INSTITUTO FEDERAL
Goiás
Câmpus Goiânia

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE GOIÁS

PRÓ-REITORIA DE ENSINO

CÂMPUS GOIÂNIA

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

GILBERTO VAMPRÉ GUIMARÃES LOBO

NATHAN SANTOS MARTINS

Desenvolvimento de um Serviço para Observabilidade de Infraestruturas Baseadas em Nuvem

Goiânia, 2020.

INSTITUTO FEDERAL
Goiás

MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
SISTEMA INTEGRADO DE BIBLIOTECAS

TERMO DE AUTORIZAÇÃO PARA DISPONIBILIZAÇÃO NO REPOSITÓRIO DIGITAL DO IFG - ReDi IFG

Com base no disposto na Lei Federal nº 9.610/98, AUTORIZO o Instituto Federal de Educação, Ciência e Tecnologia de Goiás, a disponibilizar gratuitamente o documento no Repositório Digital (ReDi IFG), sem ressarcimento de direitos autorais, conforme permissão assinada abaixo, em formato digital para fins de leitura, download e impressão, a título de divulgação da produção técnico-científica no IFG.

Identificação da Produção Técnico-Científica

[ ] Tese	[ ] Artigo Científico
[ ] Dissertação	[ ] Capítulo de Livro
[ ] Monografia – Especialização	[ ] Livro
[X] TCC - Graduação	[ ] Trabalho Apresentado em Evento
[ ] Produto Técnico e Educacional - Tipo: _____	

Nome Completo do Autor: Gilberto Vampré Guimarães Lobo
Matrícula: 2016101190259
Título do Trabalho: Desenvolvimento de um Serviço para Observabilidade de Infraestruturas Baseadas em Nuvem

Autorização - Marque uma das opções

1. (X) Autorizo disponibilizar meu trabalho no Repositório Digital do IFG (acesso aberto);
2. ( ) Autorizo disponibilizar meu trabalho no Repositório Digital do IFG somente após a data ___/___/_____ (Embargo);
3. ( ) Não autorizo disponibilizar meu trabalho no Repositório Digital do IFG (acesso restrito).

Ao indicar a opção **2 ou 3**, marque a justificativa:

( ) O documento está sujeito a registro de patente.
( ) O documento pode vir a ser publicado como livro, capítulo de livro ou artigo.
( ) Outra justificativa: _____

DECLARAÇÃO DE DISTRIBUIÇÃO NÃO-EXCLUSIVA

O/A referido/a autor/a declara que:

i. o documento é seu trabalho original, detém os direitos autorais da produção técnico-científica e não infringe os direitos de qualquer outra pessoa ou entidade;
ii. obteve autorização de quaisquer materiais inclusos no documento do qual não detém os direitos de autor/a, para conceder ao Instituto Federal de Educação, Ciência e Tecnologia de Goiás os direitos requeridos e que este material cujos direitos autorais são de terceiros, estão claramente identificados e reconhecidos no texto ou conteúdo do documento entregue;
iii. cumpriu quaisquer obrigações exigidas por contrato ou acordo, caso o documento entregue seja baseado em trabalho financiado ou apoiado por outra instituição que não o Instituto Federal de Educação, Ciência e Tecnologia de Goiás.

Goiânia, 16/12/22.
Local Data

Gilberto Vampré Guimarães Lobo
Assinatura do Autor e/ou Detentor dos Direitos Autorais

INSTITUTO FEDERAL
Goiás

MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
SISTEMA INTEGRADO DE BIBLIOTECAS

TERMO DE AUTORIZAÇÃO PARA DISPONIBILIZAÇÃO NO REPOSITÓRIO DIGITAL DO IFG - ReDi IFG

Com base no disposto na Lei Federal nº 9.610/98, AUTORIZO o Instituto Federal de Educação, Ciência e Tecnologia de Goiás, a disponibilizar gratuitamente o documento no Repositório Digital (ReDi IFG), sem ressarcimento de direitos autorais, conforme permissão assinada abaixo, em formato digital para fins de leitura, download e impressão, a título de divulgação da produção técnico-científica no IFG.

Identificação da Produção Técnico-Científica

[ ] Tese	[ ] Artigo Científico
[ ] Dissertação	[ ] Capítulo de Livro
[ ] Monografia – Especialização	[ ] Livro
[x] TCC - Graduação	[ ] Trabalho Apresentado em Evento

[ ] Produto Técnico e Educacional - Tipo: _____

Nome Completo do Autor: **Nathan Santos Martins**
Matrícula: **20161011090240**
Título do Trabalho: **Desenvolvimento de um Serviço para Observabilidade de Infraestruturas Baseadas em Nuvem**

Autorização - Marque uma das opções

1. (x) Autorizo disponibilizar meu trabalho no Repositório Digital do IFG (acesso aberto);
2. ( ) Autorizo disponibilizar meu trabalho no Repositório Digital do IFG somente após a data ___/___/_____ (Embargo);
3. ( ) Não autorizo disponibilizar meu trabalho no Repositório Digital do IFG (acesso restrito).

Ao indicar a opção **2 ou 3**, marque a justificativa:

( ) O documento está sujeito a registro de patente.
( ) O documento pode vir a ser publicado como livro, capítulo de livro ou artigo.
( ) Outra justificativa: _____

DECLARAÇÃO DE DISTRIBUIÇÃO NÃO-EXCLUSIVA

O/A referido/a autor/a declara que:

i. o documento é seu trabalho original, detém os direitos autorais da produção técnico-científica e não infringe os direitos de qualquer outra pessoa ou entidade;
ii. obteve autorização de quaisquer materiais inclusos no documento do qual não detém os direitos de autor/a, para conceder ao Instituto Federal de Educação, Ciência e Tecnologia de Goiás os direitos requeridos e que este material cujos direitos autorais são de terceiros, estão claramente identificados e reconhecidos no texto ou conteúdo do documento entregue;
iii. cumpriu quaisquer obrigações exigidas por contrato ou acordo, caso o documento entregue seja baseado em trabalho financiado ou apoiado por outra instituição que não o Instituto Federal de Educação, Ciência e Tecnologia de Goiás.

Goiânia, 16/12/22
Local Data

Nathan Martins
Assinatura do Autor e/ou Detentor dos Direitos Autorais

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE GOIÁS

PRÓ-REITORIA DE ENSINO

CÂMPUS GOIÂNIA

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

GILBERTO VAMPRÉ GUIMARÃES LOBO

NATHAN SANTOS MARTINS

Desenvolvimento de um Serviço para Observabilidade de Infraestruturas Baseadas em Nuvem

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Bacharelado em Sistemas de Informação como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Raphael de Aquino Gomes

Goiânia, 2020.

L7861d Lobo, Gilberto Vampré Guimarães.
Desenvolvimento de um serviço para observabilidade de infraestruturas baseadas em nuvem / Gilberto Vampré Guimarães Lobo, Nathan Santos Martins. – Goiânia: Instituto Federal de Educação, Ciência e Tecnologia de Goiás, Câmpus Goiânia, 2022.
57 f.: il.

Orientador: Prof. Dr. Raphael de Aquino Gomes.

TCC (Trabalho de Conclusão de Curso) – Bacharelado em Sistemas de Informação, Instituto Federal de Educação, Ciência e Tecnologia de Goiás, Câmpus Goiânia.

1. Computação em nuvem. 2. Internet das Coisas. I. Martins, Nathan Santos. II. Gomes, Raphael de Aquino (orientador). III. Instituto Federal de Educação, Ciência e Tecnologia de Goiás, Câmpus Goiânia. IV. Título.

CDD 004.6782

Ficha catalográfica elaborada pela Bibliotecária Karol Almeida da Silva CRB1/ 2.740
Biblioteca Professor Jorge Félix de Souza,
Instituto Federal de Educação, Ciência e Tecnologia de Goiás, Câmpus Goiânia.

MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE GOIÁS
CÂMPUS GOIÂNIA

GILBERTO VAMPRÉ GUIMARÃES LOBO

NATHAN SANTOS MARTINS

Desenvolvimento de um Serviço para Observabilidade de Infraestruturas Baseadas em Nuvem

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Sistemas de Informação do Instituto Federal de Educação, Ciência e Tecnologia de Goiás, como parte das exigências para obtenção do Título de Bacharel em Sistemas de Informação.

Goiânia, 08 de julho de 2022.

BANCA EXAMINADORA

Prof. Dr. Raphael de Aquino Gomes
Orientador - IFG/Câmpus Goiânia

Prof. Me. Renan Rodrigues de Oliveira
Membro Interno da Banca Examinadora - IFG/Câmpus Goiânia

Prof. Dr. Sirlon Diniz de Carvalho
Membro Interno da Banca Examinadora - IFG/Câmpus Goiânia

Dedicatória

Gostaria de dedicar este trabalho de conclusão de curso ao nosso professor de sistemas distribuídos, que tem sido uma fonte constante de conhecimento, orientação e apoio ao longo da nossa jornada acadêmica.

Ao nosso professor, obrigado por sua dedicação inabalável ao ensino e por compartilhar sua riqueza de conhecimento e experiência com seus alunos. Sua paixão pelo assunto e seu compromisso com nosso sucesso tem sido uma força motriz em nosso próprio aprendizado e crescimento.

Suas ideias e orientações nos ajudaram a entender melhor o mundo complexo e em constante evolução dos sistemas distribuídos, e somos gratos pela oportunidade de aprender com um educador tão talentoso e experiente.

Obrigado por ser um mentor inspirador e solidário e por nos ajudar a alcançar este marco importante na nossa carreira acadêmica. Somos gratos por tudo o que você fez e esperamos continuar a aprender e crescer sob sua orientação no futuro.

Agradecimentos

Gostaríamos de dedicar este trabalho de conclusão de curso às nossas famílias, amigos e professores que nos apoiaram e incentivaram ao longo da nossa jornada acadêmica.

Aos nossos pais, que sempre acreditaram em nós e nós deram o amor e a orientação de que nós precisávamos para ter sucesso. Obrigado por seu apoio inabalável e por sempre nós incentivar a ser o nosso melhor.

Aos nossos amigos, que estiveram ao nosso lado nas noites de estudo e nos momentos difíceis. Obrigado por serem uma fonte constante de motivação e por sempre levantar nossos ânimos.

Aos nossos professores, que nos desafiaram e nos inspiraram com seu conhecimento e experiência. Obrigado por compartilhar sua sabedoria e por nos ajudar a crescer como alunos e como pessoas.

Somos gratos a cada um de vocês por suas contribuições para o nosso sucesso. Essa conquista não teria sido possível sem o seu apoio. Obrigada.

A melhor maneira de prever o futuro é inventá-lo.

Alan Kay,
1940-?.

Resumo

Título: Desenvolvimento de um Serviço para Observabilidade de Infraestruturas Baseadas em Nuvem
Autores: Gilberto Vampré Guimarães Lobo e Nathan Santos Martins
Orientador: Dr. Raphael de Aquino Gomes

A expansão do alcance da tecnologia lança novos desafios aos desenvolvedores e administradores de sistemas. A solução de problemas por meio de ferramentas tecnológicas, das mais complexas às mais triviais, tem se tornado bastante comum. Além da proposta de valor, a infraestrutura por trás do software é crítica, pois ela viabiliza o pleno funcionamento do mesmo para que ele possa entregar valor. Portanto, observabilidade e monitoramento são conceitos chave para um software de sucesso. Entender os comportamentos de uma aplicação e agir proativamente antes que um problema em potencial aconteça são essenciais para se manter competitivo no mercado. Em ambientes complexos e distribuídos, muitas variáveis devem ser consideradas ao determinar a causa raiz de um problema. Atualmente, para utilizar as ferramentas de observabilidade disponíveis no mercado é necessário dedicar tempo e esforço para entender as bibliotecas e instrumentar o código-fonte. Também é essencial conhecer a engenharia de desempenho para poder interpretar as métricas e obter informações úteis. Diante dessas afirmações, esse trabalho visa fornecer uma ferramenta que auxilie a identificar possíveis gargalos e ineficiência na infraestrutura adjacente de um software, bem como sugerir melhorias a nível de máquina com base no que foi observado. A solução implementada se propõe a coletar métricas pre-estabelecidas, analisar as informações e trazer resultados que auxiliem o administrador do sistema a tomar decisões, com a finalidade de melhorar a sua infraestrutura.

Palavras–chave

Observabilidade, Computação em Nuvem, Desempenho

Abstract

Title: Service Proposal of a Cloud-Based Infrastructure Observability Platform
Authors: Gilberto Vampré Guimarães Lobo and Nathan Santos Martins
Advisor: Dr. Raphael de Aquino Gomes

The expanding reach of technology throws new challenges for developers and system administrators. Problem-solving through technological tools, from the most complex to the most trivial, has become common. In addition to the value proposition, the infrastructure behind the software is critical, as it enables it to function so that it can deliver value fully. Therefore, observability and monitoring are critical concepts for successful software. Understanding an application's behaviors and acting proactively before a potential problem happens is essential to staying competitive in the marketplace. In complex, distributed environments, many variables must be considered when determining the root cause of a problem. Currently, to use the observability tools available in the market, it is necessary to dedicate time and effort to understand the libraries and instrument the source code. It is also essential to know performance engineering to be able to interpret the metrics and obtain helpful information. Given these statements, this work aims to provide a tool that helps identify possible bottlenecks and inefficiencies in the adjacent software infrastructure and suggest improvements at the machine level based on the observed metrics. The implemented solution proposes to collect pre-established metrics, analyze the information and bring results that help the system administrator make decisions to improve its infrastructure.

Keywords

Observability, Cloud Computing, Performance

Lista de Figuras

Lista de Tabelas

Lista de Códigos de Programas

Lista de Abreviaturas e Siglas

API	*Application Programming Interface*
APIs	*Application Programming Interfaces*
AWS	*Amazon Web Services*
CI/CD	*Continuous Integration/Continuous Delivery*
eBPF	*Extended Berkeley Packet Filter*
HTTP	*Hypertext Protocol*
HTTP	*Hypertext Transfer Protocol*
I/O	*Input/Output*
IaaS	*Infrastructure as a Service*
IoT	*Internet of Things*
JSON	*JavaScript Object Notation*
MVC	*Model, View, Controller*
OOM	Out-Of-Memory
OOM	*Out of Memory*
REST	*Representational State Transfer*
SaaS	*Software as a Service*
SaaS	*Software as a Service*
SDK	*Software Development Kit*
USE	Usage, Saturation, Errors, do inglês, Uso, Saturação e Erros

Sumário

CAPÍTULO 1

Introdução

A Internet das Coisas ou, da sigla em inglês, *Internet of Things* (IoT) (LEE; LEE, 2015) permite que aplicações executem em praticamente qualquer dispositivo eletrônico com capacidade de se conectar à internet. Aparelhos que antes possuíam funções mais simples ou de usos específicos, como por exemplo relógios, telefones, televisões e geladeiras, atualmente possuem funcionalidades que podem ir muito além de suas funções primárias, ampliando os seus leques de utilidades.

