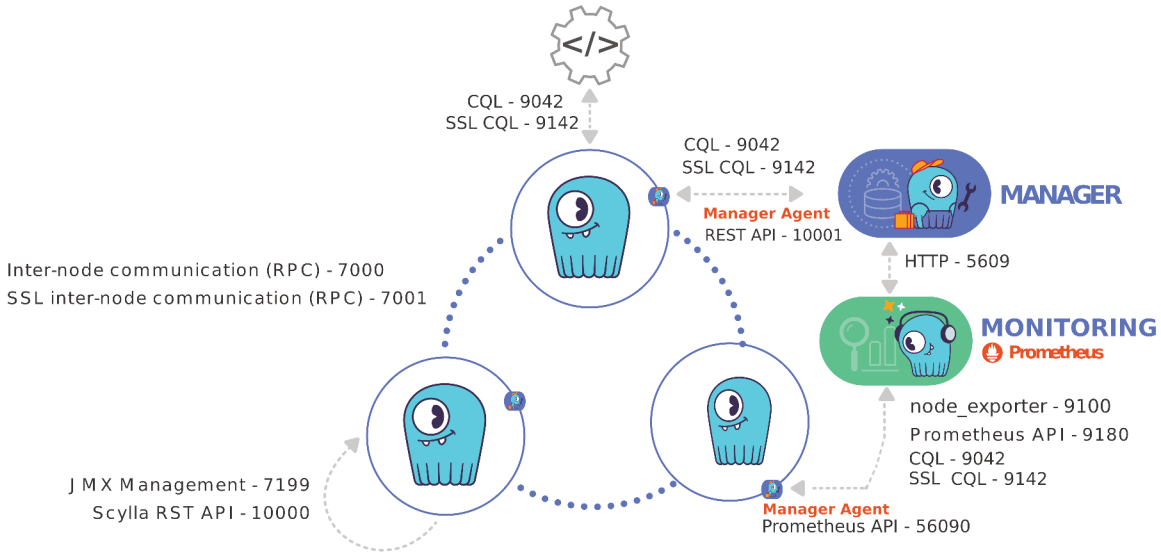
Para iniciar um *broker*, ou nó do *cluster* Kafka, basta executar o *shell script* **kafka-server-start.sh** (parte do código-fonte do *Apache Kafka*) passando como argumento o arquivo *server.properties*. O mesmo procedimento é realizado para que o *broker* Kafka seja inicializado automaticamente após o *boot* da máquina e a inicialização do *Zookeeper* (via ferramenta **systemctl** do *CentOS*).

5.3 INSTALAÇÃO E CONFIGURAÇÃO DO SCYLLA

Para a instalação de um *cluster* de banco de dados distribuído Scylla, o procedimento é semelhante ao executado para a instalação do *cluster OpenShift* e do *cluster* Kafka. Com as três máquinas *bare-metal* provisionadas, as dependências do Scylla são primeiramente instaladas. É necessária a instalação da ferramenta *Yum* (já instalada e utilizada também na instalação do Kafka), e a remoção do pacote Abrt[9] do *CentOS* (pois existe um conflito entre essa ferramenta e o mecanismo de *logging* nativo do Scylla). O pacote Abrt pode ser removido com a ferramenta Yum através do comando **sudo yum remove -y abrt**. Também é necessário a adição do repositório de pacotes EPEL[10] no *CentOS*, que pode ser feita por meio do comando **sudo yum install epel-release**.

Em relação às configurações de rede, o Scylla requer que as seguintes portas TCP sejam expostas: 9042, 9142, 7000, 7001, 7199, 10000, 9180, 9100, 9160, 10001, 5609, e 56090. Na Figura 28, disponível na documentação oficial do projeto Scylla, é apresentado um esquemático dos tipos de comunicação realizados através de cada porta.

Figura 28 – Diagrama de portas expostas para o banco de dados Scylla.



Fonte: `https://tinyurl.com/v3tjs4s` (acessado em: 24/01/2020).

A versão do Scylla a ser instalada é a 3.2.0 estável de código-aberto (não será utilizada a versão *Enterprise*). O código-fonte do Scylla é disponibilizado na pá-

[9] O Abrt é um conjunto de pequenos utilitários do sistema operacional CentOS desenvolvido para simplificação da geração de relatórios de problemas de *software* e *logging*.

[10] Pacotes Adicionais para *Enterprise Linux* (ou EPEL), é um grupo especial do do projeto *Fedora* com interesse em criar, manter e gerenciar um conjunto de pacotes de *software* adicionais com alta qualidade para o *Enterprise Linux*, incluindo, mas não se limitando a, *Red Hat Enterprise Linux (RHEL)*, *CentOS* e *Scientific Linux (SL)*.

gina oficial do projeto, no endereço eletrônico `https://www.scylladb.com/download/open-source/` (acessado em: 24/01/2020). Uma vez carregados os arquivos do Scylla em cada um dos três servidores *bare-metal*, o arquivo /**etc**/**scylla**/**scylla.yaml** deve ser editado, a fim de principalmente configurar os endereços IP para comunicação via RPC[11], endereços IP para comunicação entre os nós, e o nome unificado do *cluster*.

No Quadro 12 são apresentados os parâmetros do arquivo **scylla.yaml** que serão modificados para a implantação do *cluster* deste trabalho.

Quadro 12 – Parâmetros configurados no arquivo scylla.yaml.

Parâmetro	**Valor**
cluster_name	icpcs-scylla
listen_address	baremetal-dal11.IBM.cloud, baremetal-sao01.IBM.cloud, baremetal-fra02.IBM.cloud
seeds	baremetal-dal11.IBM.cloud, baremetal-sao01.IBM.cloud
auto_bootstrap	true
endpoint_snitch	GossipingPropertyFileSnitch
rpc_address	baremetal-dal11.IBM.cloud, baremetal-sao01.IBM.cloud, baremetal-fra02.IBM.cloud
data_file_directories	/var/lib/scylla/data

Fonte: Arquivo pessoal.

O parâmetro **endpoint_snitch** determina o tipo de *snitch* que será utilizado pelo *cluster* Scylla. O *snitch* é um mecanismo de *software* que averigua a topologia de rede do *cluster* e determina para quais nós serão distribuídas as réplicas de dados. O *snitch* do tipo **GossipingPropertyFileSnitch** permite a definição manual de como serão distribuídas as réplicas. Esse *snitch* transmite a política de replicação de dados para os outros nós baseado nas informações escritas no arquivo /**etc**/**scylla**/**cassandra-rackdc.properties**.

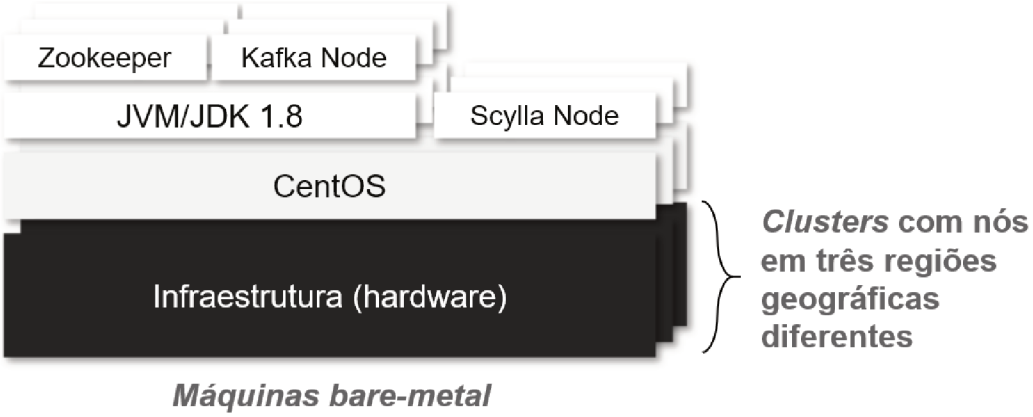
Após a configuração do arquivo **scylla.yaml**, podemos executar o *shell script* **scylla_setup.sh** para instalar e configurar cada nó do *cluster* automaticamente (o procedimento deve ser repetido nas três máquinas). Após a instalação, igualmente ao que foi realizado para o Kafka, o Scylla é configurado para inicializar junto com a inicialização do *CentOS* com o uso da ferramenta de sistema **systemctl**.

Após a instalação do *Zookeeper*, do Kafka e do Scylla, a "*stack*" do sistema implantado nos servidores *bare-metal* fica como o apresentado na Figura 29.

Com o ambiente de operação configurado, a implementação em código é iniciada, com a realização de testes diretamente nas máquinas provisionadas para produção – seguindo o estabelecido no método de desenvolvimento apresentado na Seção 4.4.

[11] *Remote Procedure Call (RPC)* é um protocolo que um *software* pode utilizar para solicitar um serviço de um *software* localizado em outro computador através de uma rede, sem precisar entender os detalhes da rede.

Figura 29 – Camadas de dados implantadas nos servidores bare-metal.



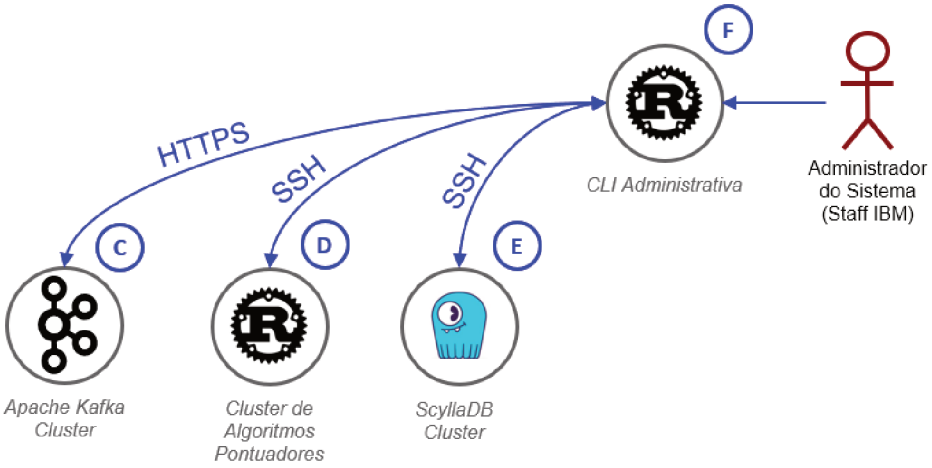
Fonte: Arquivo pessoal.

5.4 DESENVOLVIMENTO DA INTERFACE ADMINISTRATIVA

A interface administrativa será uma aplicação de linha de comando focada na centralização da administração dos componentes da camada de dados do sistema. Essa aplicação irá integrar-se com as aplicações executadas nos servidores *bare-metal* – principalmente os *clusters* Kafka e Scylla.

Na Figura 30 é apresentada parte do esquemático geral do sistema, destacando a interface administrativa e os componentes que serão integrados.

Figura 30 – Integrações entre a interface administrativa e o resto do sistema.



Fonte: Arquivo pessoal.

No que refere-se ao *cluster* Kafka, a CLI administrativa possui métodos para realizar chamadas diretamente à API administrativa do Kafka, capaz de criar tópicos e ler *logs* do *cluster*. Essa API pode ser acessada via chamada HTTPS, e para a

execução de requisições desse protocolo é usada a biblioteca **hyper** da linguagem *Rust*, possibilitando a criação e exclusão de tópicos, além do carregamento de *logs* do Kafka.

No que refere-se ao *cluster* Scylla, foi implementado na CLI administrativa métodos para chamar a ferramenta *CQLSh*. O *CQLSh* é uma aplicação de linha de comando para interagir com os bancos de dados *Cassandra* e Scylla por meio de consultas com a linguagem CQL. O executável do *CQLSh* é disponibilizado juntamente com o Scylla no diretório /bin, e utiliza o *driver* de protocolo nativo do *Python* para conectar-se aos nós do *cluster* Scylla. Através da ferramenta *CQLSh* é possível realizar qualquer operação diretamente nas tabelas do Scylla.

Recapitulando, as chamadas HTTPS para integração com o *cluster* Kafka são construídas na aplicação com o auxílio da biblioteca **hyper**, enquanto que para o acionamento da ferramenta *CQLSh* são realizados comandos SSH diretamente nos servidores *bare-metal* com os parâmetros desejados. Os comandos SSH são construídos e enviados pela aplicação com o auxílio da biblioteca **ssh2** da linguagem *Rust*.

Quanto à implementação da interface, ela é construída com o uso da biblioteca **clap** da linguagem *Rust*. Essa biblioteca possui métodos para a construção rápida de aplicações de linha de comando, com anotadores para a definição de parâmetros obrigatórios e definição de descrições, ou *helpers*, para cada um deles.

Uma vez construída a interface administrativa, o teste de funcionamento é realizado por meio da criação do tópico no Kafka e das tabelas no Scylla necessárias para o armazenamento das informações de uma competição de programação hipotética.

5.5 MODELAGEM DE ESTRUTURAS DE DADOS

Antes de partir para o desenvolvimento dos próximos incrementos de *software*, devem ser estabelecidos os tópicos do Kafka e tabelas do banco de dados Scylla.

No Kafka, um tópico é criado com 48 partições e fator de replicação igual a três. Todas as outras configurações possíveis são as padrões do Kafka. Uma partição no Kafka é como uma unidade de paralelismo – quando mais partições, mais consumidores paralelos podem ser adicionados, entretanto uma latência considerável é adicionada ao servidor Kafka conforme o número de partições dos tópicos (e também o fator de replicação) é aumentado. A quantidade de 48 partições foi estabelecida com base no número total de núcleos de processamento (lógicos) do *cluster* de servidores *bare-metal* provisionado. Em uma eventual competição, um tópico Kafka para cada desafio será necessário. como neste trabalho estamos considerando apenas um caso de estudo, um único tópico foi criado.