Além das aplicações IoT que expandiram o alcance da tecnologia, existem muitas aplicações web disponíveis para diferentes dispositivos, sendo aplicações tão simples quanto *landing pages* e *blogs*, que dependem apenas de um servidor web para fornecer as páginas, até aplicações mais robustas do tipo *Software as a Service* (SaaS), que executam no navegador do cliente e se comunicam com um servidor remoto através de um protocolo de rede. O que essas aplicações de IoT e SaaS têm em comum, além de serem capazes de se conectar à internet, é que elas precisam se comunicar com a lógica de negócio (conhecido como *backend*) para funcionarem corretamente. Por isso, ações básicas como autenticação, operações de banco de dados como leituras e escritas, e processamento de ações do cliente são todas realizadas no *backend* da aplicação, que por sua vez pode ter a sua infraestrutura hospedada em um provedor de nuvem, em um servidor local ou em um *datacenter*.

Dada a necessidade de manter um servidor para hospedar o *backend* de aplicações, adquirir um servidor físico parece ser o próximo passo natural desse processo, mas é preciso levar em conta diversos fatores que tornam a aquisição mais complexa do que aparenta, pois existem aspectos subjacentes que devem ser considerados no processo de aquisição: o custo de aquisição dos componentes, o valor das peças, o custo da infraestrutura e as adequações necessárias para executar o software, pensando nos momentos de pico de uso sem prejudicar o desempenho. Com a depreciação do hardware, que se torna defasado e perde desempenho com o tempo, é preciso ter um plano a longo prazo, que estabelece como e quando ele será atualizado e também ter peças sobressalentes para casos de falha em componentes cruciais. A disponibilização e manutenção do espaço físico e custos com pessoal são outros fatores que devem ser levados em consideração. Devido a

estas questões, o modelo de computação em nuvem vem se tornando uma opção popular, tomando o espaço de *datacenters* e servidores locais por eliminar os problemas supracitados, além dos custos reduzidos e maior facilidade de aquisição ao comparado com a aquisição de hardware.

A adoção de infraestrutura de nuvem favorece o uso de tecnologia de contêineres uma vez que os provedores possuem diversos serviços adjacentes que suportam esse tipo de implantação. Em uma máquina virtual na nuvem é possível rodar diversos contêineres de uma mesma aplicação ou aplicações diferentes, o que gera um aproveitamento melhor dos recursos computacionais, e também viabiliza o escalonamento horizontal das aplicações caso surja a necessidade (BERNSTEIN, 2014).

A adoção de nuvem também influencia no processo de desenvolvimento de software uma vez que desenvolvedores buscam criar softwares funcionais e performáticos, que realizem suas funções com maior aproveitamento dos recursos computacionais, ao passo que proveem respostas rápidas aos seus usuários. Um ponto importante para alcançar esses objetivos é a observabilidade. Em sistemas que não possuem métricas de observabilidade implementadas, as falhas tendem a aparecer somente quando o cliente final percebe algum erro, instabilidade ou indisponibilidade do sistema, o que não é bom de uma perspectiva geral de negócio. Monitorar o funcionamento do sistema em tempo real, visualizar e manter histórico das taxas de erros, analisar o uso de recursos do sistema operacional e do hardware e configurar alertas de uso desses recursos são ações que contribuem para que seja possível agir antes de um possível problema se tornar crítico e ficar visível para o cliente final.

A observabilidade é também uma grande aliada na área de custos financeiros, especialmente em ambientes de nuvem que se paga pelo tempo utilizado. O uso desnecessário ou demasiado de recursos computacionais resultam em cobranças maiores. Observando o sistema é possível notar situações em que o hardware não é suficiente para rodar a aplicação ou o contrário, o hardware está ocioso. Ao observar um sistema é possível agir preventivamente antes de uma pane que poderia inutilizar um determinado sistema por um período de tempo, mitigando falhas e gastos desnecessários (NIEDERMAIER *et al.*, 2019). Ao empregar observabilidade as ações deixam de ser reativas e se tornam proativas, pois é possível monitorar os padrões de utilização de recursos do sistema e traçar planos mais detalhados para aumentar a resiliência do mesmo e identificar potenciais gargalos antes que aconteçam.

Mesmo diante da necessidade de observabilidade, as soluções existentes são de empresas privadas, envolvem custos de assinatura e de código fonte fechado. Entre as principais nessa categoria, há as plataformas NewRelic (2022) e Datadog (2022). Ambas as plataformas funcionam através de *agentes de coleta* que capturam dados e métricas das aplicações e apontam degradações nos serviços. Contudo, saber a devida utilização

desses serviços requer alterar e instrumentar o código da aplicação. Isso significa que a equipe de desenvolvimento precisa embutir no código fonte da aplicação o *drive* desses fornecedores e saber como instrumentá-los corretamente, o que exige prática. Cada linguagem de programação existente requer um *drive* próprio compatível com ela, fazendo com que o desenvolvedor ainda dependa da disponibilidade de um *drive* para a linguagem de programação utilizada no seu software. O uso desses fornecedores gera forte acoplamento no código e adiciona uma dependência externa no próprio cerne da aplicação, que em alguns casos pode gerar consequências indesejadas como deixar o código fonte e a manutenção mais complexos. Uma atualização do *drive* do fornecedor pode quebrar ou descontinuar alguma funcionalidade utilizada e fazer a aplicação falhar. Além disso, uma indisponibilidade do fornecedor pode afetar o desempenho ou uso de recursos da aplicação, levantando problemas que são alheios à aplicação em si mas que podem ser mascarados e difíceis de encontrar, levando a horas de investigação de um problema gerado por uma dependência que deveria auxiliar. Os agentes utilizados por essas soluções são componentes de software que precisam ser executados paralelamente à aplicação, o que consequentemente aumenta a sobrecarga dos servidores.

Outro trabalho relacionado é o Prometheus (CLOUD NATIVE COMPUTING FOUNDATION (CNCF), 2022), um projeto de código aberto que disponibiliza bibliotecas de software que os desenvolvedores precisam inserir em suas aplicações para instrumentá-la. Nesse caso, instrumentar se refere ao ato de medir determinadas ações realizadas pelo software, como o tempo de uma requisição *Hypertext Transfer Protocol* (HTTP) ou uma chamada a um banco de dados. Ao instrumentar o código, o Prometheus fica apto à coleta e análise de dados da aplicação, mas não faz sugestões de melhorias, representando apenas um coletor de métricas.

No cenário acadêmico há soluções semelhantes, como as propostas por Cassagnes *et al.* (2020) e Levin e Benson (2020). Ambos citam abordagens parecidas para observabilidade mas nenhum deles oferece um produto final funcional, sendo atividades de pesquisa sem objetivar o uso em produção.

Existem algumas razões pelas quais atualmente não temos muitas boas ferramentas de observabilidade no cenário acadêmico.

Primeiro, a observabilidade é um campo relativamente novo e em rápida evolução, e muitas das ferramentas e técnicas existentes ainda estão sendo desenvolvidas e refinadas. Como resultado, pode não haver muitas ferramentas de observabilidade maduras e bem estabelecidas disponíveis no cenário acadêmico.

Em segundo lugar, a observabilidade pode ser um campo complexo e desafiador para estudar e pesquisar, e pode exigir conhecimento especializado e experiência que não está amplamente disponível na comunidade acadêmica. Isso pode dificultar o desenvolvimento de ferramentas eficazes de observabilidade pelos pesquisadores e limitar a

disponibilidade de tais ferramentas no cenário acadêmico.

Em terceiro lugar, a comunidade acadêmica geralmente se concentra na realização de pesquisas fundamentais e no desenvolvimento de novas teorias e ideias, em vez de desenvolver ferramentas e aplicações práticas. Como resultado, pode haver menos ênfase no desenvolvimento de ferramentas de observabilidade no cenário acadêmico, em comparação com outras áreas da ciência da computação e engenharia.

No geral, esses fatores podem contribuir para a atual falta de boas ferramentas de observabilidade no cenário acadêmico.

Pensando em ambientes de infraestrutura em nuvem, em conjunto com tecnologias de conteinerização, neste trabalho é proposto o desenvolvimento de uma plataforma de observabilidade que irá monitorar o uso de recursos computacionais da aplicação em contêineres. Através desta plataforma, informações do sistema observado são coletadas, persistidas e analisadas em que as métricas (baseadas em *Extended Berkeley Packet Filter* (eBPF) (BEGEL; MCCANNE; GRAHAM, 1999)) serão co-relacionadas. Como resultado da análise, a plataforma oferece sugestões de melhorias a nível de recursos de nuvem. A plataforma também será capaz de apresentar as métricas e sugestões oferecidas, a fim de apontar gargalos e sugestões de solução. Todo o monitoramento do sistema é possível sem causar sobrecarga aos sistemas rodando no contêiner.

1.1 Objetivos

O objetivo geral deste trabalho é desenvolver uma ferramenta de observabilidade, monitoramento e sugestão de infraestrutura. Para alcançar este objetivo, são estabelecidos os seguintes objetivos específicos:

- Implementar agentes coletores de métricas;
- Prover uma interface de monitoramento;
- Implementar uma API de sugestão;
- Disponibilizar um sistema gerador de gráficos.

A ferramenta como um todo deverá coletar e persistir dados de uso de recursos computacionais em ambiente de nuvem utilizando contêineres. A aplicação também deve ser capaz de fornecer sugestões de possíveis melhorias a nível de infraestrutura de nuvem, sugerindo máquinas com melhor custo benefício baseado no padrão de uso observado, junto com gráficos que demonstram tais métricas.

1.2 Estrutura do Trabalho

O restante do trabalho é organizado da seguinte forma:

O Capítulo 2 apresenta os principais conceitos relacionados ao desenvolvimento da plataforma proposta. Mais precisamente, são apresentadas definições de computação em nuvem, redução de custos, núcleo de sistema operacional, conteinerização, observabilidade e monitoramento.

O Capítulo 3 descreve a arquitetura proposta para alcançar os objetivos definidos, quais componentes serão desenvolvidos, como eles estarão organizados, como eles se relacionam entre si e seus papéis dentro do ecossistema.

O Capítulo 4 aborda a implementação do sistema baseado na arquitetura descrita no capítulo anterior. São apresentadas a implementação proposta, os resultados e a validação da solução.

O Capítulo 5 apresenta as considerações finais, os objetivos que foram alcançados, desafios, impedimentos e pontos de melhoria para futuros trabalhos.

CAPÍTULO 2

Referencial Teórico

Nesse Capítulo serão abordados os principais conceitos que servem de fundamentação teórica. São conceitos motivadores para a existência deste trabalho e importantes para o entendimento do mesmo, fornecendo uma visão geral da área de desenvolvimento em nuvem, em que esta solução é focada, bem como conceitos adjacentes e igualmente importantes.

2.1 Computação em Nuvem

Quando se fala em computação em nuvem se refere a uma tendência de mercado (PEREIRA *et al.*, 2017) que vem ganhando tração desde 2006 e nos últimos anos se tornou uma das principais formas de implantar e operacionalizar softwares modernos. Isso se deve a suas inúmeras vantagens, dentre as quais pode ser citado o custo reduzido em relação a aquisição de hardware, a facilidade na contratação dos serviços, a delegação das atividades de manutenção do hardware, a alta disponibilidade dos serviços e a facilidade para implementar entrega ágil de software (AVRAM, 2014).

Os provedores disponíveis no mercado oferecem um conjunto de serviços com um leque extenso de funcionalidades, englobando aquelas até então disponíveis apenas em servidores ou *datacenters* locais. Dentre esses produtos e funcionalidades o trabalho aqui apresentado foca na categoria *Infrastructure as a Service* (IaaS), ou em português Infraestrutura como Serviço; mais especificamente em máquinas virtuais utilizando contêineres.

As máquinas virtuais são um dos principais serviços oferecidos na categoria IaaS, representando a base para hospedagem de aplicações. Isso se deve ao fato deste serviço ser altamente configurável, sendo possível que o cliente selecione dentro de um conjunto pré-estabelecido de configurações (que representam capacidades de CPU, memória RAM, armazenamento, rede, além de aspectos de software como sistema operacional), aquela mais adequada às suas necessidades. A granularidade na seleção das configurações varia dependendo do provedor, sendo que alguns apresentam as máquinas virtuais em tipos com capacidade fixa enquanto outros permitem um ajuste fino em cada aspecto

da configuração. Dos provedores que separam por tipos, cada tipo serve um propósito de uso específico, por exemplo, alta carga de processamento, maior armazenamento ou uso comum.

Em casos de alocação de máquina virtual em conjunto com tecnologias de conteinerização, é possível que uma mesma máquina virtual abrigue múltiplos contêineres, tendo um aproveitamento melhor dos recursos computacionais da máquina virtual e permitindo executar múltiplas aplicações independentes umas das outras em uma mesma máquina virtual.

Independente do caso de uso, os provedores de nuvem oferecem múltiplos modelos de contratação dos seus serviços variando o custo, nível de comprometimento no uso e disponibilidade. Contudo, todos eles têm métodos de cobrança parecidos ou análogos:

- **reserva**: é a contratação do serviço por um período de tempo que geralmente varia de 1 a 3 anos. É cobrado um valor fixo independente do uso da máquina ou da carga de trabalho, que varia de acordo com o tipo ou configuração da máquina virtual e a periodicidade de contratação.
- **sob demanda (*on-demand*)**: é a utilização sem um contrato prévio, sendo a cobrança realizada em função do tempo que a máquina virtual está ativa.
- ***spot***: por vezes os provedores podem ter hardware ocioso, o que representa desperdício de recursos. Como forma de ocupar essa ociosidade, os provedores oferecem este modelo de contratação, que são instâncias oferecidas com alto desconto para incentivar a ocupação do hardware ocioso. Os preços desse modelo variam conforme a demanda, sendo que as instâncias só executam quando existe hardware ocioso, ou seja, caso a demanda do provedor aumente a instância pode ter a sua execução interrompida.