No banco de dados Scylla são criadas duas tabelas: uma para armazenar informações sobre os participantes, e outra para armazenar informações sobre as soluções submetidas (todos os tipos de desafios em uma única tabela). A tabela de participantes

é baseada na estrutura apresentada no Código 4.

Código 4 – CQL para criação da tabela de participantes no Scylla.

```
CREATE TABLE IF NOT EXISTS keyspace.participants (
    participant_id uuid,
    ibmer boolean,
    email text,
    full_name text,
    cpf_number int,
    birthday bigint,
    state text,
    city text,
    requested_at bigint,
    subscribed_at bigint,
    PRIMARY KEY participant_id
);
```

No Código 5 é apresentada a *string* em linguagem CQL para a criação da tabela de soluções dos desafios.

Código 5 – CQL para criação da tabela de soluções no Scylla.

```
CREATE TABLE IF NOT EXISTS keyspace.scoredsolutions (
    participant_id uuid,
    challenge_id smallint,
    report text,
    final_score int,
    submitted_at bigint,
    scored_at bigint,
    PRIMARY KEY ((participant_id, challenge_id), submitted_at)
);
```

A definição de múltiplas chaves primárias serve para a separação das tabelas em "partições" do Scylla. Por exemplo, no caso da tabela criada com o CQL apresentado no Código 5, as entradas na tabela serão automaticamente inseridas de acordo com o *Uuid*[12] do participante (*participant_id*) e seguido de um número de identificação do desafio (*challenge_id*) – no Scylla só serão aceitas consultas em CQL que declarem ambas chaves primárias. O tempo em que a solução foi submetida (*submitted_at*, com precisão de milissegundos) é acertado para ser a coluna de ordenação das entradas.

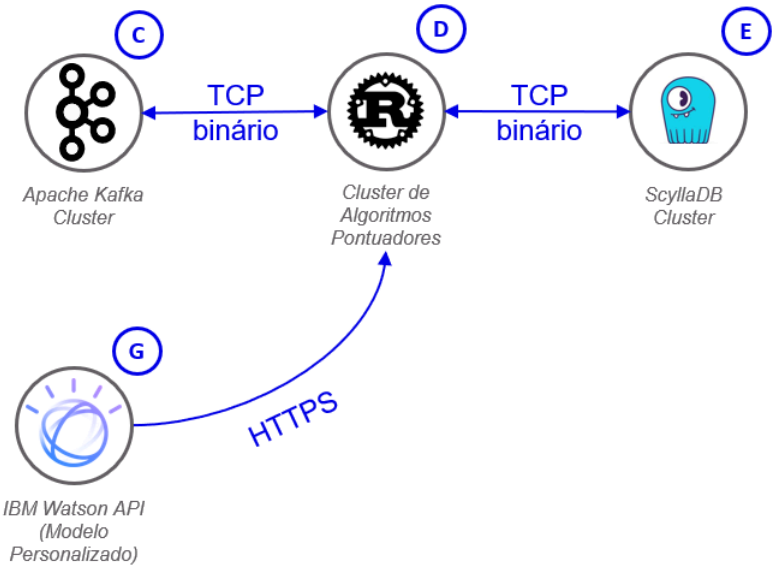
[12] Um Identificador Único Universal, do inglês *Universally Unique Identifier (UUID)*, é um número de 128 *bits* usado para identificar informações em sistemas de computação.

5.6 DESENVOLVIMENTO DOS PONTUADORES AUTOMÁTICOS

O pontuador automático é na essência uma aplicação de linha de comando compilada em um binário executável, capaz de ser acionado por meio de requisições SSH diretamente nas máquinas *bare-metal* provisionadas.

Quanto às integrações desse módulo com o restante do sistema, essa aplicação irá comunicar-se diretamente com o banco de dados distribuído Scylla, o barramento de mensagens Kafka, e as APIs do *Watson Knowledge Studio* que hospedarão os modelos de anotação textual criados pelos participantes da competição. Na Figura 31 é apresentada parte do esquemático geral do sistema, destacando a aplicação de processamento de soluções (pontuação automática) e os componentes que serão integrados.

Figura 31 – Integrações relacionadas aos pontuadores automáticos.



Fonte: Arquivo pessoal.

Seguindo o método estabelecido na Seção 4.4, os pontuadores automáticos desenvolveram-se a partir dos seguintes incrementos de *software*:

1. Desenvolvimento da estrutura *"main"* da aplicação, incluindo a interface de linha de comando e ambiente de execução (*loop* de eventos) para execução assíncrona das diversas tarefas necessárias para a pontuação de uma submissão;

2. Desenvolvimento de um mecanismo para chamar métodos da biblioteca **librd-kafka**[13] da linguagem *C* na linguagem *Rust*. Essa biblioteca é uma poderosa ferramenta largamente testada na criação de aplicações clientes para o Kafka;

[13] O código-fonte da biblioteca é disponibilizado em `https://github.com/edenhill/librdkafka` (acessado em: 27/01/2020).

3. Integração e teste do cliente Kafka no *loop* de eventos da aplicação principal;
4. Desenvolvimento de um módulo em *Rust* para comunicação com o banco de dados distribuído Scylla;
5. Integração e teste do cliente Scylla no *loop* de eventos da aplicação principal;
6. Desenvolvimento de uma biblioteca *Rust* para comunicação com a API do *Watson Knowledge Studio*;
7. Integração e teste do cliente Watson no *loop* de eventos da aplicação principal;
8. Desenvolvimento do algoritmo de pontuação; e
9. Desenvolvimento de extras, como opções de *logging* e melhorias na interface.

5.6.1 Estrutura principal e loop de eventos da aplicação

O incremento 1, referente à estrutura geral da aplicação, foi construído com o uso da biblioteca **clap** da linguagem *Rust* – a mesma utilizada para a construção da CLI administrativa. São definidos dois comandos para a aplicação: o comando *start*, para iniciar o consumo de mensagens no Kafka e todo o processo de pontuação; e o comando *stop*, para finalizar todas as sessões abertas (Kafka, Scylla, e Watson) e desligar a aplicação. A aplicação e os comandos são executados pelo sistema operacional *CentOS*, que é capaz de receber instruções via protocolo SSH da CLI administrativa, possibilitando o controle de funcionamento desse componente pelo usuário definido como administrador do sistema.

Além da interface de linha de comando, é utilizada a biblioteca **tokio** para implementação de um executor de métodos assíncronos – a linguagem Rust, diferentemente de linguagens como *Go*, *Javascript* e *C#*, não possui um ambiente de programação assíncrona integrado, e o executor deve ser completamente implementado (em nível de biblioteca). A biblioteca **tokio** fornece a implementação de um executor *multithreaded* com agendador de tarefas baseado no paradigma “roubo de trabalho”, ou *work-stealing*. No paradigma *work-stealing*, cada núcleo processador de um computador possui uma fila de tarefas, e cada tarefa é formada por instruções a serem executadas sequencialmente. No decorrer da execução das tarefas, uma tarefa pode gerar novas tarefas que muitas vezes podem ser executadas em paralelo com outras. Essas novas tarefas geradas são alocadas ao fim da fila do núcleo processador na qual elas foram originadas. Adotando uma filosofia *work-stealing*, quando um núcleo fica sem nenhuma tarefa em sua fila ele analisa as filas dos outros núcleos e “rouba” as tarefas em aguardo. Dessa forma, todo o trabalho é distribuído entre núcleos processadores inativos evitando a sobrecarga de agendamento.

A biblioteca **tokio** é construída sobre funções assíncronas (declaradas com a notação *fn async* na linguagem *Rust*), o operador *.await*, e uma estrutura de dados chamada *future*. Quando uma função assíncrona é chamada na linguagem *Rust* nenhum efeito imediato ocorre – em vez disso, a função retorna uma estrutura *future*, que representa o resultado de uma computação suspensa em aguardo para ser executada. Para executar um *future*, é necessário a aplicação do operador *.await*. Após a construção de uma aplicação de linha de comando mínima com as funções *start* e *stop*, e um executor integrado baseado na biblioteca **tokio**, o desenvolvimento passa para o incremento 2.

5.6.2 Integração com o Apache Kafka

O incremento 2 é focado na integração do pontuador automático com o Kafka. Esse módulo deve ser capaz de consumir de maneira assíncrona as mensagens armazenadas em um dos tópicos criados no barramento de mensagens. Para tal, são utilizados dois mecanismos da linguagem *Rust*: o *foreign function interface*, para chamar funções da linguagem *C*; e o *static linking*, para que o compilador gere um binário executável com todas as dependências integradas, dispensando a necessidade de instalação de bibliotecas *C* nos sistemas onde o programa será executado. O Kafka não possui um cliente oficial escrito em linguagem *Rust* (apenas disponível em *Java*), entretanto, nos últimos anos a comunidade escreveu um cliente em linguagem *C* de código-aberto que já foi amplamente testado e possui desempenho notável: a biblioteca **librdkafka**. Como ainda não existe um cliente Kafka escrito puramente em *Rust*, e somente a escrita de tal cliente poderia ser um outro trabalho de fim de curso, é tomado proveito das capacidades de integração do *Rust* com a biblioteca consolidada na linguagem *C*. Existem projetos visando a criação de um cliente Kafka puramente em *Rust*, entretanto ainda são projetos imaturos. A biblioteca **librdkafka** possui métodos para a criação de sessões e consumo de mensagens do tópico Kafka através do protocolo TCP binário. Então, assim que o pontuador é acionado com o comando *start*, uma sessão de consumo no tópico Kafka é criada. Essa mesma sessão é finalizada quando o comando *stop* é acionado.

As mensagens que chegam através de uma *stream*, ou fluxo de dados, vêm codificadas em base 64 (padrão do Kafka). Elas são decodificadas por uma função de decodificação da biblioteca **base64** do *Rust*, e uma estrutura de dados imutável do tipo apresentado no Código 6 é instanciada e transportada para o restante do algoritmo para processamento (pontuação).

Código 6 – Estrutura de dados de uma mensagem do Kafka.

```
1 struct KafkaMessage {
2     nlu_api_key: String, // chave de API do modelo anotador (Watson)
```

```
3     nlu_model_id: String, // identificador do modelo anotador (Watson)
4     cpf: String // cpf de cadastro do participante
5     participant_id: uuid::Uuid, // uuid identificador do participante
6 }
```

Munido de um mecanismo para comunicação com o Kafka, o processo de desenvolvimento passa para o incremento 3. A integração do Kafka com o *loop* de eventos da aplicação é implementada da seguinte forma: uma sessão do tipo *stream*, ou fluxo, é realizada com o *cluster* durante a inicialização do programa. Essa sessão é do tipo consumidora, e é construída por meio de uma função da biblioteca **librdkafka**. A aplicação irá continuamente verificar por novas mensagens, filtrando eventuais erros de comunicação. Quando uma eventual mensagem é capturada, ela é processada (decodificada) e transportada para uma pilha de tarefas do *loop* de eventos criado com a biblioteca **tokio**, que irá gerenciar as *threads* e o processamento da mensagem de maneira assíncrona.

Quanto à segurança de comunicação, a sessão com o Kafka é construída por meio do protocolo TCP criptografado (padrão TLS), e a lógica é implementada com a biblioteca **openssl** da linguagem *C* (compilada estaticamente com a aplicação conforme o processo descrito anteriormente). Para que o estabelecimento da sessão ocorra com sucesso, é necessário indicar o diretório onde o arquivo **pem**[14] com os certificados SSL/TLS do sistema estão localizados. Nas máquinas *CentOS*, esse arquivo **pem** é localizado por padrão no diretório /**etc**/**ssl**/**certs**/. Para fins de maior automatização da aplicação desenvolvida, é utilizada a biblioteca **openssl-probe** da linguagem *Rust* para buscar automaticamente o diretório dos certificados.

5.6.3 Integração com o banco de dados Scylla

O incremento 4 refere-se à comunicação com o banco de dados Scylla com o uso da linguagem *Rust*. Diferentemente do Kafka, o banco de dados Scylla possui um *driver* escrito puramente em *Rust*, dispensando o uso de bibliotecas da linguagem *C*. A biblioteca utilizada na aplicação é a **cdrs**, um *driver* escrito puramente em *Rust* para o banco de dados *Cassandra* – felizmente o Scylla é completamente compatível com a interface do *Cassandra*, e essa biblioteca pode ser utilizada sem nenhuma complicação.

A comunicação da aplicação com o banco de dados é realizada por meio de sessões através do protocolo TCP, também com segurança nível TLS, e o mesmo mecanismo de busca de certificados aplicado para o Kafka é aplicado para o Scylla. Quanto à integração da comunicação com o banco de dados no *loop* de eventos da

[14] *Privacy-Enhanced Mail*, ou “correio com privacidade aprimorada”, é um formato de arquivo para armazenamento de chaves criptográficas, certificados e outros tipos de dados sensíveis, com base em um conjunto de padrões IETF de 1993.

aplicação, é criado um grupo de sessões com o Scylla na inicialização da aplicação – evitando a criação e o fechamento desnecessário de sessões. As sessões criadas são compartilhada pelas *threads* da aplicação – o gerenciamento das sessões é realizado em nível de biblioteca.

Na aplicação são implementadas funções para a escrita de novos participantes e novas soluções submetidas (dois únicos tipos de estruturas de dados que serão armazenados). Não são escritas funções para deleção ou atualização de dados, que não serão necessárias. Qualquer tipo de modificação de atributo – por exemplo, no caso de um participante que informou seus dados de maneira equivocada – será feita manualmente pela *staff* dos eventos futuros utilizando a aplicação de linha de comando administrativa desenvolvida.

No Código 7 é apresentada a função para a escrita de um novo participante na competição.

Código 7 – Função codificada rigidamente para escrita no Scylla.

```
/// Function for inserting new User into Scylla Partition
pub fn insert_participant(_session: &CurrentSession, _row:
    ParticipantDataStruct) {
     let _cqlstring: &'static str = r#"
         INSERT INTO challengeks.participants (
             participant_id,
             ibmer,
             email,
             full_name,
             cpf_number,
             birthday,
             state,
             city,
             requested_at,
             subscribed_at
         ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?);"#;
     _session.query_with_values(
         _cqlstring,
         _row.into_query_values()
     ).expect(" insert_participants error ");
}
```

A sintaxe da linguagem CQL e a codificação rígida da *string* da consulta formam um forte mecanismo de proteção contra a maioria dos ataques do tipo *injection*. Nessa função, os dados informados pelo usuário são pré-processados e inseridos em uma estrutura padrão que é passada como entrada (*ParticipantDataStruct*, apresentada no Código 8), impossibilitando o roubo de dados por meio da inserção de consultas

maliciosas.

Código 8 – Estrutura com dados dos participantes.

```
struct ParticipantDataStruct {
    participant_id: uuid::Uuid, // unique identifier
    ibmer: bool, // is the participant an IBM employee?
    email: String, // participant's email
    full_name: String, // participant's full name
    cpf_number: i32, // cpf
    birthday: i64, // day of birth
    state: String, // state where participant is localized
    city: String, // city where participant is localized
    requested_at: i64, // epoch timestamp when subscribed
    subscribed_at: i64 // epoch timestamp when confirmed subscription
}
```

Dessa forma, mesmo que de alguma maneira um invasor consiga burlar os mecanismos de segurança na produção de uma mensagem para o Kafka ou consulta no Scylla (as únicas interfaces do sistema expostas ao público, protegidas pelos *proxies*), a aplicação de pontuação é incapaz de escrever ou sobrescrever dados no banco, e uma falha é retornada imediatamente no caso de inconsistência na estrutura definida no Código 8, impossibilitando a inserção de consultas diferentes do esperado no sistema – além de tudo, o usuário não recebe nenhum *feedback* específico de erro, dificultando qualquer tipo de *fingerprinting*[15] do sistema.

5.6.4 Integração com o IBM Watson

Os incrementos de *software* 6 e 7 definidos anteriormente refem-se à integração da aplicação desenvolvida em *Rust* com as APIs dos modelos criados com o *Watson Knowledge Studio*. A API exposta trabalha sobre o protocolo HTTP (com suporte à autenticação via TLS), e um cliente pode realizar uma chamada para uma *URL endpoint* (que é diferente, dependendo da região onde o modelo foi exposto), informando uma *api-key* e uma *string* de texto como entrada. Outros parâmetros de configuração podem ser determinados, como opções para extração de aspectos sentimentais do texto de entrada.

O Watson funciona com um esquema de sessões – o programa precisa se autenticar uma única vez, e posteriormente pode realizar quantas chamadas desejar. Entretanto, após um período de 5 minutos de ociosidade, é necessária a criação de uma nova sessão, ou re-autenticação. No caso dos modelos de linguagem natural, a

[15] *Fingerprinting* no contexto de segurança da informação refere-se à correlação de metainformações de um sistema computacional a fim de identificar-se serviços de rede, sistemas operacionais, bancos de dados, e outros tipos de *softwares* subjacentes.

API é completamente *stateless*, e o algoritmo não precisará guardar informação entre uma chamada de API e outra (diferente das APIs do Watson para assistentes virtuais, ou *chatbots*).

Para a execução das requisições para o Watson é possível usar a bibliteca **curl** da linguagem *C* ou as opções escritas puramente em *Rust*, como a própria biblioteca **hyper** voltada para comunicação HTTP. Neste trabalho foi escolhida a biblioteca **hyper**, e duas funções assíncronas foram escritas para a realização de requisições: uma para a autenticação e criação de uma sessão com o Watson, e outra para a realização de uma anotação textual com um modelo personalizado.

Ambas funções possuem como argumento a *api-key* da instância do Watson criada pelo participante, e a função de anotação possui também como argumento o *model-id* do modelo de anotação customizado com o *Watson Knowledge Studio* e uma *string* com o texto a ser anotado. As *endpoints* das APIs do Watson são padronizadas e codificadas rigidamente nas funções, em vez de serem passadas como argumento – para serviços do Watson criados na região **us-south** (Dallas), a *endpoint* é `https://gateway.watsonplatform.net/natural-language-understanding/api`.

No Código 9 é apresentado um exemplo de requisição à API do Watson (no formato *curl*).

Código 9 – Requisição curl à API do Watson.

```
curl -X POST \
-H "Content-Type: application/json" \
-u "apikey:{apikey}" \
-d @parameters.json \
"{url}/v1/analyze?version=2019-07-12"
```

No Código 9, os parâmetros **{apikey}** e **{url}** substituem respectivamente a chave de API do usuário e o *endpoint* da API do Watson. O argumento **parameters.json** é um arquivo com os parâmetros da requisição, com um exemplo apresentado no Código 10.

Código 10 – Parâmetros de uma requisição curl à API do Watson.

```
{
  "text": {input-string},
  "features": {
    "entities": {
      "model": {model-id},
      "sentiment": false, // disabled by default
      "emotion": false, // disabled by default
      "limit": 30 // maximum entities to be marked
    }
```

```
  }
}
```

No Código 10 o parâmetro **{model-id}** é um identificador de modelo anotador hospedado no Watson (o mesmo valor informado pelo participante no ato da submissão de sua solução), e o parâmetro **{input-string}** é o excerto de texto que será analisado pelo modelo anotador.

No Código 11 é apresentado um exemplo simples de língua inglesa com a estrutura JSON retornada pela API do Watson, descrevendo duas entidades anotadas.

Código 11 – Estrutura da resposta retornada pelo Watson.

```
{
    "entities": [
        {
            "type": "Employee",
            "text": "Vanderlei",
            "confidence": 0.988112,
            "location": [0, 8],
        },
        {
            "type": "Company",
            "text": "IBM",
            "confidence": 1.0,
            "location": [121, 123],
        }
    ]
}
```

Nota-se que o algoritmo de pontuação deverá ser bem trabalhado para que seja correto (pontue corretamente), pois devem ser identificadas apenas as entidades corretas, e nenhuma “a mais”. Além disso, não somente a existência das entidades deve ser verificada, mas também o conteúdo textual anotado e a sua posição correta nas frases de entrada.

5.6.5 Desenvolvimento do algoritmo de pontuação

Os últimos incrementos de *software* planejados para os pontuadores automáticos estão relacionados à implementação de fato do algoritmo de pontuação e sua integração com todas as APIs dos diversos outros componentes do sistema. Fazendo uso do paradigma de programação assíncrona da linguagem *Rust*, baseada em funções que retornam valores “futuros”, são realizadas dez chamadas HTTPS para cada modelo criado de maneira simultânea (em contraste com a abordagem síncrona e blo-

queante adotada no sistema de avaliação piloto). As saídas dos modelos anotadores do Watson serão então passadas como entrada para a função de avaliação.

A função de avaliação é também uma função assíncrona, que irá receber como entrada uma *string* com o JSON de saída da API do Watson, e construir um vetor com *structs* em *Rust* representando entidades de texto. Essa *struct* é apresentada no Código 12.

Código 12 – Estrutura de dados para entidades textuais.

```
1 struct WatsonTextEntity {
2     name: String, // "type" is a reserved word in Rust.
3     text: String,
4     confidence: f32
5 }
```

No Código 12 a variável **name** indica o nome dado pelo participante à entidade de texto. A variável **text** contém os caracteres identificados como uma entidade de texto, e a variável **confidence** é um valor contido entre 0 e 1 indicando a confiança do modelo na anotação realizada – esse valor é utilizado como peso multiplicador nas pontuações parciais de cada entidade detectada.

Após a criação de um vetor de tamanho fixo (30), ele é populado com uma estrutura do tipo **WatsonTextEntity** para cada entidade retornada pela API do Watson (se menos de 30 entidades forem detectadas, os outros elementos do vetor são preenchidos com estruturas vazias). Cada chamada usa como entrada um excerto de texto com temática pré-estabelecida, e os participantes possuem conhecimento prévio dos tipos de entidades que eles devem anotar. Dessa forma, o algoritmo pontuador irá percorrer o vetor de entidades e checar se os nomes das entidades (**WatsonTextEntity.name**) correspondem a uma entidade relevante ao desafio, e se o texto (**WatsonTextEntity.text**) é de fato uma entidade do tipo anotado. Essa última verificação é traiçoeira, pois o usuário pode anotar espaços ou outros caracteres além da palavra desejada, até mesmo mais de uma palavra, e mesmo assim estar correta a anotação em alguns casos. Portanto, para fins de tornar o algoritmo mais abrangente e tolerante com erros de tipagem, ou erros de anotação de palavras com a interface da ferramenta WKS, um algoritmo de cálculo de semelhança de *strings* é aplicado.

Para a verificação de similaridade entre duas *strings*, é implementado um algoritmo para o cálculo da distância de Damerau–Levenshtein, que quantifica a "distância" entre as duas sequências de caracteres – de maneira mais informal, esse algoritmo calcula o número de operações necessárias para transformar uma *string* na outra, consistindo em inserções, exclusões ou substituições de um único caractere, ou transposição de dois caracteres adjacentes (LI *et al.*, 2006). Um filtro do tipo *threshold* será usado para considerar uma anotação de entidade correta ou não – se a distância de

Damerau-Levenshtein for superior a um certo valor limite, as *strings* são consideradas diferentes, portanto, a entidade foi marcada de maneira errada e o participante não recebe pontos referentes a esse item.

Antes de iniciar a comparação das strings, o algoritmo precisa filtrar marcações que são obviamente erros: trechos de texto do gabarito que não devem conter marcações, mas o modelo do participante marcou mesmo assim. Para tal são avaliados os valores do vetor **location** retornado pela API do Watson, que são dois inteiros indicando a cardinalidade do caractere de início e de fim da palavra (ou palavras) que foram marcads. Para cada marcação extra (além do número esperado), o algoritmo irá descontar uma certa pontuação do participante. O cálculo da pontuação final é realizado segundo a Equação (2):

$$final\_score = \frac{1}{n+m} \sum_{i=1}^{n} p_i c_i \tag{2}$$

Em que n é o número total das entidades esperadas pelo pontuador e m são as entidades marcadas pelo modelo do participante que não deveriam ser marcadas. p_i e c_i são, respectivamente, a pontuação parcial e a confiança do modelo do participante. A variável **final_score** sempre possuirá um valor entre 0 e 100, e a nota do participante é reduzida de maneira diretamente proporcional ao número de entidades desnecessárias que forem marcadas pelo modelo.

As seguintes situações listadas são previstas como marcações corretas pelo algoritmo, garantindo pontuações parciais, ou seja, $p_i = 100$:

1. Texto com distância de Damerau-Levenshtein abaixo do tolerado é marcado como entidade do tipo esperado.

Já as outras situações, listadas a seguir, são consideradas erros e o participante recebe zero como pontuação parcial, ou seja, $p_i = 0$:

1. Texto com distância de Damerau-Levenshtein abaixo do tolerado, é marcado como entidade do tipo diferente do esperado.
2. Texto com distância de Damerau-Levenshtein acima do tolerado é marcado como entidade esperada ou não (distâncias fora da tolerância sempre são marcações erradas).

Ao fim do processo de comparação das respostas do Watson com o gabarito de *strings* esperadas (que é codificado rigidamente no algoritmo), uma variável do tipo *string* chamada **report** é construída com informações sobre cada etapa da avaliação, em notação JSON para posterior *parsing* simplificado em aplicações *Web*. No Código 13 é apresentado um exemplo de uma estrutura JSON resultante deste processo.