Independente do modelo de cobrança adotado, o desempenho e, consequentemente, a observabilidade se tornam elementos cruciais para maximizar o uso das máquinas virtuais. No modelo reserva, por exemplo, é interessante fazer o uso total da máquina virtual. Neste caso, a observabilidade se torna essencial para monitorar o uso de recursos da aplicação e, assim, otimizar a quantidade de contêineres que podem compartilhar a mesma máquina virtual. No caso de adoção de um modelo *on-demand*, também é de interesse do usuário monitorar o desempenho, pois paga-se pelo tempo computacional utilizado. Isso quer dizer que em quanto menos tempo a aplicação conseguir completar sua tarefa, menos gasto será necessário. Para tal, a observabilidade pode ser uma forma de identificar pontos de melhoria para que a aplicação complete um número maior de tarefas em menor tempo.

Além de permitir formas variadas de cobrança, a computação em nuvem favorece a automação de tarefas de implantação, oferecendo um conjunto de ferramentas de

Continuous Integration/Continuous Delivery (CI/CD) (BECK, 2000; HUMBLE; FARLEY, 2010) que permitem a entrega rápida, automatizada e contínua de software. Usando essas ferramentas é possível definir uma esteira de execução, de forma que sempre que uma implantação for disparada um conjunto de passos definidos é executado. Isso torna o processo de entrega de software mais rápido, confiável e totalmente automatizado (SINGH *et al.*, 2019).

2.1.1 Redução de custos

Ao optar por adquirir hardware é preciso saber, ou ter uma estimativa, do poder computacional necessário para rodar a aplicação sem gerar gargalos no desempenho ou indisponibilidade do sistema. Informação que nem sempre se tem no momento da aquisição do hardware e que pode oscilar devido a eventuais picos de utilização ou sazonalidade. Dado que existem muitas aplicações que são projetadas para a web, em que qualquer pessoa com acesso a internet pode acessá-las, o desafio de estimar a média de usuários que utilizarão o sistema simultaneamente, para então estimar um hardware adequado para suportar tal base de usuários, se torna ainda mais desafiador.

O modelo de computação em nuvem permite instanciar uma máquina virtual de maneira descomplicada, sendo muito fácil fazer os ajustes necessários dada a elasticidade. Nesse cenário se torna simples prover a infraestrutura de uma aplicação com baixo investimento ou até mesmo nenhum, pois a maioria dos provedores de nuvem oferecem um *free-tier*, classe de serviços com configurações mais básicas, gratuitos que podem ser suficientes em alguns cenários, como por exemplo para validar uma hipótese ou que muitas vezes chamamos de *MVP*, produto minimamente viável. (BHARDWAJ; JAIN, 2010).

Um produto minimamente viável é um produto ou serviço que foi desenvolvido com o número mínimo de recursos necessários para coletar *feedback* valioso dos primeiros clientes. Isso permite que as empresas validem rapidamente sua ideia de produto e coletem dados sobre seu potencial sucesso, sem investir muito tempo e recursos em um produto completo.

A nuvem é uma boa plataforma para validar um MVP porque permite que as empresas implantem seu produto ou serviço de forma rápida e fácil para um público amplo, sem ter que investir em infraestrutura e recursos caros. Isso torna mais fácil para as empresas testar seu MVP e coletar feedback dos clientes, que podem ser usados para refinar e melhorar o produto antes de lançar uma versão completa.

Além disso, a nuvem oferece uma variedade de ferramentas e serviços que podem ser usados para coletar dados e *insights* sobre o comportamento do cliente e padrões de uso, o que pode ajudar as empresas a entender melhor seu mercado-alvo e tomar decisões mais informadas sobre o desenvolvimento futuro de seus produtos.

No geral, a nuvem oferece às empresas uma maneira flexível, escalável e econômica de validar seu MVP e coletar feedback valioso dos clientes, o que pode ajudar a melhorar o produto final e aumentar suas chances de sucesso.

Outro ponto de otimização de custos ao adotar o modelo de computação em nuvem é a redução dos custos de mão de obra, dado que um único profissional consegue gerenciar uma parte significativa da infraestrutura. Além disso, o uso de esteiras de implantação tornam grande parte do processo totalmente automatizado, naturalmente diminuindo a necessidade de intervenção humana.

O cliente também não precisa se preocupar com a defasagem natural do hardware uma vez que toda especificação de instância é realizada em função da capacidade computacional, não sendo ligada a uma implementação específica de hardware.

2.1.2 Conteinerização

Contêineres fora do contexto computacional são caixas enormes de metal, como mostrado na Figura 2.1 e são usadas para transportar múltiplos bens ao redor do mundo. Embora possam ter o conteúdo heterogêneo, um contêiner é tratado como uma única unidade de transporte, sendo a única restrição que o mesmo caiba dentro do espaço delimitado pelo mesmo.

Figura 2.1: *Contêineres usados no mundo real.*

Fonte: (BRASIL IMPORTEX, 2022)

Quando se fala em software, contêineres são similares em conceito pois eles fornecem um ambiente com delimitações claras, que podem conter quaisquer tipos de softwares dentro desse espaço delimitado. São projetados para comportar softwares em

qualquer sistema adjacente[1], funcionando como uma única unidade e mantendo um ambiente controlado. Diferente, por exemplo, de uma máquina virtual que possui um *hipervisor* intermediando as interações entre a máquina virtual e o hospedeiro, conforme ilustrado na figura 2.2.

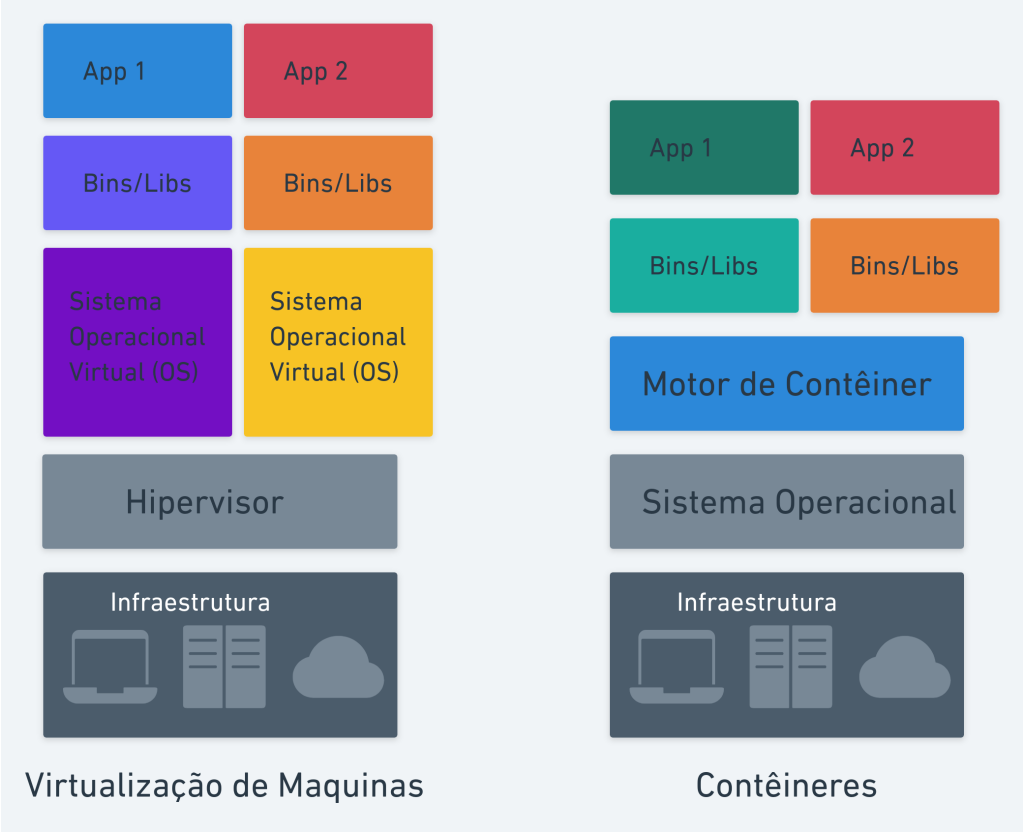

Figura 2.2: *Contêiner vs. Máquina Virtual.*

Fonte: Adaptado de (DELGADO, 2020)

Os limites de um contêiner são definidos pelo próprio usuário no momento de sua declaração. Por limites se entende os recursos computacionais que estarão disponíveis para o mesmo quando executado, como o sistema operacional, memória RAM, CPU e disco. Como cada contêiner possui um espaço delimitado e isolado a seu dispor, isso proporciona indiretamente uma outra vantagem: é possível executar aplicações que tenham as mesmas dependências mas em versões diferentes no mesmo sistema adjacente. Por exemplo, um contêiner que foi projetado para executar uma aplicação em Python 2 e outro que executa uma aplicação em Python 2.2 podem coexistir na mesma máquina sem qualquer tipo de conflito, graças à separação dos espaços entre um e outro.

Com contêineres é possível empacotar softwares e suas dependências, movê-los de lugar como uma única unidade, e executá-los em qualquer ambiente que suporte o

[1]Por sistema adjacente neste contexto se entende um servidor físico ou uma máquina virtual em um servidor de nuvem

uso da tecnologia, sendo o ambiente de computação em nuvem um dos suportes mais adotados.

Contêineres podem ser implementados usando várias tecnologias. Porém, a mais recorrente e comum utilizada na indústria são contêineres baseados em cGroups (MENAGE, 2022) a nível do núcleo Linux, implementado pela empresa norte-americana Docker (MARKETSANDMARKETS, 2022). Trata-se de uma forma de conteinerização utilizando isolamento e virtualização de processos, de forma que o contêiner consegue se isolar do resto do sistema operacional e é executado como um sistema quase independente. A quase independência se deve ao fato dele depender de uma infraestrutura para hospedá-lo, mas em termos de execução e do que acontece dentro do contêiner ele é uma unidade independente. É também através deste isolamento que se torna possível forçar limites e restrições de recursos computacionais. Com isso, o software sendo executado dentro de um contêiner não consegue utilizar mais recursos do que foi alocado ao contêiner em sua concepção.

2.2 Núcleo de Sistemas Operacionais

Conforme ilustrado na Figura 2.3, o núcleo (*kernel*) é um componente de software responsável por fazer a gestão do hardware. Quando os sistemas operacionais são analisados em forma de camadas, o núcleo é a camada mais baixa de todo o sistema. Suas responsabilidades se estendem em gestão de entrada e saída - *Input/Output* (I/O) - e controle da CPU e memória, entre os demais componentes físicos que compõem um sistema de computador.

Dentro do núcleo há dois modos em que softwares podem executar, o chamado *kernel mode* e *user mode*. Em *kernel mode*, os softwares possuem acesso direto ao hardware e sem restrições para acesso aos recursos da máquina. Já em *user mode*, os programas precisam comunicar com o núcleo através da sua *Application Programming Interface* (API), que geralmente consiste em uma ou várias bibliotecas que nada mais são que chamadas de métodos/funções que os desenvolvedores do núcleo disponibilizam para os usuários.

Neste trabalho quando é citado o núcleo, este sempre se refere ao núcleo monolítico Linux, em especifico núcleos a partir da versão 3.18, lançada em dezembro de 2014 (KERNELNEWBIES, 2014), pois esta foi a data em que foram adicionados recursos chave para a execução deste trabalho, mais especificamente capacidades de observabilidade (baseadas em eBPF). O núcleo Linux é um software de código-fonte aberto e de livre uso.

Vale ressaltar que o núcleo em si não é um sistema operacional, é apenas um componente entre vários, que juntos compõem o que no ecossistema Linux é chamado de

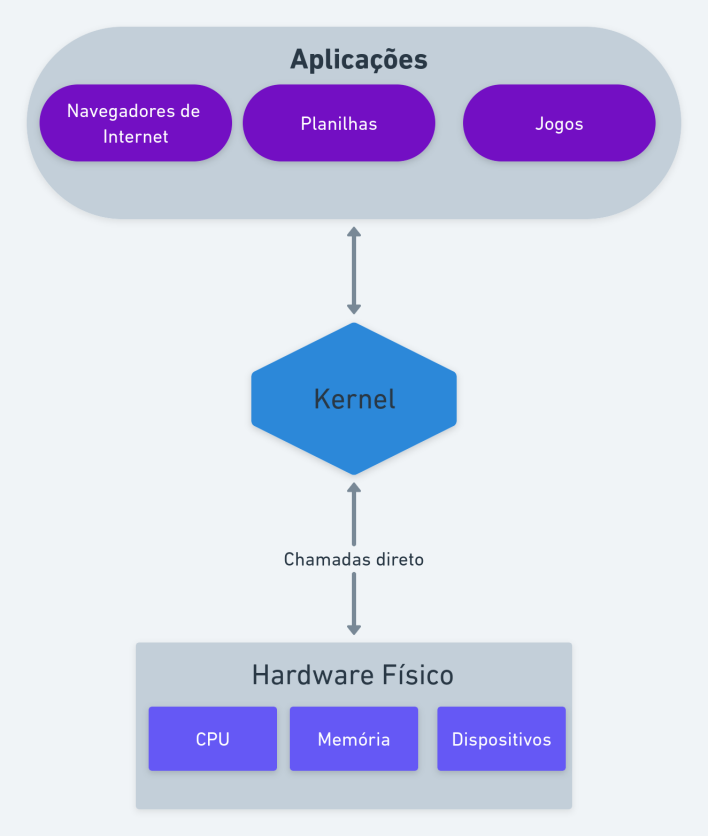

Figura 2.3: *Arquitetura de um kernel monolítico.*

Fonte: Elaborada pelos autores

distribuição, como Ubuntu (UBUNTU, 2022), Debian (INTEREST, 2022) e RedHat (HAT, 2022).

2.3 Observabilidade

A grosso modo observabilidade é fazer com que o sistema seja observável, ou seja, dar visibilidade ao que acontece internamente, fornecendo dados em tempo real e histórico dos eventos. Muitas vezes ao desenvolver sistemas, especialmente sistemas altamente distribuídos em ambientes de nuvem, é impossível saber de antemão todos os possíveis problemas que podem acontecer na execução. Dentre estes, o desempenho insatisfatório é muito comum, sendo que o administrador do sistema precisa descobrir o gargalo e resolvê-lo. Sem nenhum dado sobre o estado do ambiente e execução do sistema seu trabalho de investigação vai levar muito tempo e mesmo que ele consiga executar o sistema em ambiente local para tentar reproduzir o problema e executar alguns testes, o simples fato do ambiente ser diferente já faz com que os testes gerem resultados inválidos.

Dado as questões levantadas, a observabilidade adquire um papel de fundamental importância pois fornece aos administradores uma janela direta ao que está acontecendo

com o sistema em um determinado momento no tempo, fornecendo informações vitais como uso de CPU, uso de memória RAM, taxa de erros, uso de disco, uso de rede, tempo de resposta, quais chamadas do sistema foram executadas, quais funções da aplicação foram chamadas quantas vezes, o tempo de execução, e muitos outros.

A observabilidade dá o ferramental necessário para aliviar a tarefa de investigação, fazendo com que encontrar e resolver problemas se tornem muito mais fáceis. Com toda essa massa de dados sendo coletada, é possível transformá-la em informação ao criar gráficos baseados em métricas personalizadas, consultar dados históricos do estado do sistema, traçar padrões de comportamentos específicos, identificar acontecimentos intermitentes ou tão raros que de outra maneira seriam impossíveis de detectar.

2.3.1 Monitoramento

Monitoramento anda em conjunto com observabilidade, uma vez que são conceitos que se complementam por natureza. Embora observabilidade traga claras vantagens ao negócio e ao sistema, é o monitoramento que torna um sistema observável (SERVICES, 2022).

O monitoramento, diferente da observabilidade, vigia o comportamento do sistema e alerta os administradores caso algum cenário pré definido aconteça. Isso envolve conhecer o sistema e possíveis falhas. Em aplicações de servidores web, por exemplo, um ponto comum de atenção é o tempo de resposta do servidor, dado que atender o cliente o mais rápido possível é objetivo na maioria das aplicações dessa classe. Então é possível, tendo observabilidade instrumentada para a aplicação, monitorar o tempo de resposta de um servidor web e emitir um alerta caso a medida ultrapasse um valor determinado em um espaço de tempo.

Com um monitoramento bem definido e configurado os administradores de sistemas não precisam olhar gráficos periodicamente para saber que uma falha crítica está acontecendo no sistema. Ele será alertado ativamente pela ferramenta de monitoramento escolhida e poderá atuar ativamente em cima da falha utilizando dados de observabilidade para entender o que aconteceu.

Colocando ambos os conceitos para trabalharem juntos e aplicando estratégias de melhorias, o desperdício de recursos computacionais e indisponibilidade parcial ou total de sistemas é reduzido drasticamente.

Tabela 2.1: *Diferenças entre Observabilidade e Monitoramento*

Observabilidade	Monitoramento
O quê o meu sistema está fazendo?	O meu sistema está funcionando?
Te diz porquê algo deu errado	Te diz quando algo deu errado
Proativo por natureza	Reativo por natureza
Reduz a duração e o impacto de incidentes	Permite resposta rápida quando um incidente acontece
Foco a saúde da rede do ponto de vista do usuário final	Foco direcionado a saúde do dispositivo de rede
Mostra o número de requisições atendidas por minuto (RPM)	Alerta caso as requisições demorem a serem completadas
Salva histórico da quantidade e quais erros aconteceram	Alerta caso de alta taxa de erros
Lista as transações mais lentas	Alerta caso uma taxa de transações lentas seja alcançada
Reduz tempo para encontrar a causa raíz e solucionar o problema	Precisa lidar manualmente com o alerta

Fonte: (INC., 2020) (PDFMANUALS, 2021)

CAPÍTULO 3

Projeto Arquitetural

Nesse capítulo é apresentado a arquitetura proposta para a solução como um todo, demonstrando a natureza distribuída do sistema, e a arquitetura de cada microsserviço individualmente. Será abordado a organização dos componentes que constituem a aplicação, seus métodos de comunicação, os possíveis fluxos da informação e as motivações para as decisões arquiteturais.

3.1 Arquitetura

O projeto do sistema utiliza uma arquitetura *cloud-native*, ou seja, pensada desde o início para ser executada em um ambiente de nuvem. Esta é definida como um sistema distribuído e é composta pelos componentes a seguir, conforme ilustrado na Figura 3.1:

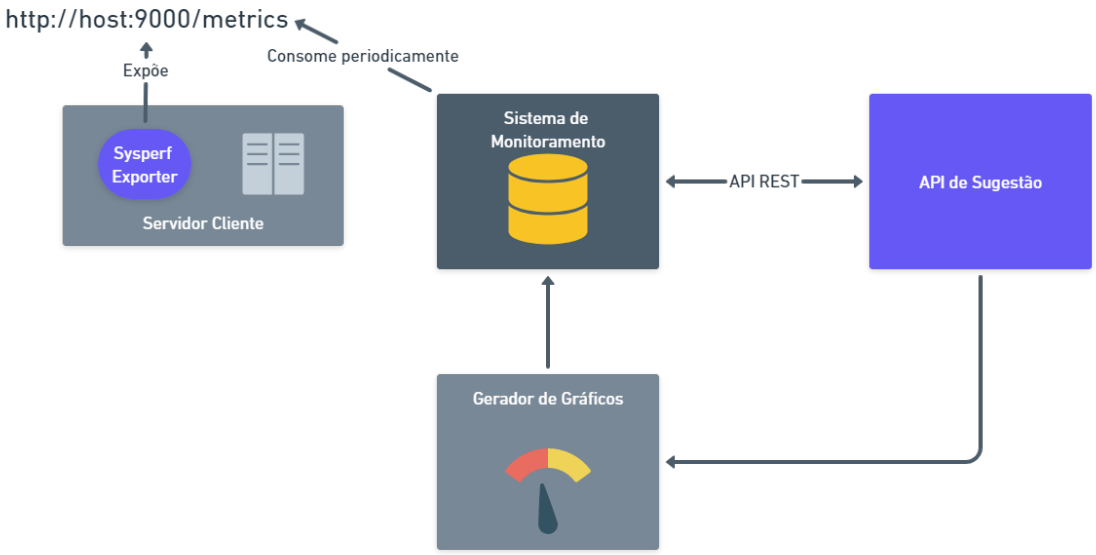

Figura 3.1: *Arquitetura proposta para o sistema.*

Fonte: Elaborada pelos autores

Dentre os componentes arquiteturais, em sua concepção é preciso atender aos seguintes requisitos:

- Desenvolver um software estilo cliente capaz de rodar em sistemas *Linux x86_64*;
  - Este cliente deverá ser auto configurável, com o mínimo de intervenção externa possível;
  - Este cliente deverá rodar com o mínimo de sobrecarga possível para evitar o *efeito do observador* e coletar dados;
- Desenvolver um software de servidor capaz de receber esses dados;
- Desenvolver um software de servidor capaz de gerar gráficos;
- Desenvolver um software de servidor capaz de gerar sugestões de maquinas de nuvem com base em heurísticas conhecidas.

A arquitetura tal qual foi planejada viabiliza o desenvolvimento e implantação de cada um dos seus componentes de maneira desacoplada e independentes uns dos outros, se comportando como peças autônomas dentro do ecossistema. Ainda existe um certo grau de dependência, pois um componente que depende de alguma operação fornecida por outrem continua dependendo dessa operação, mas essa dependência deixa de ser na implementação da operação e passa a ser na interface de comunicação.

A comunicação entre as partes são definidas por *Application Programming Interfaces* (APIs) com contratos de entrada e saída de dados bem definidas. Cada elemento possui a sua própria interface, expondo as requisições que ele pode atender e os dados resultantes, sendo que a implementação da interface é independente dos demais componentes, não importando como a mesma é implementada para que os integradores saibam como se comunicarem. Dessa forma, não importa como os componentes trabalham internamente, desde que a entrada e saída de dados permaneçam com os mesmos moldes. Consequentemente, seguindo essas diretrizes, foram desenvolvidos componentes permutáveis e um ecossistema com baixo acoplamento, gerando maior flexibilidade entre os seus membros, desde que os possíveis substitutos implementem a interface de comunicação definida. Por exemplo, não importa como a API de Sugestão é implementada, se ela utiliza *Golang* ou *Python* como linguagem de programação, se ela utiliza arquitetura Hexagonal (GRIFFIN, 2021) ou *Model, View, Controller* (MVC) (MAJEED; RAUF, 2018), desde que ela continue implementando as interfaces que o sistema de monitoramento utiliza, a comunicação entre as duas partes não sofrerá nenhum prejuízo.

Em uma arquitetura de sistemas distribuídos se tem maior flexibilidade no momento de desenvolvimento, evolução da solução e entrega. Como é suficiente ter os contratos de comunicação bem definidos nas *APIs*, cada desenvolvedor pode trabalhar na implementação de um serviço independente do outro, acelerando o processo de desenvolvimento e melhorando a distribuição de tarefas e evolução.

Após o desenvolvimento, a implantação de cada componente também pode, e é realizada, independente uma da outra, cada um com a sua esteira de *CI/CD*, reduzindo

o tempo de inatividade do sistema como um todo, já que cada módulo fica indisponível somente pelo tempo que está acontecendo a implantação. Dessa maneira, é possível fazer uma implantação, por exemplo, do componente Gerador de Gráficos sem que a API de Sugestões fique totalmente indisponível. Ela ainda será capaz de executar grande parte das suas funções durante o período de tempo da implantação do Gerador de Gráficos, como sugerir melhorias, sem que o sistema fique totalmente indisponível.

Por se tratar de sistemas menores, o próprio tempo de implantação é drasticamente reduzido, o que consequentemente também reduz o tempo de qualquer indisponibilidade parcial durante a implantação.

Essa arquitetura provê um fluxo de trabalho que viabiliza entregas menores e constantes, já que cada microsserviço possui um escopo limitado e especializado, e a implantação possui impacto reduzido. É possível evoluir as funcionalidades internas de cada componente do sistema sem causar impacto aos demais componentes, contanto que sempre mantenha a consistência dos contratos. Também é possível fazer a implantação sem afetar o sistema como um todo, ou afetando minimamente. Em contrapartida a uma arquitetura monolítica, por exemplo, em que a adição ou alteração de componentes pode ter um impacto maior devido ao alto acoplamento entre seus componentes internos, ou, mesmo em sistemas monolíticos que conseguem atingir baixos níveis de acoplamento, por vezes a complexidade do sistema é tão grande quanto a quantidade de código escrita, desacelerando o ritmo de desenvolvimento e adição de novas funcionalidades.

Ainda utilizando o exemplo de sistemas monolíticos, a implantação de novas funcionalidades por vezes gera um tempo de indisponibilidade do sistema como um todo, já que todas as funcionalidades do sistemas funcionam como um bloco conjunto. Por vezes, para mitigar a indisponibilidade total do sistema é necessário empregar táticas mais complexas e custosas como *Blue/Green deployment* (FOWLER, 2010). Como há blocos de funcionalidades bem distintos na solução proposta, foi possível separá-los em domínios menores e mais especializados sem dificuldades.

3.1.1 Agente coletor

O fluxo das informações começa no agente coletor, chamado de *Sysperf Exporter*. Ele é o componente responsável pela coleta dos dados que vão alimentar as métricas propostas. O agente coletor é um binário executado no hospedeiro da aplicação a ser monitorada como, por exemplo, um servidor. Ele executa em plano de fundo enquanto o ciclo de vida do hospedeiro continuar ativo. Sua função primária é coletar os dados de uso dos recursos computacionais do sistema como memória RAM, CPU e disco. A coleta é feita observando as chamadas ao *kernel* realizadas por processos do usuário. Além disso, ele deve transformar esses dados para um formato mais legível para serem coletados pelo

sistema de monitoramento.

À medida que os dados vão sendo coletados, os agentes coletores fazem a transformação dos mesmos e vão alocando-os em lotes. Antes de serem alocados ao lote, os dados já devem estar estruturados no formato final em que serão persistidos no sistema de monitoramento. Assim, toda e qualquer lógica intrínseca a algum dos dados relativos às métrica são de responsabilidade dos agentes coletores e ele deve definir quais campos serão salvos. Por exemplo, ao coletar as métricas de uso de memória RAM, o agente coletor deve ser capaz de ler a fonte fornecedora dos dados relativos a essa métrica, captar qual o comando que originou esse uso e a quantidade de memória RAM usada pelo mesmo; e a partir desses dados transformá-los para uma estrutura de dados reconhecível pelo Sistema de Monitoramento, para que seja adicionado ao lote e posteriormente capturado.

3.1.2 Sistema de monitoramento

O lote de dados fica à disposição nos Agentes Coletores, em memória, para o sistema de monitoramento realizar a coleta em um intervalo de tempo preestabelecido. Toda a comunicação do repasse dos dados entre os agentes coletores e o sistema de monitoramento é realizada utilizando o protocolo de comunicação *Hypertext Protocol* (HTTP).

Os agentes coletores expõem uma *pull* API (CAO *et al.*, 2016) com uma rota conhecida ao sistema de monitoramento. Dessa forma, ao fazer uma requisição nessa rota conhecida o sistema de monitoramento recebe o lote de dados coletados até o momento. O intervalo de coleta dos dados foi definido em cinco segundos, sendo que esse valor é configurável por um arquivo de configuração no momento da implantação, ou em tempo de execução pela *API*. Esse intervalo foi definido para mitigar o impacto no sistema hospedeiro do Agente Coletor. Por se tratar de um programa que roda no sistema hospedeiro ele naturalmente utiliza uma quantidade de recursos computacionais, de forma que o objetivo é minimizar esse uso. Além disso, o intervalo foi definido de forma a não gerar acúmulo de dados em memória nos Agentes Coletores e acabar gerando uma sobrecarga, ao passo que também não sejam feitas requisições constantes a ponto de utilizar muitos recursos de I/O e onerar a CPU do sistema hospedeiro.