Código 13 – Exemplo simplificado de relatório gerado pelo algoritmo.

```
{
    "report": [
        {
            "expected": {
                "type": "Employee",
                "text": "Vanderlei",
                "location": [0 8]
            },
            "marked": {
                "type": "Employee"
                "text": "Vanderlei "
                "location": [0 9],
                "confidence": 1.0,
                "dl_dist": 1
            },
            "partial_score_1-of-2": 100
        },
        {
            "expected": {
                "type": "Company",
                "text": "IBM Corporation",
                "location": [110 124]
            },
            "marked": {
                "type": "Employee"
                "text": "IBM Corporation "
                "location": [110 124],
                "confidence": 0.98,
                "dl_dist": 0
            },
            "partial_score_2-of-2": 0
        }
    ],
    "final_score": 50
}
```

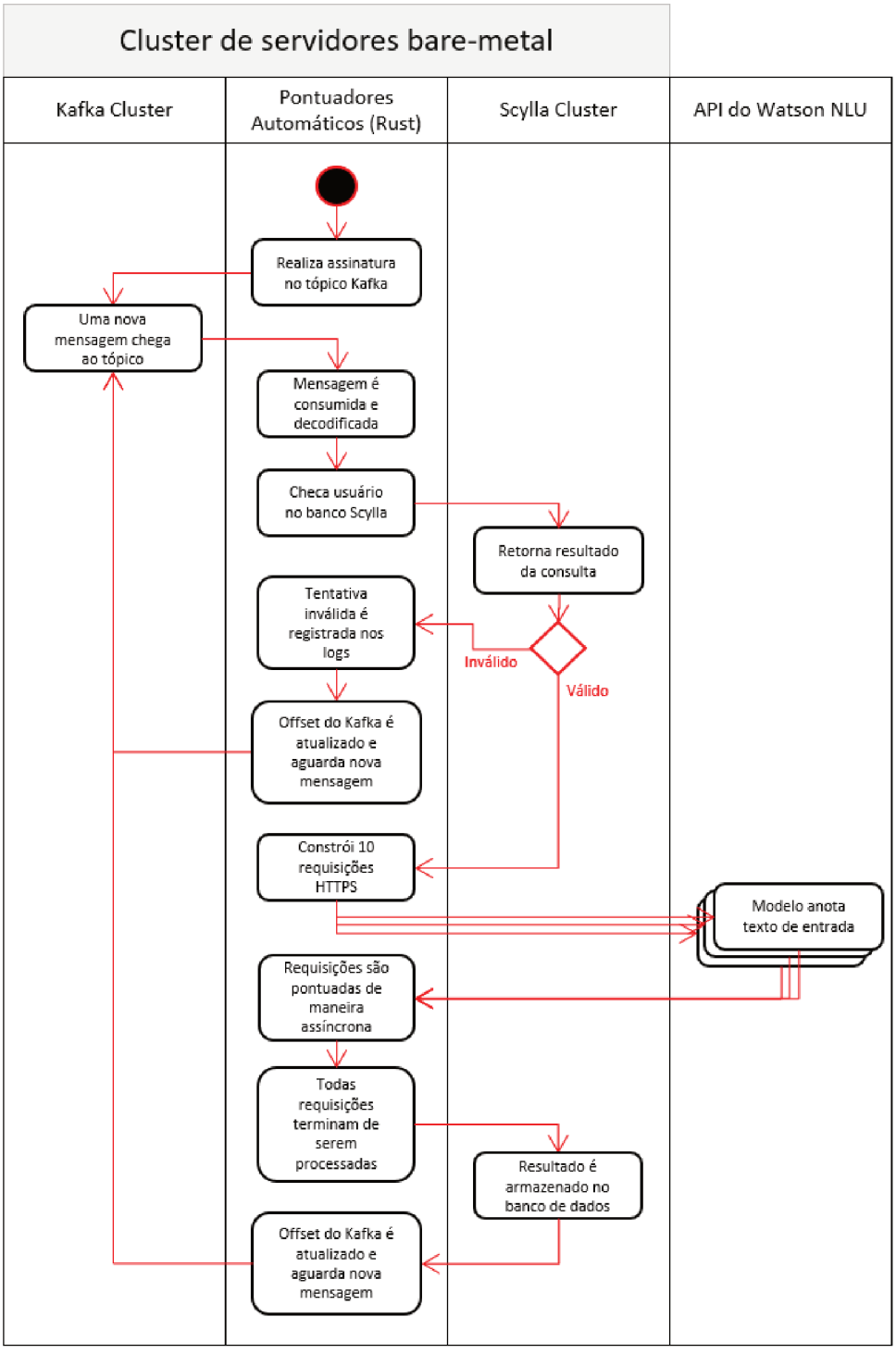
Nota-se que a segunda entidade avaliada foi marcada na posição correta, entretanto o tipo marcado é diferente do esperado, contabilizando zero pontos parciais nesse caso, independentemente da distância de Damerau-Levenshtein nula.

5.6.6 Visão geral do pontuador automático

Na prática, o algoritmo espera pela conclusão de 10 resultados futuros aninhados, que inciaram-se com as chamadas HTTPS para a API do Watson. Na Figura 32

é apresentado o fluxo de alto nível de execução de um pontuador automático e os componentes com os quais ele se comunica.

Figura 32 – Fluxo de atividades do algoritmo de pontuação automática.



Fonte: Arquivo pessoal.

O paralelismo no consumo de mensagens do Kafka é realizado por meio da execução de múltiplas instâncias do binário executável pelo sistema operacional, e

esse *cluster* de pontuadores é registrado como um *consumer group* no tópico Kafka, onde cada instância do binário consome mensagens únicas que são distribuídas nas partições do tópico.

5.7 DESENVOLVIMENTO DOS *WEB* PROXIES

Segundo o esquemático do sistema planejado apresentado na Figura 20, os *proxies* são representados pelos componentes B (*proxy* para acesso ao Kafka) e H (*proxy* para acesso ao Scylla). Ambos componentes serão hospedados e executados no *cluster OpenShift*, e portanto será necessária a criação de imagens CRI-O para o encapsulamento das aplicações. Em relação à implementação em código, foi escolhido o *framework* enxuto para a criação de *Web servers* conhecido como **actix-web**, apresentado na Seção 2.5.4.

Os *proxies Web* serão duas aplicações separadas, entretanto poderiam ser uma única aplicação sem maiores complicações. Foi optado por separá-las pois uma das rotas seria utilizada com muito mais frequência que a outra (o acesso aos *rankings* será muito mais requisitado do que a aplicação para submissão de soluções). Mesmo sendo aplicações separadas, em nível de lógica de implementação elas são bem semelhantes.

A imagem CRI-O, ou Docker, é idêntica para os dois componentes. Um arquivo **Dockerfile** personalizado é construído pelo autor deste trabalho para a compilação dos executáveis das aplicações e subsequente execução – o arquivo é apresentado no Código 14.

Código 14 – Dockerfile das imagens para aplicações Rust.

```
# --- Build Image
FROM us.icr.io/vnderlev-cr/pfc-ubuntu1804:latest as cargo-build

## Update cargo-build
RUN apt-get update

## Install required C libraries
RUN apt-get install -y cmake musl-tools libhwloc-dev hwloc

## Test hwloc lib
RUN lstopo

## Install musl-gcc
RUN git clone git://git.musl-libc.org/musl
WORKDIR /musl
RUN ./configure --prefix=/usr/src/rust_application
RUN make install
```

```
18 RUN musl-gcc --version
19
20 ## Copy local files to WORKDIR
21 COPY . /usr/src/rust_application
22
23 ## Set WORKDIR
24 WORKDIR /usr/src/rust_application
25
26 ## Setup rustlang for cross-platform-compilation
27 RUN rustup target add x86_64-unknown-linux-musl
28
29 ## Build rust-application
30 #RUN CC_x86_64_unknown_linux_musl="x86_64-linux-musl-gcc" \
31 #    cargo build --target x86_64-unknown-linux-musl --release
32 RUN cargo build --target x86_64-unknown-linux-musl --release
33
34 # Check if binary is built correctly
35 RUN file /usr/src/rust_application/target/x86_64-unknown-linux-musl/
    release/rust_application
36
37 # --- Release Image
38 FROM alpine:latest
39
40 # Upgrade all system packages
41 RUN apk --no-cache update && \
42     apk --no-cache upgrade
43
44 # Set WORKDIR
45 WORKDIR /home/rust_application/bin/
46
47 # Copy binaries from cargo-build
48 COPY --from=cargo-build \
49         /usr/src/rust_application/target/x86_64-unknown-linux-musl/
    release/rust_application \
50         .
51
52 # Copy entrypoint shell script
53 COPY ./entrypoint.sh .
54
55 # set entrypoint.sh file permissions
56 RUN chmod +x /home/rust_application/bin/entrypoint.sh
57
58 # Entrypoint
59 CMD ["/home/rust_application/bin/entrypoint.sh"]
```

Na **Dockerfile** apresentada, a construção da imagem é realizada em duas etapas. Primeiro, o código-fonte é simplesmente transportado para um contêiner baseado

no *Linux Ubuntu* (imagem **ubuntu1804-musl:latest**, também criada pelo autor deste trabalho, e hospedada no registro privado da IBM). Na imagem construída, baseada no *Ubuntu 18.04 LTS*, estão instalados o compilador *Rust* e o compilador *C*, assim como todas as dependências de linguagem *C* que serão utilizadas para construção dos executáveis dos dois *Web proxies*, e a biblioteca padrão *Musl*[16]. Após a compilação dos binários executáveis, a construção da imagem passa para o segundo estágio, onde os executáveis são transportados para uma imagem final baseada no sistema operacional *Alpine Linux*, que é extremamente leve. Dessa forma, temos como resultado uma aplicação miniaturizada, capaz de ser escalada rapidamente pelo *OpenShift* quando necessário. A imagem do *Linux Ubuntu* é destruída após o processo, restando apenas as camadas da imagem miniaturizada – que ao final do processo possui apenas *17 megabytes*. Para fins de comparação, uma imagem *Ubuntu* com a JVM instalada chega próximo de 2 *gigabytes* de tamanho. A instanciação de novos contêineres baseados em imagens grandes é consideravelmente demorada, o que não será um problema para os *proxies* desenvolvidos, já que são implementados como binários executáveis em imagens miniaturizadas.

Uma vez construída a imagem, a implantação no *cluster OpenShift* é realizada por meio de arquivos de extensão **yaml**, que definem como a aplicação será organizada no *cluster* – de maneira idêntica ao Kubernetes, as aplicações podem ser implantadas em uma combinação de *Deployments*, *Services*, e recursos *Ingress*. Neste trabalho ambos os *proxies* são implantados no *OpenShift* como *Deployments*, com um *Service* e o *Ingress* correspondente. No arquivo **yaml** é também configurada a quantidade de réplicas e política de carregamento de imagens. Os *Services* e *Ingress* basicamente expõem uma rota entre os às aplicações replicadas e a Internet, realizando o balanceamento de carga de maneira automática.

Todo o processo de construção dos contêineres é automatizado por meio de *shell scripts*, que facilitam a cadeia de desenvolvimento – uma alteração no código-fonte das aplicações *Rust* é realizada, e com a simples execução do *script*, o novo código é empurrado para um repositório *Git*[17] para controle de versionamento, o processo de construção da imagem é realizado, a imagem é armazenada em um registro privado na IBM Cloud, e o *cluster OpenShift* realiza a atualização da aplicação sem interrupção do serviço.

A estrutura principal de ambas as aplicações é idêntica, baseada na biblioteca **actix-web**. São configurados *middlewares* do tipo *logger* padrão, disponibilizados na própria biblioteca. É desabilitado o padrão TLS nas requisições HTTP das aplicações, pois essa função é delegada diretamente ao balanceador de carga do *Red*

[16] O *Musl* é uma implementação da *libc*, ou biblioteca padrão da linguagem *C*, segundo as especificações de funcionalidades descritas na ISO *C* e nas normas técnicas *POSIX*.

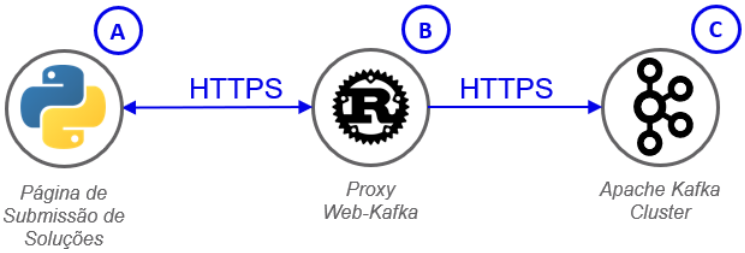
[17] O *Git* é um sistema distribuído de controle de versões de arquivos, criado por Linus Torvalds e muito utilizado para coordenar trabalho entre programadores.

Hat OpenShift – baseado no Nginx – e toda comunicação entre o balanceador e os contêineres, como é interna, não é criptografada para melhoramento de desempenho (essa é uma prática comum, tanto no *Kubernetes* quanto no *OpenShift*, e não acarreta em brechas de segurança).

5.7.1 Proxy para produção de mensagens Kafka

Na Figura 33 é apresentada parte do esquemático geral do sistema, destacando o Kafka, o *Web proxy* implementado, e a aplicação *Web* para submissão de soluções.

Figura 33 – Integrações do *Web proxy* para o *cluster* Apache Kafka.



Fonte: Arquivo pessoal.

O fluxo de processamento de uma requisição ao *proxy* para o Kafka inicia-se pela codificação do conteúdo da mensagem recebida da aplicação de submissões (parâmetros são enviados por meio de um formulário), é posteriormente é construída uma estrutura de dados do tipo JSON com uma chave **records** com o conteúdo codificado como valor. Essa estrutura JSON será o corpo de uma requisição HTTP que será feita pelo *Web proxy* para a API de produtores do *cluster* Kafka, e o código em *Rust* para construção dela é apresentado no Código 15.

Código 15 – Esquemas dos dados tratados no proxy para o Kafka.

```
let value = serde_json::json!(
{
    "nlu_api_key": payload.0.nlu_api_key,
    "nlu_model_id": payload.0.nlu_model_id,
    "cpf": payload.0.cpf,
    "participant_id": payload.0.participant_id
});

let value_b64 = base64::encode_config(&value.to_string(), base64::
    STANDARD);

let record = serde_json::json!(
{
```

```
    "records": [
        {
            "value": value_b64
        }
    ]
});
```

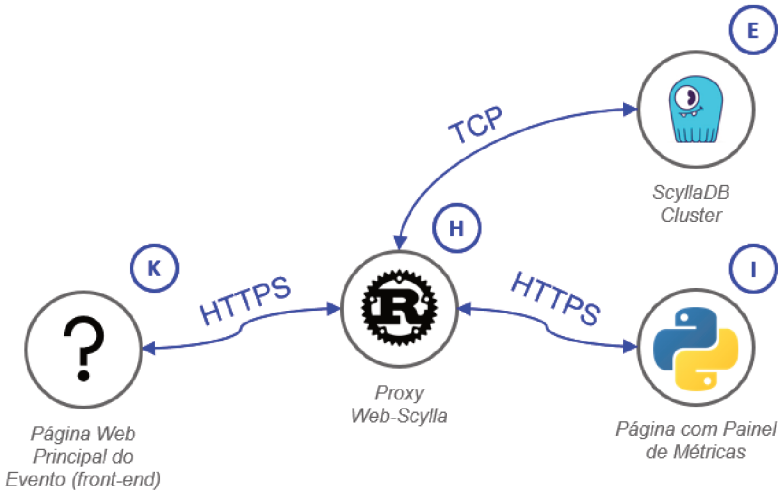
Após a construção da mensagem Kafka a ser enviada, o cabeçalho da requisição é também configurado, com as chaves de segurança e endereços do *cluster*, e logo em seguida é disparada a requisição HTTP de maneira assíncrona – o *Web proxy* não irá ficar bloqueado enquanto aguarda os ACKs do *cluster* Kafka.

O papel desta aplicação é cumprido então sucintamente, sendo o de intermediário entre as requisições realizadas pelas aplicações *Web* e o *cluster* Apache Kafka. Esse *Web proxy* facilita o ciclo de desenvolvimento de *software* na empresa, que pode delegar o desenvolvimento das interfaces das aplicações *Web* para terceiros, ou para outros desenvolvedores especializados, mantendo desacoplada a lógica de segurança.

5.7.2 Proxy para consultas ao Scylla

Na Figura 34 é apresentada parte do esquemático geral do sistema, destacando o banco de dados Scylla, o *Web proxy* implementado e as aplicações *Web* que interagirão com ele.

Figura 34 – Integrações do Web proxy para o cluster Scylla.



Fonte: Arquivo pessoal.

O *proxy* para acesso ao banco de dados do sistema possuirá quatro rotas, listadas a seguir:

1. Rota para apresentar os 100 participantes mais bem pontuados (em um desafio individual ou no geral);
2. Rota para apresentar uma tabela com informações detalhadas de todas as submissões (de um desafio individual) com métricas importantes;
3. Rota para apresentar uma tabela com informações demográficas detalhadas dos participantes;
4. Rota para apresentar uma tabela com informações e métricas detalhadas sobre o engajamento no evento.

Cada uma quando requisitada por uma aplicação cliente, irá realizar consultas com senteças preparadas e parametrizadas ao banco Scylla, de maneira a nunca permitir um ataque do tipo *injection*. As APIs do *Web proxy* aceitam o mínimo possível de parâmetros (apenas um número inteiro para filtragem dos dados por desafio) e já são preparadas para cada caso de uso planejado para o sistema.

5.8 DESENVOLVIMENTO DA APLICAÇÃO DE SUBMISSÃO

A aplicação para a submissão de soluções é uma aplicação personalizada para cada desafio, que pode ser executada tanto diretamente na máquina do participante como na IBM Cloud. Durante uma competição futura, essa aplicação será fornecida com erros propositais, para que o participante os descubra e corrija-os antes de poder submeter sua solução.

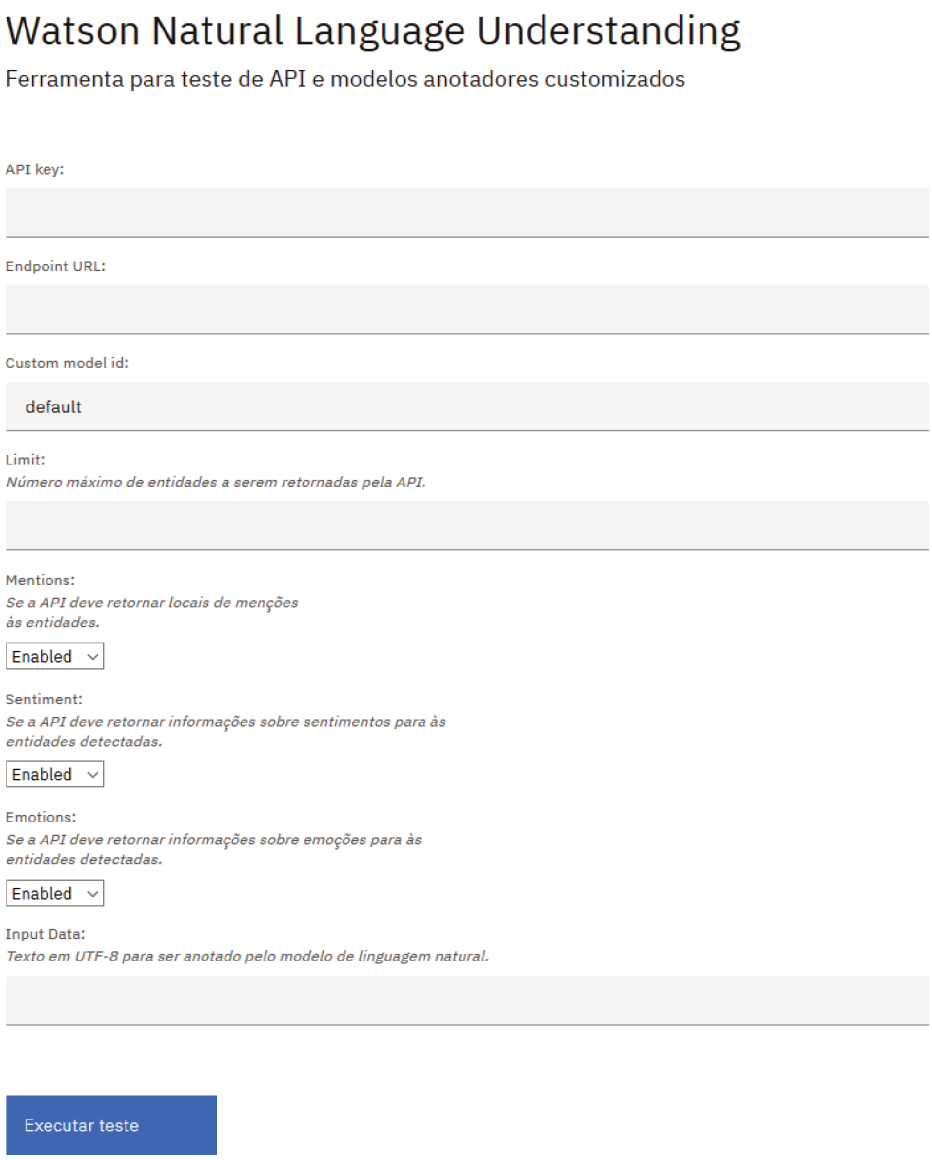
Para o caso de estudo escolhido neste trabalho, e de acordo com os requisitos funcionais definidos no Quadro 5, foram implementados formulários para teste de API de modelo anotador textual do Watson e para submissão de modelo para avaliação. A aplicação é simples, sem a configuração de *gateway* uWSGI, como é feito para o painel de métricas – já que só haverá um único usuário para cada instância da aplicação em execução: o próprio participante, dispensando balanceamento de carga.

É utilizado o pacote *Python* **ibm-watson** e **watson-developer-cloud** para criar sessões e realizar chamadas diretamente à API do *Watson Natural Language Understanding*. Na aplicação *Flask* é definida uma única rota, que irá apresentar o conteúdo estático (interface e formulário) para o usuário. Como supracitado em várias ocasiões, para a realização de um teste de API o participante deve informar sua **api-key**, **model-id**, e **input** (texto de entrada), com parâmetros opcionais como **url-endpoint** e **limit** (número máximo de entidades que devem ser retornadas pelo Watson). Clicando no botão de teste, a biblioteca **flask-tables** é acionada para gerar o código estático HTML de uma tabela organizada, apresentando todas as entidades e relações anotadas pelo modelo de maneira de fácil compreensão por um humano, ao contrário da resposta

pura em formato JSON. Os resultados, que aparecem prontamente na tela, acompanham um botão para retornar ao formulário de testes, e também um novo formulário, para que o participante realize a submissão da sua solução para pontuação automática. Uma vez preenchidos os dados e pressionado o botão de submissão, uma chamada é realizada ao *Web proxy* criado para intermediar a produção de mensagens no Apache Kafka.

Na Figura 35 é apresentada a interface gráfica da aplicação de teste e submissão de soluções (modelo anotador de texto criado com WKS). A página principal da aplicação é um formulário simples, conforme especificado nos requisitos.

Figura 35 – Interface gráfica: tela de teste de modelo.

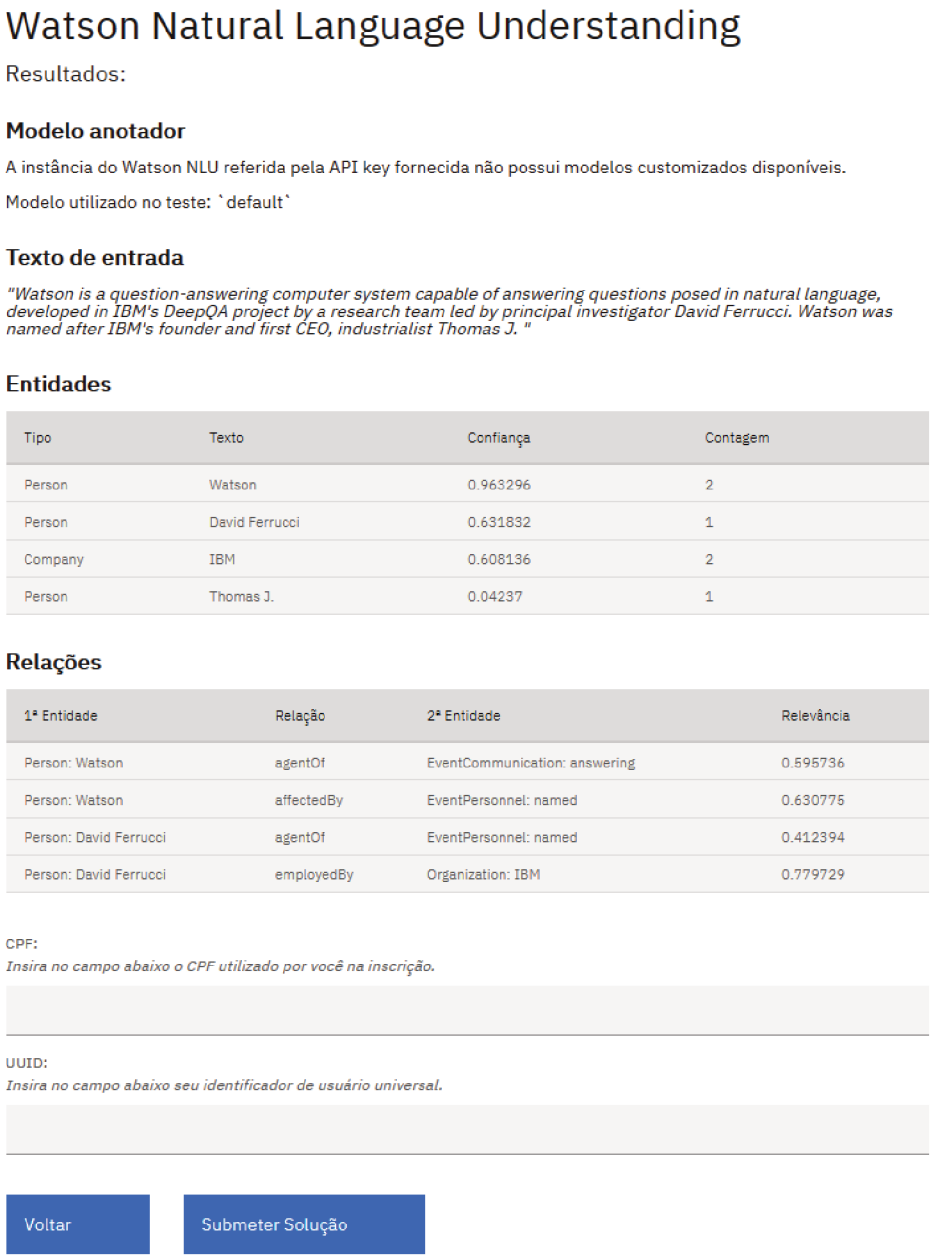


Fonte: Arquivo pessoal.

Na Figura 36 é apresentada a tela da interface gráfica da aplicação de teste e

submissão de soluções, após a realização de um teste de modelo. É indicada a tabela gerada com os resultados – entidades e relações entre entidades anotadas, nesse caso pelo modelo **default** do Watson para um exemplo de texto de entrada em língua inglesa.

Figura 36 – Interface gráfica: tela de resultados de modelo.



Fonte: Arquivo pessoal.

Essa aplicação será executada nas máquinas dos participantes e também não existem requisitos de desempenho ou funcionamento sem interrupção. Dessa forma, o servidor experimental integrado ao Python Flask é usado para servir a aplicação.

5.9 DESENVOLVIMENTO DO PAINEL DE MÉTRICAS

5.9.1 Arquitetura de implantação da aplicação Flask

O painel de métricas é uma aplicação *Web* escrita em *Python*, implantada de modo semelhante aos *Web proxies* escritos em *Rust*. Cada aplicação é encapsulada em um contêiner e executada no *cluster OpenShift* de maneira replicada. Foi desenvolvido um arquivo **Dockerfile** para a construção em dois estágios das imagens dos contêineres que hospedarão a aplicação. Assim como no caso dos *proxies*, a imagem oficial do *Alpine Linux* é utilizada como base, na qual o Python 3.8.0 e bibliotecas necessárias são importadas – uma imagem extremamente pequena é gerada (de cerca de *12 Megabytes*). A **Dockerfile** da imagem base para a aplicação em *Python Flask* é apresentada no Código 16.