3.1.3 API de sugestão

A API de Sugestão é uma API Web, um microsserviço, ela faz a ingestão dos dados históricos de métricas persistidos no sistema de monitoramento e instrumenta as tarefas de análise dos mesmos. A API tem duas funcionalidades, sendo elas o motor de experimentos e hipóteses e a provisão das métricas coletadas.

A função primária da API é, mediante requisição do cliente, realizar uma série de experimentos com os dados de uso coletados em uma janela de tempo e, a partir dos resultados desses experimentos, elaborar hipóteses do desempenho da infraestrutura observada. Uma vez que a hipótese foi elaborada, a API procede com a sugestão do que pode ser melhorado na infraestrutura com uma heurística simples do tipo "se *X* então *Y*", por exemplo. Para ilustrar, se for identificado que durante a janela de tempo que o experimento foi executado a aplicação do cliente teve o consumo de CPU alto consistentemente, a API deve sugerir que ele migre para uma máquina com maior capacidade de CPU.

Na sugestão a API também leva em consideração múltiplas condições que podem decorrer do experimento ao formular a sua hipótese, de maneira que caso seja identificado múltiplos pontos deficientes. Ela deve sugerir uma máquina que atenda todos os pontos levantados, ou que esteja mais próxima de atender. Em adição a simplesmente sugerir que o cliente precisa aumentar a CPU, como no caso do exemplo, a API também conta com um banco de dados com informações dos principais provedores de nuvem no mercado, tornando-a capaz de retornar para o cliente uma lista com os provedores, suas respectivas máquinas e preços que podem lhe atender conforme o critério identificado e listando o melhor custo benefício, auxiliando-o na tomada de decisão.

A função secundária da API, a provisão das métricas coletadas, fornece acesso a todas as métricas que são coletadas pelos Agentes Coletores. O cliente pode querer analisar as métricas e entender como a API chegou a determinada sugestão antes de tomar qualquer ação, ou ele simplesmente quer analisar os dados coletados por suas próprias razões. Para isso a API também fornece aos clientes acesso a todas as métricas, dada a janela de tempo de um experimento, representadas em gráficos produzidos pelo componente Gerador de gráficos.

3.1.4 Gerador de gráficos

Para a provisão dos dados estatísticos a API conta utiliza um componente adicional que gera e exporta gráficos informativos relativos às métricas coletadas, e também realiza a comunicação por meio de requisições HTTP.

Os componentes da API são organizados em uma arquitetura hexagonal, que ajuda a manter o código reutilizável, mantém alta coesão, baixo acoplamento, é agnóstica à tecnologia utilizada na implementação e produz componentes mais fáceis de serem testados e substituídos.

3.2 Detalhamento e cenário de uso

A arquitetura planejada tem dois fluxos principais: o *Sysperf Exporter* e sua interação com o Sistema de Monitoramento; e a API de Sugestão e as suas interações com o Sistema de Monitoramento, o Gerador de Gráficos e o cliente final.

Na parte da arquitetura relativa ao *Sysperf Exporter*, o cliente precisa executar o binário no servidor em que ele vai rodar a aplicação a ser monitorada (aplicação alvo). Esse processo pode ser adicionado na própria esteira automatizada de implantação da aplicação alvo. O servidor precisa ter um DNS exposto que deve ser configurado como ponto de entrada do *Sysperf Exporter*, para que o Sistema de Monitoramento possa capturar os lotes de métricas montados pelo mesmo.

Com a configuração inicial realizada e o *Sysperf Exporter* executando, ele imediatamente começará a coletar as métricas e, dado um intervalo de tempo configurável, o Sistema de Monitoramento fará a coleta no DNS e rota especificada.

As responsabilidades de cada componente são bem definidas, o *Sysperf Exporter* reúne e monta as métricas e o Sistema de Monitoramento faz a coleta e a persistências dos dados.

Na outra ponta da arquitetura existe a API de Sugestão que é responsável por transformar as métricas coletadas em informação e fazer a interação com o cliente final.

A API é um sistema Web independente, disponível para os clientes acessarem as métricas que foram coletadas e solicitar experimentos e sugestões.

O cliente solicita um experimento informando uma janela de tempo, ao qual a API usará como base as métricas que foram coletadas dentro do período informado.

A API solicita as métricas ao Sistema de Monitoramento passando os parâmetros informados pelo cliente. Com as métricas em mãos ela transforma esses dados em informação, aplicando a heurística definida para testar os dados, levantar hipóteses do que pode ser melhorado e listar sugestões de máquinas. A API então responde o cliente final com as hipóteses e sugestões elencadas, conforme ilustrado na Figura 3.2.

O outro fluxo da API consiste em prover as métricas que são salvas no Sistema de Monitoramento para o cliente, por meio de um painel com múltiplos gráficos informativos relativos às métricas definidas neste projeto.

A API recebe a solicitação de visualização de métricas do cliente informando uma janela de tempo, faz uma requisição ao Gerador de Gráficos repassando a janela de tempo informada, que por sua vez consulta o Sistema de Monitoramento usando os parâmetros para popular o painel gráfico e responder de volta à API, que retorna ao cliente final o painel gerado, conforme ilustrado na Figura 3.3.

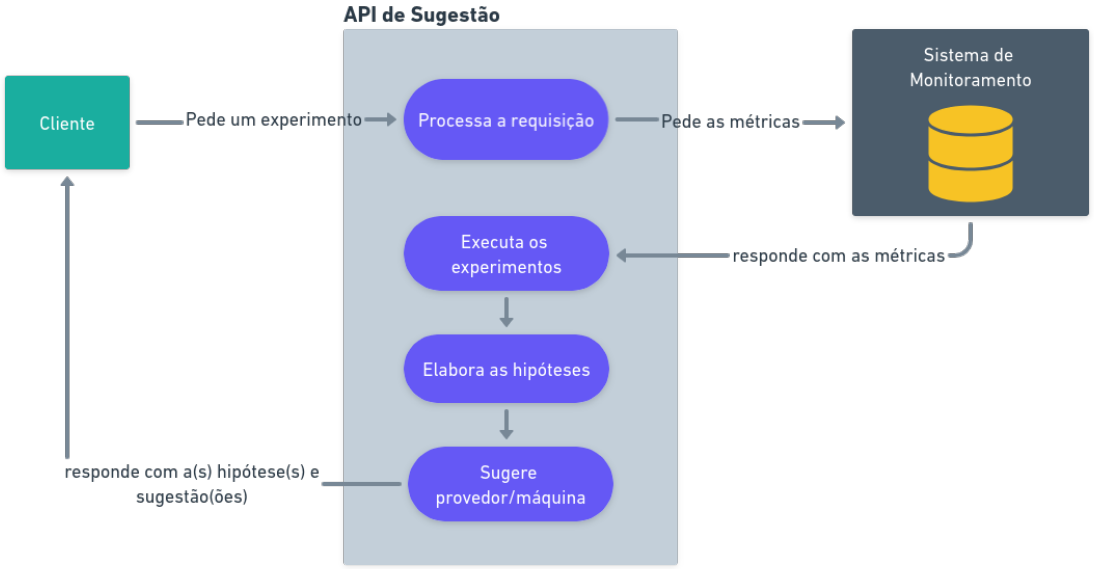

Figura 3.2: *Fluxo da arquitetura em uma requisição de sugestão.*

Fonte: Elaborada pelos autores

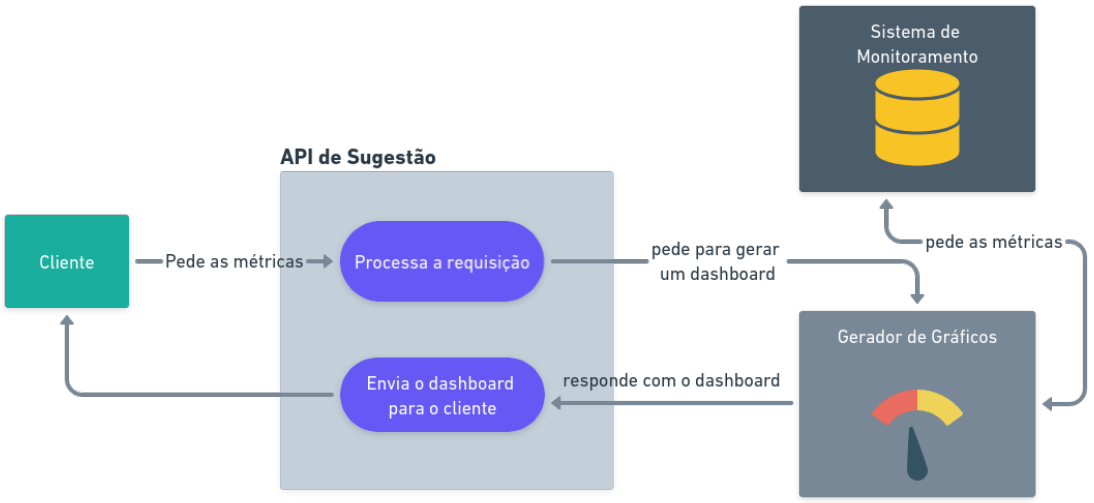

Figura 3.3: *Fluxo da arquitetura em uma requisição de métrica.*

Fonte: Elaborada pelos autores

CAPÍTULO 4

Implementação e Validação

Nesse capítulo serão apresentados os detalhes de como a arquitetura planejada foi implementada, bem como os detalhes de implementação de cada microsserviço individualmente. Também foi tratado a validação do resultado desse trabalho, apresentando a execução do sistema e os resultados alcançados.

4.1 Implementação

A implementação foi realizada em etapas e de maneira sequencial, a fim de garantir uma integração funcional ao final do ciclo de desenvolvimento.

Dado a natureza da arquitetura de microsserviços planejada, o desenvolvimento de múltiplos componentes poderia ter sido realizado simultaneamente, sem a necessidade de seguir uma ordem específica, mas por ser apenas dois desenvolvedores, foi adotada a opção por seguir uma sequência de entregas empírica.

O desenvolvimento foi iniciado pela construção do Agente Coletor e foi realizada sua integração com o Sistema de Monitoramento. Em seguida, foi realizado o desenvolvimento da API de Sugestão e integrada com o Sistema de Monitoramento. Assim foi garantido o pleno fluxo das métricas de ponta a ponta.

Com a API de Sugestão integrada ao Sistema de Monitoramento, foi procedido com o desenvolvimento da lógica de experimentos, hipóteses e sugestões. Por fim. a API de Sugestão foi integrada ao Gerador de Gráficos para fornecer mais um ângulo de visualização para o cliente final. A seguir são apresentados maiores detalhes.

4.1.1 Sistema de Monitoramento

O sistema de monitoramento foi implementado usando o Prometheus, uma ferramenta de observabilidade e monitoramento que, de acordo com a sua própria definição, "é uma ferramenta de monitoramento e alerta de sistemas de código aberto" (CLOUD NATIVE COMPUTING FOUNDATION (CNCF), 2022). Ela coleta e salva dados de uso dos recursos de sistemas em documentos no formato de chave e valor em uma série temporal, de maneira

que é possível fazer consultas e acompanhar os acontecimentos ao longo do tempo. O Prometheus é uma ferramenta já bem consolidada no mercado, utilizada por grandes empresas como Docker (INC., 2022), Digital Ocean (DIGITALOCEAN, 2022), Ericsson (ERICSSON, 2022) e várias outras. Lançado no ano de 2016 (FOUNDATION, 2016), o Prometheus já teve vários anos de amadurecimento. No repositório The Linux Foundation (2022) da solução nota-se uma comunidade ativa e em constante evolução. Por isso, além do fato de ser uma solução de código aberto, que fornece uma parcela das funcionalidades necessárias sem custos financeiros, a confiabilidade e maturidade do sistema fizeram com que ele se tornasse a primeira e única escolha para esse componente.

O processo principal do Prometheus é um servidor que é responsável por coletar os dados das aplicações e armazená-las. A ferramenta disponibiliza uma API HTTP para a consulta externa aos dados salvos em seu banco, que podem servir de insumo para outras aplicações. Para que ela possa realizar a coleta efetiva dos dados, a aplicação alvo[1] precisa ser instrumentada utilizando uma das bibliotecas (ou *Software Development Kit* (SDK)) próprias de acordo com a linguagem de programação escolhida na implementação.

O SDK do Prometheus é responsável por organizar e gerenciar os dados em lotes, bastando ao cliente instrumentar os dados e adicioná-los ao lote através da API fornecida. A aplicação também precisa disponibilizar uma rota HTTP (chamada de *target*) para o Prometheus. É nessa rota que a ferramenta fará as requisições de coleta dos lotes de dados.

O *target* é definido na configuração do Prometheus e pode ser controlado através da API administrativa do mesmo, possibilitando a adição e edição de *targets* em tempo de execução, o que deixa o serviço mais configurável. É possível adicionar novas aplicações ao monitoramento sem a necessidade de realizar uma nova implantação sempre que precisar adicionar ou remover um *target*. O Prometheus ainda dá a opção para o cliente criar alertas personalizados, baseados em alguma métrica definida pelo cliente e caso essa métrica ultrapasse o valor estabelecido um alerta é emitido. A emissão de alertas pode ser integrada a vários sistemas externos ao Prometheus como, e-mail, *PagerDuty* (PAGERDUTY, 2022), *Slack* (TECHNOLOGIES, 2022), etc.

4.1.2 Agente Coletor de Métricas

O Agente Coletor de Métricas (*Sysperf Exporter*) é um binário executável implementado utilizando a linguagem de programação Golang (GOOGLE, 2022) (também conhecida por *Go*) e compilada para *linux86_64*.

Go é conhecida por ser uma linguagem de programação compilada, rápida em execução, fornece suporte a programação concorrente e gera binários pequenos, sendo

[1] A aplicação alvo no contexto do projeto é o agente coletor de métricas.

todos esses aspectos requisitos para o projeto proposto, pois é de suma importância que o *Sysperf Exporter* exerça o mínimo de influência possível no seu hospedeiro, para que não afete a execução da aplicação alvo do monitoramento e não produza efeitos colaterais na sua execução. Dessa forma, o *Sysperf Exporter* deve ser um agente "invisível" no sistema hospedeiro em termos de uso de recursos computacionais.

O *Sysperf Exporter* é implementado em um único módulo, separado em pequenas funções especializadas na captura de cada métrica. Na sua função *main* ele instrumenta o *SDK* do Prometheus, realiza as configurações para iniciar os coletores de métricas e configura o *handler* HTTP que é responsável por expor a rota definida no *target* do Prometheus, criando um servidor HTTP para lidar com as requisições de coleta de dados.

Após a definição das configurações, o *Sysperf Exporter* cria uma *goroutine* para cada métrica a ser coletada. *goroutines* em Go são *threads* leves gerenciadas pelo *runtime* do Go, que fornecem as vantagens da programação concorrente em mais alto nível de programação, com APIs simples de serem usadas.

Cada *goroutine* criada para cada métrica executa um bloco de função em um laço de repetição que dura enquanto a aplicação estiver executando. Dentro desse laço é executada a função que coleta a métrica em intervalos de tempo, sendo o intervalo configurável por variáveis de ambiente.

4.1.2.1 Coleta de métricas

A implementação realizada considera um conjunto padrão de métricas de uso do recurso computacional. Estas métricas foram definidas de forma a serem genéricas o suficiente para serem relevantes em qualquer contexto. No total capturamos cerca de 97 métricas, sendo 54 métricas que foram coletadas de maneira ativa e 43 delas sendo métricas expostas pelo próprio agente coletor. Entre as métricas que foram coletadas:

- Aceleração da CPU (cpu_core_throttles_total): é a quantidade de vezes que a CPU teve que interromper seu funcionamento. Esse cenário pode acontecer por vários motivos, desde problemas físicos como alta temperatura e pouco fluxo de ar para dentro do computador, até problemas como muitos programas sendo executados ao mesmo tempo.
- Uso de memória RAM: quantos bytes estão sendo usados pela memória principal do computador.
- Uso de memória SWAP: quantos bytes estão sendo usados pela memória secundária do computador, geralmente um tipo de memória mais lento, apenas usado quando a memoria principal está indisponível.

Conforme ilustrado na Figura 4.1, as funções de coleta de dados leem diretamente dos arquivos do sistema operacional os processos em execução e os recursos usa-

dos. Para tal, é feito uso direto dos sistema de arquivos `/proc/`, comum aos sistemas operacionais de distribuição Linux. Este diretório "contêm uma hierarquia de arquivos que representam o estado atual do *kernel*, permitindo que usuários e aplicações vejam a visão do *kernel* em relação ao sistema" (FEDORA, 2021).

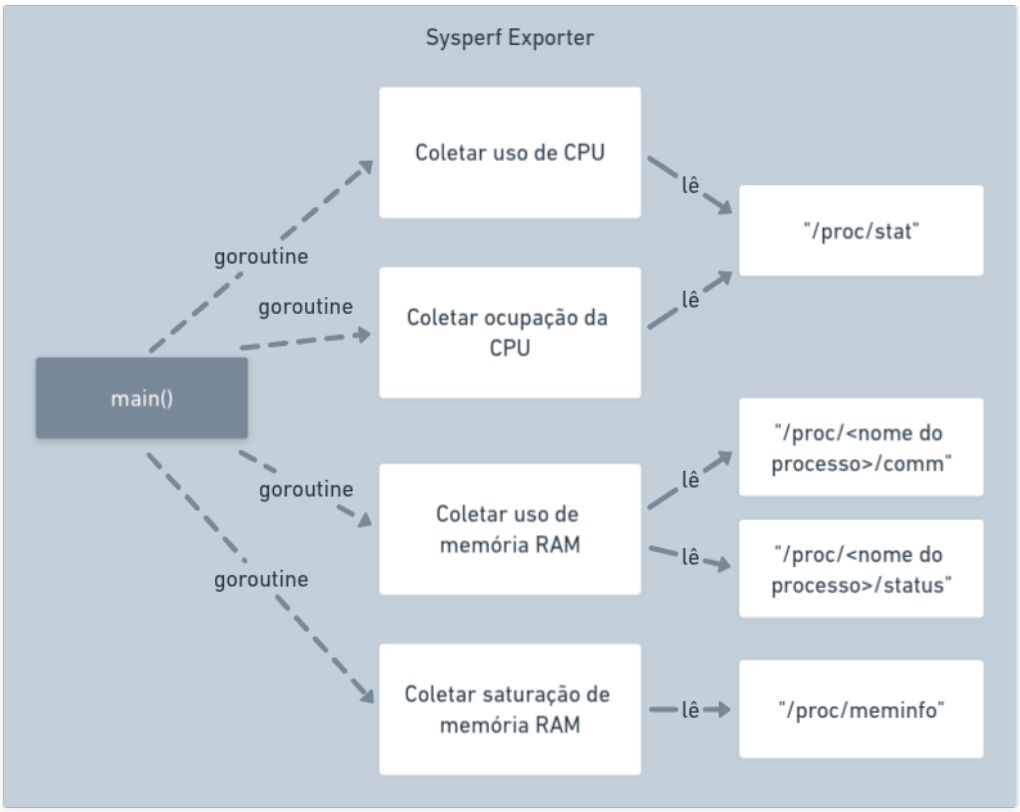

Figura 4.1: *Implementação do Sysperf Exporter.*

Fonte: Elaborada pelos autores

O *Sysperf Exporter* consegue, através do `/proc/` ter acesso direto ao que está acontecendo no sistema naquele momento pelo ponto de vista do *kernel*. Cada função de coleta acessa o(s) arquivo(s) específico(s) no `/proc/` que contém as informações necessárias para montar os dados das métricas.

A função que lê esses arquivos precisa realizar a conversão dos mesmos, que vêm em formato de texto, para estruturas de dados capazes de serem persistidas e consultadas em banco de dados. Depois que a métrica é coletada do `/proc/` e foi convertida, seu valor é retornado para o método *main* que então enfileira a métrica no lote de coleta do Prometheus.

O Código 4.1 apresenta a coleta de métricas cujo ciclo foi apresentado, considerando apenas uso de CPU e memória. O laço de repetição continua em execução indefinidamente enquanto a aplicação estiver executando.

Código 4.1 Instrumentação das goroutines de coleta de dados

```
// uso de CPU
go func() {

  for {
    sample, err := SampleCPUUsage(agentTime)
    if err != nil {
      log.Fatal(err)
    }
    cpuUsageGauge.Set(sample.Usage)
    cpuSaturationGauge.Set(sample.Busy)
    time.Sleep(agentTime) // Intervalo de execução do agente
  }

}()

// uso de memória
go func() {

  for {
    samples, err := SampleMemoryUsage()
    if err != nil {
      log.Fatal(err)
    }

    for _, sample := range samples {
      memUsageGauge.WithLabelValues(strings.TrimSuffix(sample.Command, "\n"), hostname).Set(
          sample.SizeKb / 1000)
    }

    time.Sleep(agentTime) // Intervalo de execução do agente
  }

}()
```

Um das principais propostas do agente coletor seria capturar métricas oriundas de contêineres, mas para coletar essas métricas se torna necessário o uso de eBPF. Devido a enorme dificuldade em utilizar essa tecnologia no momento, isto tornaria a implementação muito frágil e propensa a não funcionar.

Foram desenvolvidos alguns protótipos que funcionaram por curtos períodos de tempo, mas foi o suficiente para validar que é capaz de realizar o que era desejado. Através do eBPF é possível coletar e ler as métricas de cada contêiner individualmente e fazer sugestões a nível de contêiner, além de sugestões a nível de máquina.

A principal dificuldade nesse quesito é a falta de portabilidade do eBPF, que dificulta rodar até na mesma versão da distribuição Linux. Um código escrito em uma semana, na outra semana devido a uma atualização já para de funcionar, ou seja, ainda está muito volátil embora esteja a cada dia que passa mais maduro.

4.1.3 Gerador de Gráficos

O gerador de gráficos foi implementado usando Grafana (LABS, 2022), uma suíte de ferramentas de código aberto que incluem unificação e visualização de métricas, *logs* e rastreabilidade. Unificar a visualização de dados de diferentes banco de dados em diferentes locais e gerar *dashboards* informativos é o cerne do Grafana. Dessa maneira ela fornece uma lente que abrange todo um ecossistema em um único lugar.

Os *dashboards* do Grafana são altamente configuráveis, cabendo ao usuário a customização conforme a sua necessidade. Para uso mais genérico, o Grafana provê *dashboards* padrão que já vem na solução inicial, sendo a customização e criação de novas visualizações opcionais. Nos *dashboards* podem ser incluídos toda sorte de gráficos como o mostrado na Figura 4.2, o que inclui mapas geográficos, mapas de calor, histogramas, etc.

Figura 4.2: *Exemplo de um gráfico no Grafana.*

Fonte: Elaborada pelos autores

O Grafana suporta a construção de consultas que são usadas para montar os gráficos dos *dashboards*. É por meio dessa ferramenta que a personalização dos gráficos é realizada. A linguagem utilizada para escrever as consultas depende do banco em que os dados serão consultados. Por exemplo, neste trabalho os dados são persistidos no Prometheus, logo para construir os gráficos das métricas é preciso escrever as consultas na linguagem utilizada pelo Prometheus, a PromQL (AUTHORS, 2022), e assim para cada banco e sua respectiva linguagem nativa.

O Grafana não retém dados, pois não possui um banco de dados interno para gerar as visualizações. Ao invés disso, ele retém a suite possui uma vasta coleção de bibliotecas capazes de se conectar a várias bases de dados diferentes. Mesmo as bases de dados que por ventura não são suportados nativamente pela ferramenta, dada a sua natureza de código aberto, é possível escrever *plugins* que satisfaçam a integração.

A implantação do Grafana pode ser realizada em nuvem ou *on-premise*, atendendo às necessidades especificas de cada negócio. O cliente que não pode correr absolutamente nenhum risco de vazamento de dados, como sistemas financeiros ou governa-

mentais, pode optar por gerenciar sua própria infraestrutura e realizar a implantação do Grafana *on-premise*. Já clientes que não tem essa restrição podem fazer a implantação no ambiente de nuvem, deixando a instrumentação da infraestrutura mais simples. Ainda é possível utilizar o Grafana na nuvem no modelo *Software as a Service* (SaaS), que não requer nenhum tipo de implantação, bastando se conectar e usar.

Neste projeto o Grafana é usado conectado a base de dados do Prometheus, que é onde as métricas das aplicações são persistidas. Sua responsabilidade é gerar os *dashboards* e gráficos dispondo as métricas das aplicações para agregar informação ao cliente final, sendo que esses gráficos são exportados em tempo real à medida que a API de sugestão requisitar.

Os gráficos do Grafana são representações das consultas realizadas usando o *PromQL*, uma variante de SQL especializada em dados de séries temporais, os gráficos são organizados de maneira simples com a seguinte nomenclatura:

- **Painel** - constitui um gráfico individual
- **Dashboard** - constitui uma coleção de paneis

Cada *dashboard* é persistido em formato *JavaScript Object Notation* (JSON) sendo este texto puro.

4.1.4 API de Sugestão

A API de Sugestão é um servidor HTTP que representa o ponto central de interação do cliente final[2]. É através desta API que o cliente requisita e recebe as sugestões de melhorias de infraestrutura e pode consultar as métricas de suas aplicações.

O fluxo da aplicação começa com uma requisição HTTP do cliente pedindo a realização de um experimento. A API integra diretamente com o Prometheus onde é feita a busca dos dados históricos de performance da aplicação. Uma vez que o Prometheus fornece os dados, os mesmos são encaminhados para o motor de inteligência, que realiza a análise e gera as sugestões de melhorias na infraestrutura. A análise é então retornada para o cliente como resposta da requisição HTTP no formato *JavaScript Object Notation* (JSON).

A análise da infraestrutura sempre é realizada sob demanda, visto que o cliente precisa fazer um requisição para que ela aconteça. A API possui uma segunda rota que é responsável por exportar os gráficos das métricas para o cliente. Os gráficos são gerados em formato de série temporal expondo todas as métricas previamente listadas. Para cada métrica é gerado um gráfico. A API é implementada utilizando a linguagem de

[2]Cliente final neste contexto se refere aos interessados em observar o desempenho de uma aplicação.

programação Python, que é uma linguagem com amplo suporte para desenvolvimento Web, capaz de entregar aplicações em pouco tempo de desenvolvimento, seguindo o conjunto de restrições de arquitetura *Representational State Transfer* (REST) (HAT, 2020). Isso significa que a API não guarda nenhum estado e a informação necessária para realizar toda e qualquer operação, tais como autenticação, parâmetros adicionais necessários para uma operação, etc., tem que estar atrelada à requisição HTTP. Esses parâmetros podem ser passados no corpo da requisição, por meio de parâmetros de URL ou cabeçalhos.

A análise é feita através de uma abstração chamada Usage, Saturation, Errors, do inglês, Uso, Saturação e Erros (USE) criada pelo Brendan Gregg (GREGG, 2022) que consiste em analisar cada recurso individualmente (CPU, Memória RAM, Disco, etc.) nas três categorias: uso, saturação e erros. Embora pareça simples, quando a análise é feita olhando as métricas em conjunto, ajuda a reconhecer gargalos ou ineficiências. Por exemplo, considere o uso de memória RAM durante um período de uma hora, se o uso permanece em 80% constantemente, sua saturação, ou seja uso de memória SWAP, também foi constante e engatilhou 2 vezes um Out-Of-Memory (OOM), que é um erro a nível de kernel e significa que algum processo solicitou mais memória do que havia disponível, nesse exemplo fica evidente que existe um gargalo de memória.

4.2 Validação

A estratégia de validação adotada neste trabalho consiste em configurar dois ambientes distintos e isolados que possam simular o funcionamento integrado de todos os componentes da arquitetura. Em um dos ambientes o *Sysperf Exporter* deve estar ativo e coletando métricas e no outro ambiente deve estar ativos o Sistema de Monitoramento, Gerador de Gráficos e a API de Sugestão.

Para considerar o experimento como válido o *Sysperf Exporter* deve ser capaz de coletar as métricas definidas no ambiente em que ele se encontra, o Sistema de Monitoramento deve ser capaz de coletar as métricas no *Sysperf Exporter* e persisti-las em seu banco de dados, o Gerador de Gráficos deve ser capaz de consultar o Sistema de Monitoramento, gerar os painéis com as métricas recebidas e enviá-las à API de Sugestão, e a API de Sugestão deve ser capaz de se comunicar com o Sistema de Monitoramento e consultar as métricas. Além disso, esta última deve ser capaz de pedir os painéis ao Gerador de Gráficos, recebê-los e enviá-los como resposta, experimentar com as métricas solicitadas, gerar hipóteses e sugestões e retornar esse resultado como resposta ao cliente.