Código 16 – Dockerfile das imagens para aplicações Python.

```
# --- Base Image
FROM alpine:latest

# Upgrade all system packages and install dependencies
RUN apk --no-cache update && \
    apk --no-cache upgrade && \
    apk add --no-cache \
    git bash pcre-dev libffi-dev openssl-dev bzip2-dev \
    zlib-dev readline-dev build-base postgresql-dev \
    linux-headers py3-netifaces jpeg-dev xz-dev

# Set Python version
ARG PYTHON_VERSION='3.8.0'

# Set pyenv home
ARG PYENV_HOME=/root/.pyenv

# Install pyenv, then install python versions
RUN git clone --depth 1 https://github.com/pyenv/pyenv.git $PYENV_HOME
   && \
    rm -rfv $PYENV_HOME/.git
ENV PATH $PYENV_HOME/shims:$PYENV_HOME/bin:$PATH
RUN pyenv install $PYTHON_VERSION
RUN pyenv global $PYTHON_VERSION
RUN pip install --upgrade pip && pyenv rehash

# Install requirements
RUN pip install \
    uwsgi click itsdangerous MarkupSafe Jinja2 Werkzeug flask \
    Six flask-cors flask_table setuptools netifaces flask_socketio \
```

```
30     greenlet monotonic dnspython gevent pillow pandas numpy bokeh
31
32 # Clean
33 RUN rm -rf ~/.cache/pip
34
35 # Set working directory
36 WORKDIR /usr/src/app
37
38 # Add flask server files
39 COPY . /usr/src/app/
40
41 # Set entrypoint.sh permissions
42 RUN chmod +x /usr/src/app/entrypoint.sh
43
44 # Run server
45 CMD ["/usr/src/app/entrypoint.sh"]
```

O conteúdo do Código 16 apresentado acima apresenta o primeiro estágio de construção da imagem das aplicações, que consiste na preparação do ambiente Alpine Linux, com a instalação inicial de dependências como o **openssl** e posterior instalação limpa do *Python 3.8.0*, com a inclusão de diversas bibliotecas via gerenciador de pacotes Pip[18]. Por fim é definido o *shell script* /**usr**/**src**/**app**/**entrypoint.sh** como ponto de entrada de execução do contêiner, apresentado no Código 17.

Código 17 – Conteúdo do arquivo entrypoint.sh.

```
1 #!/bin/sh
2 echo "Starting Flask Server..."
3 uwsgi --ini uwsgi.ini
```

O *shell script* definido como ponto de entrada para o contêiner serve para iniciar o uWSGI, que irá realizar a ponte de comunicação entre as requisições do balanceador de carga do *OpenShift*, e instâncias multiplicadas da aplicação em *Python Flask*.

O arquivo de configuração do uWSGI é apresentado no Código 18.

Código 18 – Conteúdo do arquivo uwsgi.ini.

```
1 [uwsgi]
2 module = flask_server:flask_server
3 pcre = True
4 ssl = True
5 zlib = True
6 debug = False
```

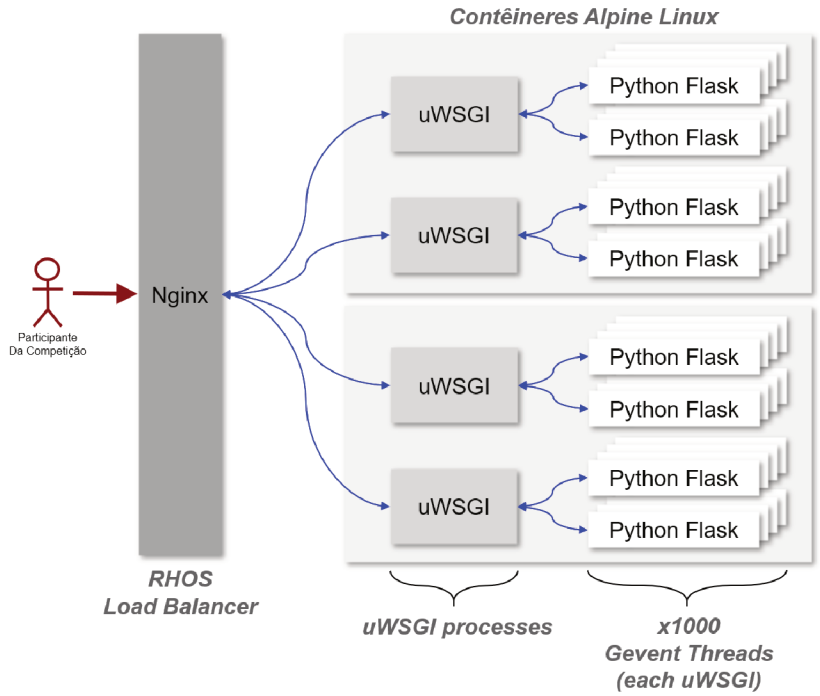
[18] Gerenciador de pacotes, ou bibliotecas de *software*, padrão da linguagem de programação *Python*.

```
malloc = libc
master = True
processes = 2 // numner of uWSGI instances
gevent = 1000 // number of Python async cores
http = :5000
http-websockets = True
vacuum = True
die-on-term = True
```

O uWSGI é configurado então como *proxy* reverso, recebendo comunicação do servidor Nginx (OpenShift) e transportando-a pra as múltiplas instâncias de aplicações Flask replicadas. Nota-se que o uWSGI está a nível do contêiner, o que significa que cada contêiner executará um *proxy* reverso uWSGI (2 processos em cada contêiner, mais especificamente), com múltiplas instâncias da aplicação Flask subordinadas. O uWSGI é configurado para utilizar 1000 *threads* Gevent – uma biblioteca de alto nível que oferece funcionalidades avançadas de programação assíncrona sobre a API nativa do Python (*threads* do tipo M-N).

A Figura 37 apresenta arquitetura da aplicação Flask em termos de servidores, *proxies* e clientes.

Figura 37 – Balanceamento de carga nas aplicações Python.



Fonte: Arquivo pessoal.

Essa configuração daria teoreticamente a cada contêiner a capacidade de tratar 2000 requisições HTTP simultaneamente (embora o painel de análises não precise de

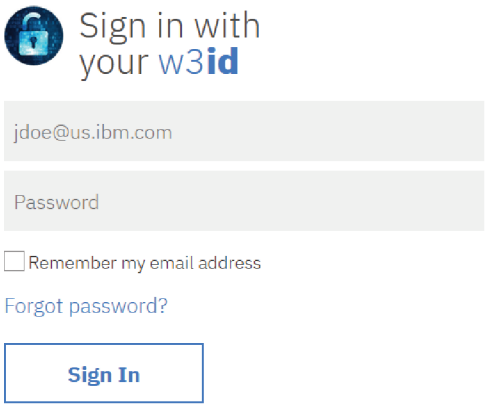
uma banda de serviço desse tamanho, essa arquitetura de implantação de aplicação Python é bem interessante caso o uso da linguagem *Rust* não seja muito bem recebido por futuros desenvolvedores, visto que a linguagem possui uma difícil curva de aprendizado. Sendo assim, existe a opção de implementar também os *proxies* como aplicações Flask.

5.9.2 Camada de segurança implementada

Como nova funcionalidade ao painel de análise, foi implementada uma camada de segurança baseada em autenticação de dois passos – implementada com a API de login corporativo da IBM (W3id), que faz todo o processo de envio de código de autenticação e gerenciamento de senhas. Dessa forma, o painel só pode ser acessado pela intranet ou pela VPN[19] da IBM.

Na Figura 38 é apresentada a tela de login, que é gerada automaticamente pelo serviço de login corporativo da IBM.

Figura 38 – Tela de login no painel de análises.



Fonte: Arquivo pessoal.

5.9.3 Interface desenvolvida para o painel de análises

Conforme o especificado, a interface desenvolvida para o painel de análises é também baseada no *Carbon Design System (CDS)* da IBM, entretanto dispensando a necessidade de instalação do *Node.js* e os milhares de pacotes e dependências usualmente necessários para se usar o CDS, alcançando um visual praticamente idêntico embora sem as facilidades de um *framework* de desenvolvimento *front-end* para *Web*.

[19] *Virtual Private Network*, é um mecanismo de *software* para estender uma rede privada através de uma rede pública. Computadores conectados via VPN aparentam estar conectados em uma rede privada, mesmo acessando-a através de uma rede pública.

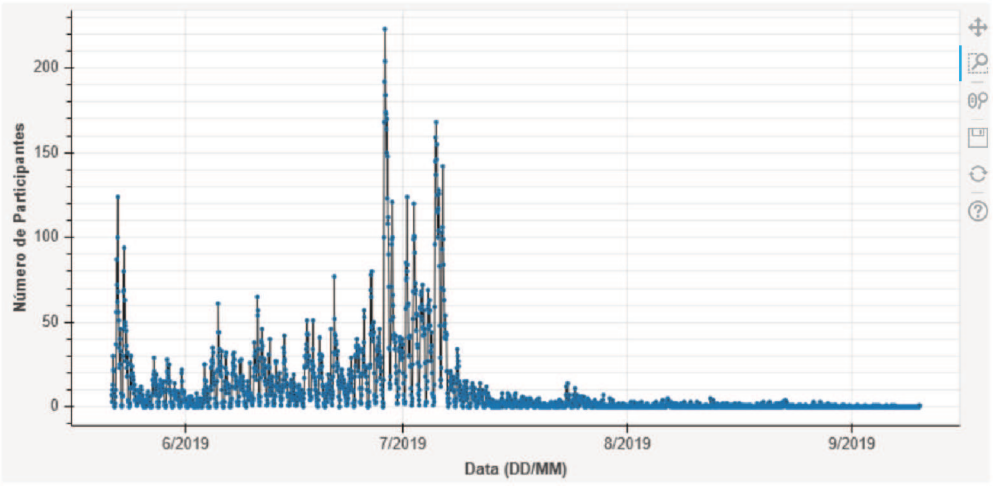
Na Figura 39 é apresentada a tela principal do painel de análises, com algumas informações gerais sobre o evento piloto.

Figura 39 – Tela principal do painel de análises.

Fonte: Arquivo pessoal.

Na Figura 40 é apresentada uma visualização interativa da frequência de novas inscrições, com totais de novos registros por período de uma hora, durante todo o evento piloto.

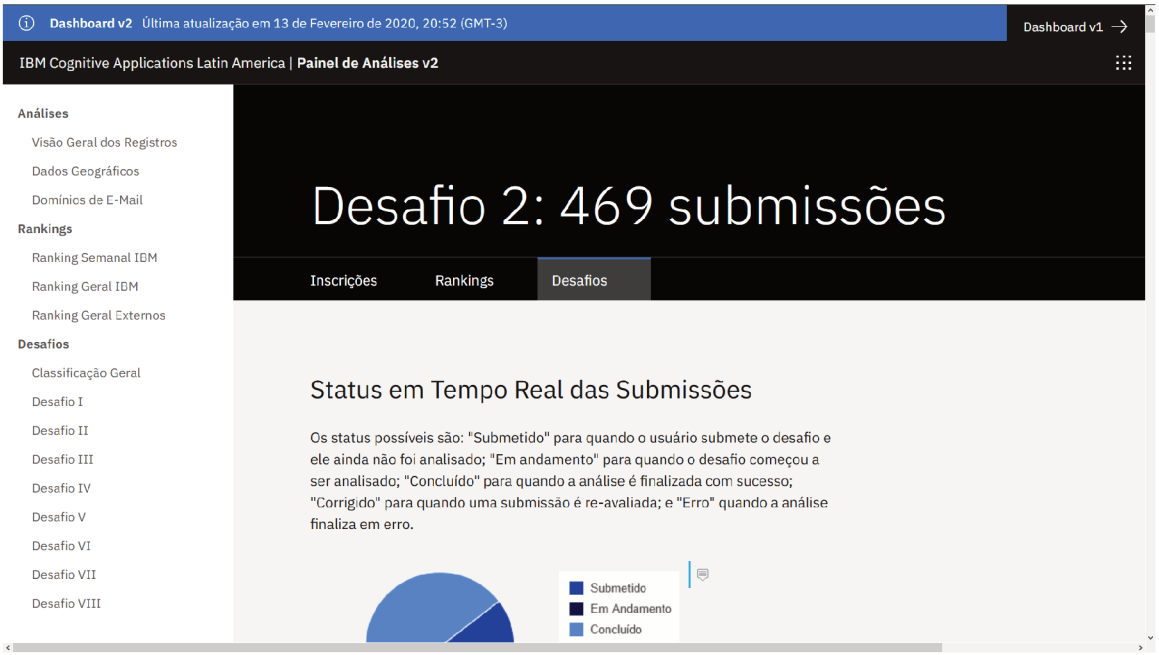
Figura 40 – Gráfico interativo com frequência de novos participantes.



Fonte: Arquivo pessoal.

Na Figura 41 é apresentada a tela com informações detalhadas de um desafio específico. Foi criada uma função para a geração de código *javascript* dos gráficos interativos (com o uso da biblioteca **bokeh** da linguagem *Python*), e todas as páginas dos desafios seguem um mesmo *template*, com visualizações em gráfico pizza do status das submissões, gráfico de barras com a distribuição de pontuações obtidas, e uma tabela com ranking das submissões.

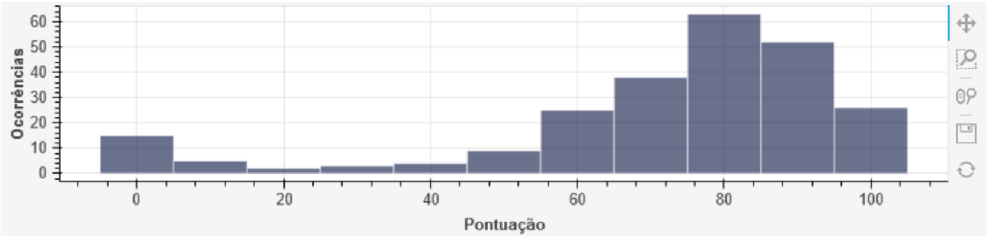
Figura 41 – Tela de detalhamento de desafios do painel de análises.



Fonte: Arquivo pessoal.

Nessa mesma página é gerada uma visualização do mesmo tipo da Figura 40, com a frequência de submissões para determinado desafio. Além disso, são apresentados histogramas de pontuação, apresentados na Figura 42.

Figura 42 – Distribuição de pontuação em um dos desafios do sistema piloto.