Como parte da validação foi desenvolvido uma aplicação simples utilizando Golang, que consiste em um servidor Web que “vaza” (utiliza intensamente) memória propositalmente. A aplicação foi chamada de leaky_web_app (MARTINS, 2022), tendo como propósito simular uma aplicação problemática e gerar métricas para que o *Sysperf*

Exporter colete. Ela possui uma rota aberta que ativa o processo de vazamento de memória.

O vazamento de memória funciona da seguinte maneira: a aplicação verifica quantas CPUs lógicas a máquina disponibiliza e escolhe aleatoriamente quantas das CPUs disponíveis serão utilizados concorrentemente no processo de vazamento de memória. Para cada um dos CPUs selecionados a aplicação então roda a rotina de vazamento de memória em uma *goroutine* (*threads* da linguagem Golang). A aplicação então cria dois objetos do tipo "T" que possuem membros internos também do tipo "T" e outro membro que é uma coleção de bytes, remove a referência de ambos do garbage colector para que eles continuem crescendo indefinidamente, e atribui o ponteiro de um para o outro como membros internos, criando uma dependência cíclica e eventualmente gerando um *Out of Memory* (OOM), do inglês, falta de memória; vazando para a memória *heap*.

Para realizar a validação das implementações foi montado um ambiente remoto na nuvem *Amazon Web Services* (AWS), cujo diagrama de implantação é apresentado na Figura 4.3. No ambiente remoto, nomeado de *cliente*, foi utilizado uma máquina virtual do tipo *t2.micro* com 1GB de RAM e 1 núcleo de CPU e no ambiente *back-end* uma máquina *t3.medium* com 4GB de RAM e 2 núcleos de CPU, conforme apresentado na Figura 4.4.

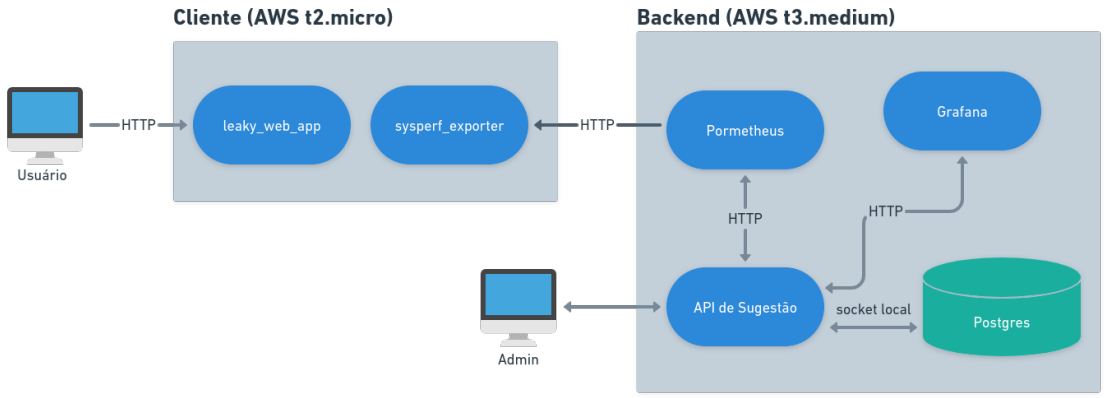

Figura 4.3: *Estrutura do ambiente de validação.*

Fonte: Elaborada pelos autores.

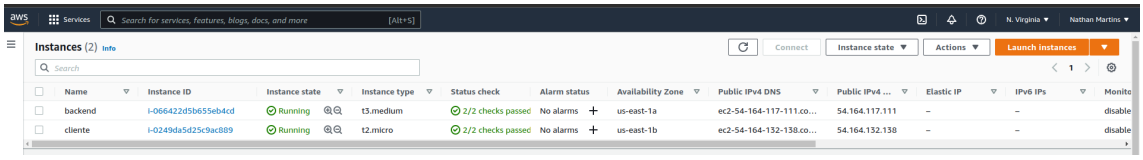

Figura 4.4: *Máquinas virtuais na AWS utilizadas para validação.*

Fonte: Elaborada pelos autores.

Para a execução da validação, na máquina cliente foram executados dois binários, o leaky_web_app (este sendo executado dentro de um contêiner para não causar uma pane

completa e para que este possa ser reiniciado sempre que estourar seu limite de memória) e o agente coletor de métricas sysperf que foi executado sem o uso de contêineres. O webapp foi executado atendendo tráfico HTTP na porta 8081 e o exportador também HTTP na porta 8081, ambas abertas para a internet toda. A Figura 4.5 apresenta esses serviços em execução.

Figura 4.5: *Serviços em execução no cliente.*

Fonte: Elaborada pelos autores.

Figura 4.6: *Cadastrando cliente através da API.*

Fonte: Elaborada pelos autores.

Na máquina *back-end* foram executados 3 sistemas (com seus sub-sistemas e dependências incluídas) sendo eles: a API de Sugestão sysperf_api com seu banco de dados, o Sistema de Monitoramento *Prometheus* e o sistema Gerador de Gráficos *Grafana*. Todos estes sistemas foram executados sem o uso de contêineres, atendendo a chamadas HTTP nas portas 8081, 9090, e 3000, respectivamente. A única porta aberta externamente à internet foi a porta 8081, do *back-end*.

Após iniciar cada um dos sistemas, foi realizada uma requisição na rota do sysperf_api para cadastrar a máquina cliente na própria API, como mostrado na Figura 4.5. A requisição foi usada para cadastro também no Prometheus através da integração com este sistema, como mostrado nas Figuras 4.6 e 4.7.

Também foi realizada uma requisição na rota raiz do leaky_web_app que ativa o vazamento de memória, além de uma requisição sysperf_api para criar o experimento, como mostrado na Figura 4.8.

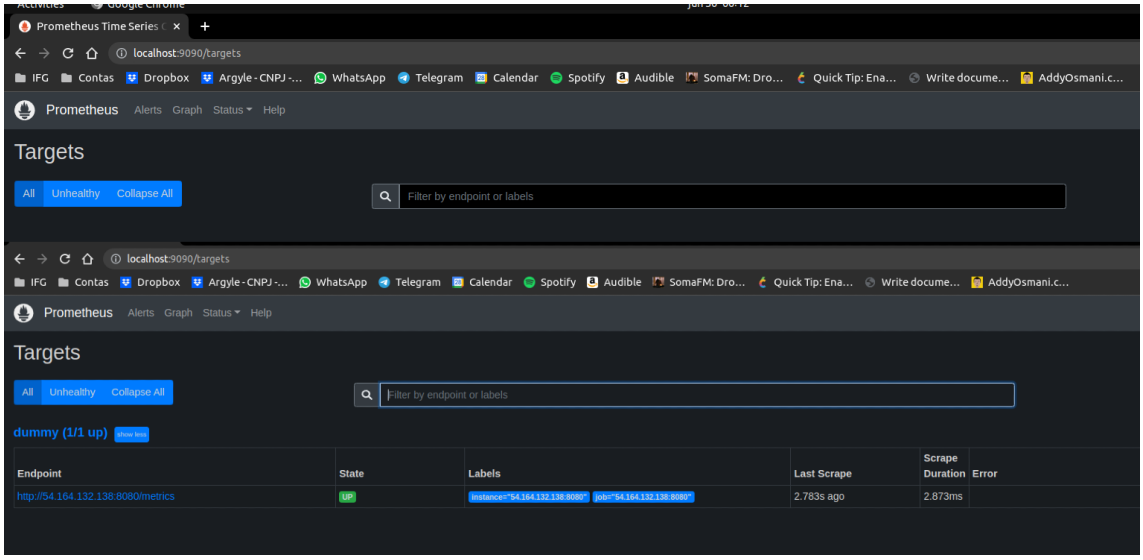

Figura 4.7: *Prometheus antes e depois de cadastrar a máquina cliente.*

Fonte: Elaborada pelos autores.

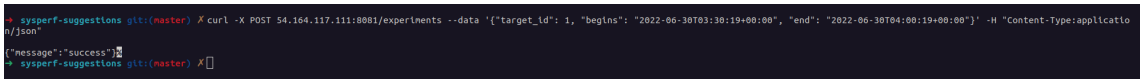

Figura 4.8: *Criando o experimento de 30 minutos.*

Fonte: Elaborada pelos autores.

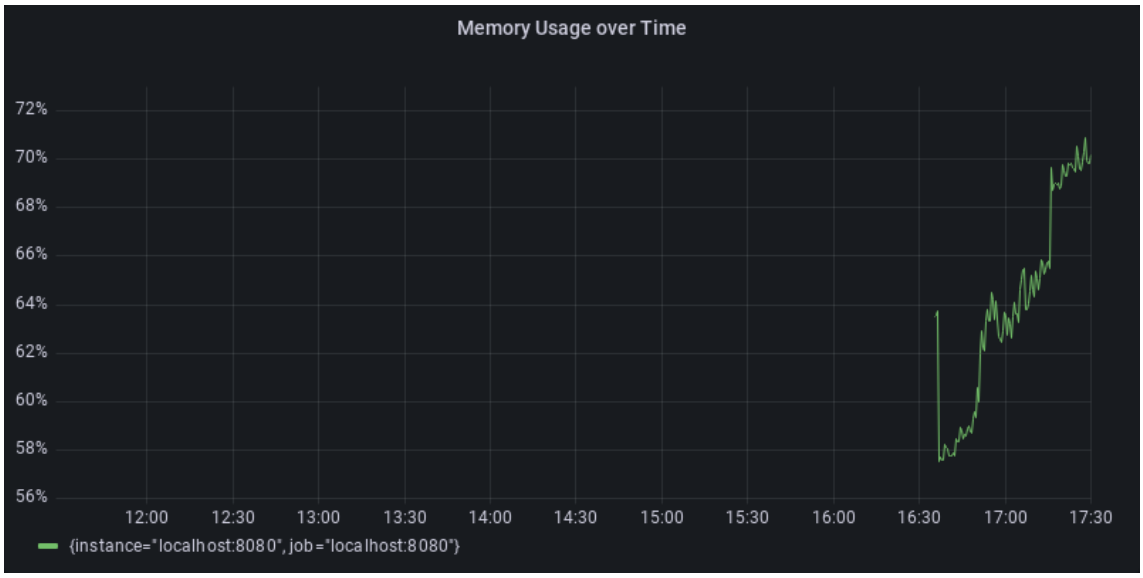

Figura 4.9: *Resultado do experimento em gráfico extraído do base64*

Fonte: Elaborada pelos autores.

Após aguardar trinta minutos para que acumule algumas métricas no banco de dados, foi realizada uma requisição na API de Sugestão pedindo um painel de métricas com a janela de 2022–06–30T03:30:19+00:00 até 2022–06–30T04:00:19+00:00 e foi recebido o painel conforme imagem 4.9 que vem no codificada na API em formato

Figura 4.10: *Resultado do experimento e o gráfico em base64*

Fonte: Elaborada pelos autores.

base64. Na imagem nota-se que o uso de memória é elevado conforme o esperado.

Para validar a funcionalidade de hipóteses e sugestões, foi realizada uma requisição na rota que cria um experimento, com duração de 40 minutos através de duas chamadas para a API de sugestão. Com isso, foi aguardado a duração do experimento e, consequentemente, aguardado cerca de 3 minutos (realizando chamadas HTTP para consultar o estado) para a elaboração da hipótese na API. Ao final foi obtida a confirmação da hipótese, cujo resultado informou que a aplicação está com problemas de memória, conforme o esperado, e fez a sugestão de melhoria de máquina virtual para uma *t2.medium*, concluindo assim a validação ponta-a-ponta dos componentes. O painel obtido é apresentado em base64 na Figura 4.10 e em forma gráfica na Figura 4.9.

A Figura 4.11 apresenta a checagem de saúde dos sistemas no *back-end*.

```
→ ~ curl -v http://54.164.117.111:8081/health
*   Trying 54.164.117.111:8081...
* Connected to 54.164.117.111 (54.164.117.111) port 8081 (#0)
> GET /health HTTP/1.1
> Host: 54.164.117.111:8081
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Date: Thu, 30 Jun 2022 23:07:52 GMT
< Content-Length: 40
<
* Connection #0 to host 54.164.117.111 left intact
{"status":"sysperf-backend is healthy."}%
→ ~ curl -v http://54.164.117.111:9090/-/healthy
*   Trying 54.164.117.111:9090...
* Connected to 54.164.117.111 (54.164.117.111) port 9090 (#0)
> GET /-/healthy HTTP/1.1
> Host: 54.164.117.111:9090
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Thu, 30 Jun 2022 23:07:54 GMT
< Content-Length: 23
< Content-Type: text/plain; charset=utf-8
<
Prometheus is Healthy.
* Connection #0 to host 54.164.117.111 left intact
→ ~ curl -v http://54.164.117.111:3000/api/health
*   Trying 54.164.117.111:3000...
* Connected to 54.164.117.111 (54.164.117.111) port 3000 (#0)
> GET /api/health HTTP/1.1
> Host: 54.164.117.111:3000
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Cache-Control: no-cache
< Content-Type: application/json; charset=UTF-8
< Expires: -1
< Pragma: no-cache
< X-Content-Type-Options: nosniff
< X-Frame-Options: deny
< X-Xss-Protection: 1; mode=block
< Date: Thu, 30 Jun 2022 23:07:56 GMT
< Content-Length: 69
<
{
  "commit": "0641b5d0c",
  "database": "ok",
  "version": "9.0.2"
* Connection #0 to host 54.164.117.111 left intact
```

Figura 4.11: *Verificação de saúde dos sistemas no backend*

Fonte: Elaborada pelos autores.

CAPÍTULO 5

Considerações finais

A proposta deste trabalho foi alcançada através da implementação de todos os componentes propostos, sendo o resultado obtido a aplicação da prova de conceito, que demonstra que ainda há pontos a serem melhorados.

A funcionalidade da solução bruta apresentada demonstra que é possível desenvolver uma ferramenta de observabilidade que não requer instrumentação do código-fonte, com capacidade de identificar possíveis problemas de infraestrutura e sugerir máquinas virtuais de provedores reais, que atendam aos requisitos do sistema.