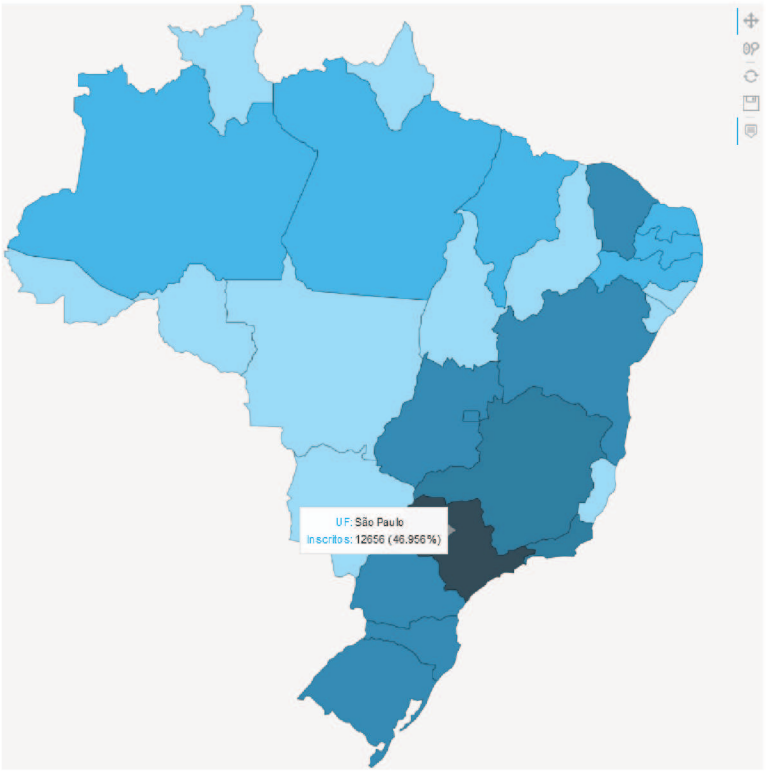
Fonte: Arquivo pessoal.

Na Figura 43 é apresentada a tela de análise geográfica dos participantes do evento, com mapa coroplético do Brasil.

Figura 43 – Tela de dados geográficos do painel de análises.

Fonte: Arquivo pessoal.

Figura 44 – Mapa coroplético do Brasil (dados reais do evento piloto).



Fonte: Arquivo pessoal.

6 ANÁLISE DO SISTEMA, TESTES E RESULTADOS

Uma vez finalizada a implementação da primeira versão completa do sistema, passamos para a análise dos resultados. O sistema foi avaliado segundo duas métricas principais: a qualidade das interfaces de usuário (painel de análise e interface de linha de comando); e o desempenho do sistema, sobretudo a capacidade de tratamento simultâneo das submissões de soluções (pontuação) e requisições simultâneas às aplicações *Web* do sistema. Também são comentados todos os outros objetivos, desde a segurança, tratamento da privacidade, custos e manutenção, e também sobre o método de desenvolvimento adotado.

6.1 INTERFACES DE USUÁRIO

Quanto às interfaces de usuário desenvolvidas, antes mesmo do início do desenvolvimento deste trabalho (durante o evento piloto), o autor foi responsável pelo desenvolvimento do antigo painel de análises, que já havia sido avaliado como excelente pelos orientadores e outros superiores na empresa. Nesse componente do sistema, apenas alguns pequenos ajustes foram realizados, sobretudo na camada de segurança, enquanto que a nova interface permaneceu bem semelhante à anterior – a maioria das mudanças deu-se no *back-end* da aplicação, e na otimização do conteúdo estático das páginas *Web* (miniaturização de CSS e HTML). A interface dessa aplicação foi novamente avaliada como excelente pelos orientadores e colegas de trabalho.

Foram adicionados mecanismos para filtragem de informações nas tabelas. Anteriormente, isso não era possível – todas as informações eram apresentadas, e ordenadas apenas por posição no ranking ou *timestamp* de registro. No novo painel de análise é possível filtrar e ordenar dados praticamente em qualquer combinação, nova mudança que foi muito bem recebida pelos usuários finais.

Quanto à interface de linha de comando para gerenciamento das camadas de dados do sistema e validadores, por ser uma ferramenta com frequência de utilização baixa e apenas por desenvolvedores (administradores do sistema), ela cumpre de maneira direta os requisitos funcionais sem precisar de uma interface gráfica – enviar comandos diretamente às APIs do Kafka, do Scylla, e dos pontuadores automáticos.

6.2 SEGURANÇA E PRIVACIDADE

Quanto à segurança, foram implementados mecanismos de desacoplamento do banco de dados do sistema e das interfaces expostas ao público, por meio do uso do barramento de dados – essa foi a maior mudança em relação à arquitetura do fluxo de dados do sistema novo para o sistema piloto. Além disso, as requisições

enviadas passam pelos *Web proxies* e são processadas previamente, antes mesmo de serem armazenadas temporariamente nos tópicos do *cluster Kafka*. Dessa forma, o funcionamento do sistema de pontuação e tratamento de inscrições é completamente opaco ao usuário final, que é incapaz de realizar ataques do tipo *SQL Injection* (além de que, o banco de dados utilizado não é nem mesmo compatível com SQL).

O usuário final também recebe apenas um *feedback* mínimo sobre as entradas que são realizadas no sistema – uma resposta com apenas dois estados: se a requisição chegou ou não ao *Web proxy* – se a requisição foi descartada por inconsistência nos dados e não foi escrita no tópico Kafka, o usuário nunca sabe de imediato, impossibilitando a sondagem do funcionamento interno do sistema. O único *feedback* disponível é o ranking que se altera conforme novas soluções são pontuadas, *feedback* esse que é indireto.

Quanto à privacidade, o banco de dados é completamente criptografado e todas as informações são anonimizadas antes de serem apresentadas no painel de análises. Qualquer participante pode requisitar a deleção completa de suas informações através de comunicação via e-mail, e a equipe tem um certo prazo (conforme legislação local) para realizar a remoção completa dos dados. Ao fim do evento todos os dados dos participantes são anonimizados permanentemente.

6.3 DESEMPENHO E BENCHMARKINGS

A análise de desempenho do sistema foi realizada utilizando tanto ferramentas de *software* específicas para testes de desempenho de aplicações, quanto *scripts* com testes automatizados preparados pelo próprio autor desse trabalho. Foram avaliados dois componentes críticos: o *Web proxy* que integra com o *cluster* Kafka, e as aplicações de pontuação automática de soluções. Em outras palavras, o produtores e os consumidores conectados ao *cluster* Kafka.

Para a averiguação do desempenho do *Web proxy* para o Apache Kafka, foi utilizada a ferramenta *Apache Bench*[1] da Fundação Apache, construída originalmente para a realização de testes com fins de darem uma noção de como os servidores HTTP Apache conseguiam lidar com cargas de requisições – especialmente, essa ferramenta é capaz de indicar quantas requisições por segundo um servidor HTTP consegue tratar.

Os testes realizados com a ferramenta *Apache Bench* são configurados da seguinte forma:

1. O parâmetro **-c** indica o nível de concorrência do teste de carga, e é o que indica a escala do estresse aplicado no *Web proxy*;

[1] Página oficial da ferramenta disponível em `https://httpd.apache.org/docs/2.4/programs/ab.html` (acessado em: 09/02/2020).

2. O parâmetro **-n** indica a quantidade de requisições totais que serão realizadas durante todo o teste – essa opção apenas define o tamanho do teste de carga a ser realizado; e

3. O parâmetro **-k**, que é um valor booleano, é configurado como *true*. Esse parâmetro habilita a funcionalidade "KeepAlive" dos *browsers* modernos (que é habilitada por natureza). Como o objetivo é simular o uso da aplicação por outros usuários, essa opção é habilitada.

Aplicando o comando apresentado no Código 19, estaremos aplicando 500 conexões simultâneas até que 20000 requisições sejam tratadas pelo *proxy*.

Código 19 – Exemplo de comando para execução de teste com a ferramenta Apache Bench.

```
1 ab -k -c 500 -n 20000 http://host:port/
```

Na Quadro 13 são apresentados os resultados realizados no *Web proxy* escrito em linguagem *Rust* e hospedado no *cluster OpenShift*.

Quadro 13 – Resultados do teste de carga com Apache Bench para proxy Kafka.

Métrica ab.exe	**Resultado**
Concurrency Level	500
Time taken for tests	4812ms
Complete requests	20000
Failed requests	0
Non-2xx responses	20000
Keep-Alive requests	20000
Total transferred	438,095,236 bytes
HTML transferred	109,523,809 bytes
Requests per second	4166.67 [#/sec] (mean)
Time per request	120 [ms] (mean)
Time per request	0.24 [ms] (mean, across all concurrent requests)

Fonte: Arquivo pessoal.

Nota-se que o desempenho alcançado é considerado ótimo, levando em conta que na edição do evento piloto 26 mil participantes inscreveram-se, com cerca de 3 mil usuários ativos (que enviaram pelo menos uma solução para pontuação). Vale lembrar que no evento piloto não era permitida a re-submissão. Com a capacidade de tratamento de submissões identificada nos testes básicos com *Apache Bench*, já é possível notar que o novo sistema será capaz de atender à demanda prevista (500 submissões de soluções simultâneas é uma estimativa relativamente elevada). Para garantia ainda maior, basta escalar horizontalmente ou verticalmente o sistema,

alterando-se a infraestrutura subjacente e realizando a implantação novamente do sistema (que pode ser feita rapidamente com a ajuda dos *scripts* preparados).

Quanto aos testes do pontuador automático para o desafio hipotético de linguagem natural, um *script* preparado pelo autor foi utilizado, com a intenção de calcular os períodos de tempo gastos pelo algoritmo de pontuação em cada etapa: no lançamento das 10 requisições, e posteriormente na avaliação das respostas (algoritmo de cálculo da distância de Damerau-Levenshtein), e também na concatenação das *strings* no relatório de formato JSON. O critério de sucesso será a comparação com o mesmo tempo de processamento de uma submissão pelo sistema piloto, contando desde a leitura do banco de dados PostgreSQL até a escrita dos resultados.

Os resultados do teste comparativo de velocidade de processamento para o sistema piloto e o novo sistema são apresentados na Tabela 14.

Tabela 14 – Tempo de processamento para cada tarefa de pontuação.

Tarefa	**Novo Sistema**	**Sistema Piloto**
Consumo de mensagem Kafka via TCP binário e decodificação de base 64 para UTF-8	1.2ms	-
Disparo de gatilho de monitoração do PostgreSQL (1 serverless function)	-	170ms
Consulta ao PostgreSQL (1 linha), via HTTPS	-	345ms
Consulta ao ScyllaDB (1 linha), via TCP binário	11ms	-
Finalização de 10 requisições à API do Watson Natural Language Understanding (modelo com 10 entidades)	460ms	8102ms
Comparação com gabarito e cálculo da pontuação final (Damerau-Levenshtein para cada entidade no desafio – são 50 comparações). O novo sistema inclui logging aqui	10.2ms	1890ms
Logging (escrita em MongoDB via HTTPS)	-	5595ms
Tempo total para pontuação de uma solução	**483ms**	**16102ms**

Fonte: Arquivo pessoal.

Vale ressaltar que o teste realizado considera apenas a pontuação de uma submissão – o que não contabiliza os ganhos de paralelismo e também os provenientes da programação assíncrona, onde as esperas das requisições são não-bloqueantes. Esse teste visa mostrar que somente a alteração de uma arquitetura de microsserviços para uma aplicação “monolítica” já eliminou múltiplos pontos de comunicação interna através de rede, e consequentemente trouxe uma redução drástica de latência. Segundo a própria literatura atual sobre arquitetura de microsserviços, a alta latência decorrente do aumento de requisições entre os múltiplos serviços é um dos principais problemas a serem enfrentados – juntamente com a integração do sistema (NEWMAN, 2014). Além disso, a arquitetura de microsserviços traz um problema sério no aspecto de segurança da informação. Os vetores de ataque são multiplicados, em contraste

com a arquitetura “monolítica” de pontuador automático implementada.

Em um terceiro teste, é avaliada a métrica de soluções pontuadas por unidade de tempo, apresentados no Tabela 15 – nesse teste, os benefícios do paralelismo e do algoritmo assíncrono tornam-se aparentes.

Tabela 15 – Máquinas virtuais provisionadas para o sistema.

Sistema	**# de soluções pontuadas**	**Tempo de execução do teste**	**Taxa de pontuações**
Sistema Piloto (“3 nós” serverless)	10000	13.86 horas	12/minuto
Sistema Piloto (“3 nós” serverless com logging desligado)	10000	9.72 horas	17.1/minuto
Novo sistema (3 nós)	10000	201.25 segundos	2979/minuto
Novo sistema (3 nós com geração de relatório desligada)	10000	187.00 segundos	3205/minuto

Fonte: Arquivo pessoal.

Nota-se uma velocidade de pontuação bem superior ao sistema piloto – para fins de escala, o evento piloto no total possuía 9000 submissões (entre 8 desafios, que ocorreram num período de 6 semanas). Para a pontuação de todos eles com a funcionalidade de *logging* habilitada, seriam necessárias cerca de 14 horas, enquanto que o novo sistema realizaria essa pontuação em cerca de quatro minutos (com *logging*). Essa discrepância se dá na redução drástica do número de requisições HTTPS. No sistema piloto, cada função *serverless* precisava realizar autenticação com o banco de dados MongoDB para a escrita de *logs*, o que praticamente dobrava o número de requisições necessárias.

Na realização dos testes não foi de fato realizada uma requisição HTTPS para a API do Watson (dado que para uma situação real, seria necessária a implantação de milhares de modelos, e o uso de uma única API de modelo tornaria-se um gargalo claro no sistema). Então, foi realizada a simulação de uma requisição HTTPS ao Watson por meio da suspensão da *thread* por 350 milissegundos (tempo médio calculado de uma requisição para o modelo anotador “gabarito” criado pelo autor, para o desafio de linguagem natural).