Apesar de haver alternativas ao se considerar cada componente individualmente, o valor está na solução como um todo trabalhando em conjunto. Desta forma, os resultados apresentados neste trabalho podem ser aplicados com o objetivo de facilitar o acesso a informação de nível gerencial do desempenho de sistemas, assim como, a formulação de sugestões de melhorias empregando o mínimo de esforço, podendo ser utilizada tanto por especialistas em termos de tecnologia, quanto por leigos.

5.1 Dificuldades encontradas no desenvolvimento do trabalho

5.1.1 Monitoramento de contêineres isoladamente

Usando eBPF e realizando alguns testes manuais é possível realizar o monitoramento por contêiner de maneira isolada do restante do sistema. Foi possível executar o agente coletor em uma máquina virtual, porém houve dificuldade em viabilizar a execução do mesmo dentro de um contêiner e ainda ser capaz de coletar métricas de outros contêineres no mesmo ambiente, esse caso de uso agrega valor, pois existem ambientes que não permitem a execução de binários que não estejam em contêineres.

O resultado deste trabalho é capaz de monitorar, por exemplo, um servidor como um todo, mas não consegue monitorar cada contêiner executando no servidor individualmente.

5.1.2 Portabilidade do eBPF

Uma das principais dificuldades do eBPF é sua portabilidade. Por ser uma tecnologia a nível de *kernel* é necessário que algumas bibliotecas e arquivos de código-fonte estejam disponíveis para que o programa compile e execute corretamente. Essa necessidade abre margem para uma outra classe de problemas: compatibilidade entre versões diferentes do *kernel*. Essas mudanças geram um enorme inconveniente, pois causam o término da aplicação, e por menor que sejam as diferenças entre uma versão e outra, é necessário implementar uma versão para lidar com cada término, tornando o propósito de fazer uma ferramenta agnóstica ao sistema operacional e suas versões impraticável.

Ainda colocando esse problemas em perspectiva, foi adotada a opção de diminuir o escopo da aplicação, focando em sistemas de distribuição Linux. Dessa maneira, foi possível encontrar um terreno comum em como e onde captar as métricas necessárias para a realização desse trabalho sem grande prejuízo. O terreno comum nesse caso é o sistemas de arquivos `/proc/`, que possibilita a leitura de todos os dados necessários para completar as métricas, sendo escrito pelo *kernel*, com o único revés de ter que lidar com a conversão de arquivo de texto para uma estrutura de dados entendível pelo programa.

Para lidar com o problema de portabilidade, desde 2021 existe uma iniciativa chamada *eBPF CO-RE Compile Once, Run Everywhere* mas ainda está sendo desenvolvida e não está pronta para uso em produção.

5.1.3 Dificuldade de programar em C com a API do *kernel*

O único jeito de usar o eBPF é através da API do Kernel que é feito na linguagem C. Essa API é bastante complexa e requer um estudo aprofundado para entender como tudo funciona, o que fugia do escopo da implementação inicial. Além disso, o ecossistema ao redor dessa linguagem é bem antigo e às vezes meio arcaico, o que gerava bastante confusão ao tentar compilar os binários.

5.1.4 *Back-end*

Inicialmente a proposta incluía um servidor HTTP mais robusto, com um banco de dados ao lado. A comunicação seria realizada utilizando o protocolo *gRPC* (INDRASIRI; KURUPPU, 2020), visando deixar a comunicação mais simples e com um ponto único. Contudo, essa arquitetura se mostrou muito complexa e desnecessária para a primeira versão.

Também estava no planejamento adicionar um componente de *Machine Learning* para traçar padrões de uso das aplicações observadas, mas não houve tempo hábil para a implementação.

5.1.5 Tolerância a falhas

Atualmente o *Sysperf Exporter* não possui nenhum tipo de tolerância a falhas, significando que caso aconteça algum erro de tempo de execução, a aplicação é encerrada e para de coletar métricas. Para o cliente isso implica em ter que acessar o seu ambiente e executar a aplicação novamente para que ela volte a coletar métricas.

Quando o *Sysperf Exporter* falha não existe uma maneira de o cliente saber, ele só irá notar quando ele perceber que as suas métricas não estão sendo salvas.

5.2 Trabalhos Futuros

Com base nos resultados obtidos e dificuldades encontradas no desenvolvimento do trabalho foram observados os seguintes trabalhos futuros:

- Criar uma versão com monitoramento de contêineres para poder fazer sugestões mais completas além de apenas máquinas virtuais como um todo. Seria interessante poder analisar contêineres e sugerir melhorias a este nível.
- Adicionar mais recursos a serem analisados, apesar de serem coletadas múltiplas métricas atualmente a análise é realizada apenas para memória RAM.
- Tornar o *back-end* mais robusto com uso de tarefas assíncronas para realizar os experimentos. Atualmente é realizado de forma síncrona que pode ocasionar sobrecarga na API de Sugestão ou até mesmo perca de uma analise de longa-duração em andamento.
- Melhorar a resiliência do agente coletor para que este seja mais robusto no tratamento de erros e o seu funcionamento não seja interrompido.
- Implementar um modelo de inteligência artificial ou *Machine Learning* para permitir análises com previsões, além de permitir o sistema ajustar suas análises através da propagação reversa.
- Criar funcionalidade de relatórios que permita exportar as análises para uso em *Business Intelligence*.
- Aumentar a quantidade de métricas atualmente coletadas pelo agente.
- Criar uma interface gráfica para uso da API de sugestão.
- Implementar a gestão de usuários e grupos.

Referências Bibliográficas

AUTHORS, Prometheus. **QUERYING PROMETHEUS**. 2022. Disponível em: `https://prometheus.io/docs/prometheus/latest/querying/basics/`. Acesso em: 26 jun. 2022. 43

AVRAM, Maricela-Georgiana. Advantages and challenges of adopting cloud computing from an enterprise perspective. **Procedia Technology**, Elsevier, v. 12, p. 529–534, 2014. 22

BECK, Kent. **Extreme programming explained: embrace change**. [S.l.]: Addison-Wesley Professional, 2000. 24

BEGEL, Andrew; MCCANNE, Steven; GRAHAM, Susan L. Bpf+ exploiting global data-flow optimization in a generalized packet filter architecture. *In*: **Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 1999. p. 123–134. 20

BERNSTEIN, David. Containers and Cloud: From LXC to Docker to Kubernetes. **IEEE Cloud Computing**, IEEE, v. 1, n. 3, p. 81–84, 2014. 18

BHARDWAJ, Leena Jain Sushil; JAIN, Sandeep. Cloud computing: A study of infrastructure as a service (iaas). **International Journal of Engineering and Information Technology**, v. 2, n. 1, p. 1–4, 2010. 24

BRASIL IMPORTEX. **10 modelos de contêineres e suas características**. 2022. Disponível em: `https://brasilimportex.com.br/2020/07/29/10-modelos-de-containers-e-suas-caracteristicas/`. Acesso em: 01 jul 2022. 25

CAO, Hanyang; FALLERI, Jean-Rémy; BLANC, Xavier; ZHANG, Li. JSON Patch for Turning a Pull REST API into a Push. *In*: SPRINGER. **International Conference on Service-Oriented Computing**. [S.l.], 2016. p. 435–449. 34

CASSAGNES, Cyril; TRESTIOREANU, Lucian; JOLY, Clement; STATE, Radu. The rise of ebpf for non-intrusive performance monitoring. *In*: . [S.l.: s.n.], 2020. p. 1–7. 19

CLOUD NATIVE COMPUTING FOUNDATION (CNCF). **OVERVIEW | Prometheus**. 2022. Disponível em: https://prometheus.io/docs/introduction/overview/. 19, 38

DATADOG. **Modern Application Performance Monitoring (APM) End-to-end distributed tracing and APM at scale, correlated to all telemetry. No sampling, no blindspots, no limits.** 2022. Disponível em: https://www.datadoghq.com/product/apm/. 18

DELGADO, Caio. **Containers ou Máquinas Virtuais - Quando usar cada um?** 2020. Disponível em: https://caiodelgado.dev/container-vm/. Acesso em: 15 maio 2022. 26

DIGITALOCEAN, LLC. **Digital Ocean**. 2022. Disponível em: https://www.digitalocean.com/. Acesso em: 26 jun. 2022. 39

ERICSSON, Telefonaktiebolaget LM. **Ericsson**. 2022. Disponível em: https://www.ericsson.com/en/about-us. Acesso em: 26 jun. 2022. 39

FEDORA. **Chapter 19. The proc File System**. 2021. Disponível em: https://docs.fedoraproject.org/en-US/Fedora/14/html/Deployment_Guide/ch-proc.html#:~:text=The%20%2Fproc%2F%20directory%20(also,kernel's%20view%20of%20the%20system. Acesso em: 14 jun. 2022. 41

FOUNDATION, The Linux. **Prometheus 1.0 is here**. 2016. Disponível em: https://www.cncf.io/blog/2016/07/18/prometheus-1-0-is-here/. Acesso em: 26 jun. 2022. 39

FOWLER, Martin. **BlueGreenDeployment**. 2010. Disponível em: https://martinfowler.com/bliki/BlueGreenDeployment.html. Acesso em: 26 jun. 2022. 33

GOOGLE. **Golang programming language**. 2022. Disponível em: https://go.dev/solutions/#case-studies. Acesso em: 15 maio 2022. 39

GREGG, Brendan. **The USE Method**. 2022. Disponível em: https://www.brendangregg.com/usemethod.html. Acesso em: 12 jun. 2022. 45

GRIFFIN, Jesse. Hexagonal-driven development. *In*: **Domain-Driven Laravel**. [S.l.]: Springer, 2021. p. 521–544. 32

HAT, Inc. Red. **O que é API REST?** 2020. Disponível em: https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api. Acesso em: 16 jun. 2022. 45

____. **RedHat**. 2022. Disponível em: https://www.redhat.com/pt-br. Acesso em: 08 jun. 2022. 28

HUMBLE, Jez; FARLEY, David. **Continuous delivery: reliable software releases through build, test, and deployment automation**. [S.l.]: Pearson Education, 2010. 24

INC., Docker. **Docker**. 2022. Disponível em: https://www.docker.com/. Acesso em: 26 jun. 2022. 39

INC., Pepperdata. **Why Observability is Vital in Today's Big Data World**. 2020. Disponível em: https://www.pepperdata.com/blog/observability-vital-big-data/. Acesso em: 08 jun. 2022. 30

INDRASIRI, Kasun; KURUPPU, Danesh. **gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes**. [S.l.]: O'Reilly Media, 2020. 52

INTEREST, Inc. Software in the Public. **Debian**. 2022. Disponível em: https://www.debian.org/index.pt.html. Acesso em: 08 jun. 2022. 28

KERNELNEWBIES. **Linux 3.18 has been released**. 2014. Disponível em: https://kernelnewbies.org/Linux_3.18. Acesso em: 08 jun. 2022. 27

LABS, Grafana. **Grafana: The open observability platform**. 2022. Disponível em: https://grafana.com/. Acesso em: 16 jun. 2022. 43

LEE, In; LEE, Kyoochun. The Internet of Things (IoT): Applications, investments, and challenges for enterprises. **Business horizons**, Elsevier, v. 58, n. 4, p. 431–440, 2015. 17

LEVIN, Joshua; BENSON, Theophilus A. Viperprobe: Rethinking microservice observability with ebpf. *In*: **2020 IEEE 9th International Conference on Cloud Networking (CloudNet)**. [S.l.: s.n.], 2020. p. 1–8. 19

MAJEED, Abdul; RAUF, Ibtisam. Mvc architecture: a detailed insight to the modern web applications development. **Peer Review Journal of Solar & Photoenergy Systems**, v. 1, n. 1, p. 1–7, 2018. 32

MARKETSANDMARKETS. **Docker Monitoring Market by Component**. 2022. Disponível em: https://www.marketsandmarkets.com/Market-Reports/docker-monitoring-market-53974646.html. 27

MARTINS, Nathan S. **Leaky Web App**. 2022. Disponível em: https://github.com/nathanmartins/leaky_webapp. Acesso em: 26 jun. 2022. 45

MENAGE, Paul. **Control Groups**. 2022. Disponível em: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html. Acesso em: 12 jun. 2022. 27

NEWRELIC. **Unified monitoring for your apps and microservices**. 2022. Disponível em: https://newrelic.com/products/application-monitoring. 18

NIEDERMAIER, Sina; KOETTER, Falko; FREYMANN, Andreas; WAGNER, Stefan. On observability and monitoring of distributed systems–an industry interview study. *In*: SPRINGER. **International Conference on Service-Oriented Computing**. [S.l.], 2019. p. 36–52. 18

PAGERDUTY, Inc. **PagerDuty**. 2022. Disponível em: https://www.pagerduty.com/. Acesso em: 26 jun. 2022. 39

PDFMANUALS. **Observabilidade da rede vs. monitoramento: Qual é a diferença?** 2021. Disponível em: https://pdfmanuals.pt/observabilidade-da-rede-vs-monitoramento-qual-e-a-diferenca/. Acesso em: 08 jun. 2022. 30

PEREIRA, Jorge; SILVA, Evaldo O. da; BATISTA, Thais; DELICATO, Flávia C.; PIRES, Paulo F.; KHAN, Samee U. Cloud adoption in brazil. **IT Professional**, v. 19, n. 2, p. 50–56, 2017. 22

SERVICES, Inc Amazon Web. **Observabilidade, obtenha insights e melhore a performance de seus aplicativos, usuários e infraestrutura**. 2022. Disponível em: https://aws.amazon.com/pt/products/management-and-governance/use-cases/monitoring-and-observability. Acesso em: 08 jun. 2022. 29

SINGH, Charanjot; GABA, Nikita Seth; KAUR, Manjot; KAUR, Bhavleen. Comparison of different ci/cd tools integrated with cloud platform. *In*: IEEE. **2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)**. [S.l.], 2019. p. 7–12. 24

TECHNOLOGIES, LLC Slack. **Slack**. 2022. Disponível em: https://slack.com/intl/pt-br/. Acesso em: 26 jun. 2022. 39

THE LINUX FOUNDATION. **repositório prometheus**. 2022. Disponível em: https://github.com/prometheus/prometheus. 39

UBUNTU, Ltd. **Ubuntu**. 2022. Disponível em: https://ubuntu.com/. Acesso em: 08 jun. 2022. 28