É complicado comparar os dois sistemas de maneira “justa”, visto que a implantação em *bare-metal* obviamente é mais poderosa computacionalmente em comparação à arquitetura *serverless*. Entretanto, são discutidos na Seção 6.4 alguns contrapontos ao novo sistema, principalmente em relação a custos e complexidade de manutenção.

6.4 CUSTO E MANUTENÇÃO

No portal *Web* da IBM Cloud, a estimativa de custo mensal é 826 dólares para cada máquina bare-metal de 16 núcleos, 165 dólares para cada maquina virtual, e 2000 dólares de licença de uso da *Red Hat OpenShift* – totalizando 3981 dólares mensais de custo do sistema, sendo aproximadamente metade disso voltado para licenciamento (custo real inexistente à IBM, que desde 2019 é proprietária da Red Hat). Já para o sistema piloto, o custo total durante todo o evento foi de cerca de 340 dólares para o pagamento de computação *serverless*, distribuídos e um período de um mês e meio. Entretanto, mesmo com o custo reduzido do sistema piloto, também precisa ser contabilizado o custo para manutenção do portal *Web* da competição piloto, que foi realizado através de um contrato de cerca de 50 mil reais (aproximadamente 13 mil dólares) com uma empresa terceira, que incluía também ações de marketing e divulgação do evento.

Como é desconhecida a parcela delegada ao marketing e à manutenção do sistema, além da margem de lucro da empresa terceira, fica complicada a comparação entre os custos. Todavia, a partir dos dados disponíveis, o custo com o novo sistema é certamente mais caro porém provavelmente não muito superior, levando em conta que a hospedagem das páginas *Web* pode ser realizada juntamente nas mesmas máquinas provisionadas.

Um ponto a ser ressaltado, é que é possível provisionar um *cluster OpenShift* já instalado, diretamente na IBM Cloud, em vez da contratação e infraestrutura e instalação direta como foi realizado neste trabalho. A contratação de um *cluster OpenShift* varia com a capacidade do mesmo, mas é bem mais barata, no geral, do que o provisionamento e manutenção de 7 máquinas virtuais e três servidores *bare-metal*. Dessa forma, o gerenciamento do sistema todo ficaria também unificado em uma única plataforma, reduzindo a complexidade em detrimento de desempenho.

6.5 MÉTODO DE DESENVOLVIMENTO APLICADO

O método de desenvolvimento de *software* aplicado é avaliado pelo autor como essencial para trabalhos do tipo (onde segurança é primordial). A variação adotada do método *cleanroom* também se mostra superior quando a velocidade de lançamento é importante, visto que o método tradicional envolve a modelagem matemática do *software*, que para sistemas modernos é inviável. De fato, a melhor característica do método é o desacoplamento da equipe de desenvolvimento da equipe de teste. Embora esse desacoplamento não tenha sido realizado de início, após a aprovação da realização de um evento maior, o sistema passará por análises de segurança e testes por outras equipes, finalmente completando a aplicação do método de desenvolvimento proposto. Outras mudanças futuras são discutidas no próximo capítulo.

6.6 SOBRE OS OBJETIVOS ALCANÇADOS

Considerando todos os objetivos definidos no início do projeto, especificados na Seção 1.4.1, o projeto é avaliado como um sucesso.

Na Tabela 16 são apresentados os objetivos específicos e a situação de cada um ao final deste projeto de fim de curso.

Tabela 16 – Objetivos alcançados no projeto de fim de curso.

Objetivo	**Situação**
Formalização conforme ISO/IEC/IEEE 29148:2018	Completamente alcançado
Entrega do sistema de pontuação automática (desafio de processamento de linguagem)	Completamente alcançado
Entrega de sistema de análise de métricas	Completamente alcançado
Entrega de sistema completo para tratamento de inscrições	Parcialmente alcançado
Cumprimento com LGPDP e GDPR	Completamente alcançado
Implantação em ambiente de produção, integração e testes	Completamente alcançado
Execução de testes comparativos em ambiente de produção entre o novo sistema e o sistema piloto	Completamente alcançado

Fonte: Arquivo pessoal.

O único dos objetivos inicialmente planejados que não foi completamente alcançado foi o desenvolvimento completo de um sistema de tratamento de inscrições – não foi desenvolvida a interface de usuário (ou *front-end*) dessa aplicação.

Conforme justificado anteriormente, nesse tipo de evento público a empresa possui uma equipe de designers especializada na criação de interfaces, a fim de garantir a homogeneidade quando a marca da empresa é exposta. Por essa razão, foi decidido pelo próprio autor, e também recomendado pelo orientador na empresa, a derrubada desse objetivo. Em contrapartida, foram focados esforços no desenvolvimento do *back-end* para o tratamento das inscrições, que foi implantado e testado em ambiente de produção, alcançando resultados excelentes.

7 CONSIDERAÇÕES FINAIS

Durante as atividades de estágio na IBM dedicadas a este projeto de fim de curso, o autor entrou em contato com tecnologias de ponta variadas, muitas delas ainda em processo de amadurecimento. Foram trabalhados aspectos que vão desde a implantação até às fases de teste e avaliação dos sistemas desenvolvidos. Também foram muito bem trabalhados os aspectos de método de desenvolvimento e integração de sistemas de *software*, além da produção de documentação técnica, que permitiram o desacoplamento e a facilidade de integração entre os componentes do sistema – a definição rígida das interfaces de *software* são imprescindíveis para evitar erros e problemas futuros, e é um esforço valioso para a empresa onde as atividades do projeto de fim de curso foram realizadas.

O novo sistema desenvolvido, além de tudo, poderá continuar contando com o mesmo esquema de desenvolvimento baseado em equipes pequenas como antes, com a diferença de que o processo de implantação e operações (DevOps) será centralizado na administração de um *cluster* Red Hat OpenShift, enquanto que outros desenvolvedores poderão simplesmente focar no design de novos desafios para as eventuais competições futuras.

O sistema desenvolvido neste projeto passará por testes mais robustos de segurança, realizados por equipes apartadas do desenvolvimento. Além disso, o sistema também passará por algumas mudanças que foram recém-planejadas, como a alteração do banco de dados distribuído *Scylla* para uma das soluções proprietárias da IBM com criptografia pervasiva e ferramentas integradas de auditoria e anti-adulteração. É também prevista a transferência dos componentes implantados em servidores *bare-metal* para o *cluster OpenShift*, fazendo maior proveito das capacidades de gerenciamento e automatização desse complexo sistema criado pela Red Hat, e também centralizando toda a administração do sistema somente em uma única plataforma, com toda camada de infraestrutura gerenciada pela IBM Cloud.

No momento de publicação da versão final desta monografia, o autor já iniciou o trabalho de migração do Scylla para o *IBM Db2 on Cloud*, e a migração do *Kafka* oficial oferecido pela Fundação Apache para o *IBM Event Streams* (uma variante proprietária da IBM do mesmo projeto de código-aberto). Quanto às mudanças de qualidades do sistema, é previsto que pouco será alterado quanto ao desempenho do sistema, entretanto as capacidades de gerenciamento e segurança serão em muito aprimoradas, já que as soluções proprietárias da IBM já vêm com interfaces gráficas e documentação rica para uso interno.

REFERÊNCIAS

BITNER, Bill; GREENLEE, Susan. **z/VM – A Brief Review of Its 40 Year History**. 1. ed. [*S.l.*: *s.n.*], 2012. Disponível em: http://www.vm.ibm.com/vm40hist.pdf. Acesso em: 31 out. 2019.

BUYYA, Rajkumar; BROBERG; GOSCINSKI. **Cloud Computing Principles and Paradigms**. 1. ed. [*S.l.*: *s.n.*], 2010.

CERN. **ALICE Data Acquisition**. [*S.l.*], 2019. Disponível em: http://aliceinfo.cern.ch/Public/en/Chapter2/Chap2_DAQ.html. Acesso em: 31 out. 2019.

CNET. **IBM sells PC group to Lenovo**. [*S.l.*], 2004. Disponível em: https://www.cnet.com/news/ibm-sells-pc-group-to-lenovo/. Acesso em: 9 mar. 2020.

COMPUTERWORLD. **IBM exits semiconductor business and re-creates itself**. [*S.l.*], 2014. Disponível em: https://www.computerworld.com/article/2835940/ibm-exits-semiconductor-business-and-recreates-itself.html. Acesso em: 9 mar. 2020.

COMPUTERWORLD. **IBM to acquire PwC Consulting for $3.5 billion**. [*S.l.*], 2002. Disponível em: https://www.computerworld.com/article/2576700/ibm-to-acquire-pwc-consulting-for--3-5-billion.html. Acesso em: 9 mar. 2020.

ERIKSEN, Marius. **Your Server as a Function**. [*S.l.*], 2013. Disponível em: https://monkey.org/%5C%7Emarius/funsrv.pdf. Acesso em: 31 out. 2019.

FORBES. **IBM Buys Privately Held Softlayer For $ Billion**. [*S.l.*], 2019. Disponível em: https://www.forbes.com/sites/bruceupbin/2013/06/04/ibm-buys-privately-held-softlayer-for-2-billion/#3aeaa5a333a6. Acesso em: 9 mar. 2020.

FORTUNE. **IBM Tops 2018 Patent List as A.I. and Quantum Computing Gain Prominence**. [*S.l.*], 2019. Disponível em: https://fortune.com/2019/01/07/ibm-tops-2018-patent-list-as-ai-and-quantum-computing-gain-prominence/. Acesso em: 9 mar. 2020.

GOLDBERG, Robert P. **Architectural Principles for Virtual Computer Systems**. 1. ed. [*S.l.*: *s.n.*], 1973. Disponível em: https://apps.dtic.mil/dtic/tr/fulltext/u2/772809.pdf. Acesso em: 31 out. 2019.

GREULICH. **A View from Beneath the Dancing Elephant: Rediscovering IBM's Corporate Constitution**. 1. ed. [*S.l.*: *s.n.*], 2014.

HEWITT, Carl; BISHOP, Peter; STEIGER, Richard. **A Universal Modular Actor Formalism for Artificial Intelligence**. [*S.l.*], 1973.

HILL, Mark; MARTY, Michael. **Amdahl's Law in the Multicore Era**. 1. ed. [*S.l.*: *s.n.*], 2008. Disponível em: `https://research.cs.wisc.edu/multifacet/papers/tr1593_amdahl_multicore.pdf`. Acesso em: 18 jan. 2020.

LI, Mu *et al.* **Exploring Distributional Similarity Based Models for Query Spelling Correction**. 1. ed. [*S.l.*: *s.n.*], 2006.

LINGER, Richard; TRAMMELL, Carmen. **Cleanroom Software Engineering Reference**. Pittsburgh, PA, 1996. Disponível em: `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12635`.

LUKŠA, Marko. **Kubernetes in Action**. 1. ed. [*S.l.*]: Manning Publications, 2017.

MELL, Peter; GRANCE, Timothy. **The NIST Definition of Cloud Computing**. Gaithersburg, Maryland, 2011. Disponível em: `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf`. Acesso em: 31 out. 2019.

MILLS, Marlan; HEVNER, Alan. **Cleanroom Software Engineering**. 1. ed. [*S.l.*]: Institute of Electrical e Electronics Engineers, 1987. Disponível em: `https://ieeexplore.ieee.org/document/1695817`. Acesso em: 31 out. 2019.

NEWMAN, Sam. **Building Microservices: Designing Fine-Grained Systems**. 1. ed. [*S.l.*: *s.n.*], 2014.

REUTERS. **IBM closes $34 billion deal to buy Red Hat to boost cloud business**. [*S.l.*], 2019. Disponível em: `https://www.reuters.com/article/us-redhat-m-a-ibm-eu/ibm-closes-34-billion-deal-to-buy-red-hat-idUSKCN1U41DA`. Acesso em: 9 mar. 2020.

REZAEI, Arash; GUZ, Zvika; BALAKRISHNAN, Vijay. **ScyllaDB and Samsung NVMe SSDs Accelerate NoSQL Database Performance**. [*S.l.*], 2017. Disponível em: `http://tinyurl.com/msl-scylladb`. Acesso em: 31 out. 2019.

SCYLLADB INC. **Building the Real-Time Big Data Database**: Seven Design Principles behind Scylla. ScyllaDB Inc. 2018. Disponível em: `https://www.scylladb.com/wp-content/uploads/Scylla_Seven_Design_Principles.pdf`. Acesso em: 31 out. 2019.

SIMSON, Ernest von. **The Limits of Strategy: Lessons in Leadership from the Computer Industry**. 1. ed. [*S.l.*: *s.n.*], 2009.

SMITH, James E.; NAIR, Ravi. **Virtual Machines**: Versatile Platforms for Systems and Processes. 1. ed. [*S.l.*: *s.n.*], 2005.

STONEBRAKER, Michael. **A Case for Shared Nothing**. Berkeley, California, 1986. Disponível em: `http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf`. Acesso em: 31 out. 2019.

STROUSTRUP, Bjarne. **The Design and Evolution of C++**. 1. ed. [*S.l.*]: Addison-Wesley, 1994.

THE RUST REFERENCE. **Rust Documentation: The Rust Reference**. 2020. Disponível em: `https://doc.rust-lang.org/reference/index.html`. Acesso em: 13 jan. 2020.

VYUKOV Syzbot and the Tale of Thousand Kernel bugs. [*S.l.*: *s.n.*]. `https://www.youtube.com/watch?v=qrBVXxZDVQY`. Accessed: 2019-10-31.

WSJ. **IBM to Acquire SPSS, Adding to Acquisitions**. [*S.l.*], 2009. Disponível em: `https://www.wsj.com/articles/SB124878176796786611`. Acesso em: 9 mar. 2020.