UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

ADRIANO LIMA CÂNDIDO

MIGRAÇÃO DE UMA PLATAFORMA DE *OFFLOADING* PARA A ABORDAGEM DE MICROSSERVIÇOS

FORTALEZA

2019

ADRIANO LIMA CÂNDIDO

MIGRAÇÃO DE UMA PLATAFORMA DE *OFFLOADING* PARA A ABORDAGEM DE MICROSSERVIÇOS

Dissertação apresentada ao Curso de Programa de Pós-Graduação em Ciências da Computação do do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciências da Computação. Área de Concentração: Engenharia de Software

Orientador: Prof. Dr. Fernando Antonio Mota Trinta

Coorientador: Prof. Dr. Paulo Antonio Leal Rego

FORTALEZA

2019

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C223m Cândido, Adriano Lima.
Migração de uma plataforma de offloading para a abordagem de microsserviços / Adriano Lima Cândido. – 2019.
85 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2019.
Orientação: Prof. Dr. Fernando Antonio Mota Trinta.
Coorientação: Prof. Dr. Paulo Antonio Leal Rego.

1. Arquitetura de Software. 2. Migração para Microsserviços. 3. Arquitetura Monolítica e Microsserviços. 4. Mobile Cloud Computing. 5. Offloading. I. Título.

CDD 005

ADRIANO LIMA CÂNDIDO

MIGRAÇÃO DE UMA PLATAFORMA DE *OFFLOADING* PARA A ABORDAGEM DE MICROSSERVIÇOS

Dissertação apresentada ao Curso de Programa de Pós-Graduação em Ciências da Computação do do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciências da Computação. Área de Concentração: Engenharia de Software

Aprovada em: 25/10/2019

BANCA EXAMINADORA

Prof. Dr. Fernando Antonio Mota Trinta (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Paulo Antonio Leal Rego (Coorientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Paulo Antonio Leal Rego (Coorientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Gibeon Soares De Aquino Junior
Universidade Federal do Rio Grande do Norte (UFRN)

Prof. Dr. João Bosco Ferreira Filho
Universidade Federal do Ceará (UFC)

AGRADECIMENTOS

Agradeço a Deus por ter me protegido e guidado nessa jornada.

Aos meus familiares, em especial minha mãe Erbênia e meu irmão André, pelo apoio, carinho e dedicação.

Aos meus orientadores, professor Fernando Trinta e professor professor Paulo Rego, pela dedicação nas orientações acerca da temática estudada, na auxílio para resolução dos problemas que surgiram no decorrer do curso e pelo tempo, paciência e confiança que me foram fornecidas.

Aos professores João Bosco e Gibeon Soares, pela disponibilidade em participar banca examinadora, bem como as contribuições e sugestões para melhoria do estudo.

Aos meus colegas e amigos do Little Great, em especial ao Anderson Almada, pelo auxílio e contribuições que foram essenciais na construção desta pesquisa.

Aos meus alunos e orientandos, pela paciência e compreensão nos dias mais atribulados, especialmente aos membros da fábrica de software, projeto o qual sempre levarei no coração.

Agradeço à professora Kerma Márcia, pelas contribuições ao decorrer dessa jornada e pelas palavras que me incentivaram a confiar mais no meu potencial.

E por fim, ao professor José Diener, pela confiança e respeito a mim atribuído, e principalmente por ter me ensinado que a computação não é formada só pelos algoritmos e máquinas, mas também pelas pessoas ao nosso redor. Obrigado meu amigo.

“Se não der certo da primeira vez, chame de versão 1.0.”

(Alan Kay)

RESUMO

Os dispositivos móveis estão se tornando cada vez mais presentes no cotidiano das pessoas. Contudo, a mobilidade proporcionada pelos dispositivos móveis traz consigo suas limitações como menor capacidade de armazenamento e processamento. Apesar da substancial melhoria das novas gerações de *smartphones* e outros dispositivos móveis, a quantidade de informações e a complexidade dos procedimentos delegados a estes dispositivos ainda impõe certas restrições para processamento de certas tarefas, principalmente em relação ao consumo de energia. Isto é especialmente problemático para aplicações móveis sensíveis a contexto, uma classe particular de aplicações móveis que utiliza informações obtidas do ambiente de execução do usuário, para adaptar seu comportamento em prol de benefícios para a experiência do usuário, ou mesmo do funcionamento do dispositivo móvel. Uma das possíveis abordagens para diminuir este problema é a *Mobile Cloud Computing* (MCC). No contexto de MCC, surgem algumas soluções para auxiliar na descentralização do processamento de dados e operações, diminuindo o consumo energético dos dispositivos. Uma delas é a técnica conhecida como *offloading*. Ao longo dos últimos anos, várias plataformas de suporte ao *offloading* tem sido propostas, dentre elas, o *Context Acquisition and Offloading System* (CAOS). Porém, em sua versão atual, o CAOS apresenta problemas causado por sua arquitetura monolítica, tais como, forte acoplamento e falta de escalabilidade horizontal. Estes dois aspectos são fortemente conectados. Para tratar a questão da escalabilidade em software monolíticos, uma abordagem recente que tem recebido muita atenção é o uso de microsserviços. O presente estudo tem como objetivo realizar a migração do CAOS para uma arquitetura de microsserviços, visando alcançar os benefícios que essa arquitetura fornece. Para avaliar a nova versão concebida foram realizados 02(dois) experimentos: um teste de desempenho e outro teste de escalabilidade. O primeiro objetivou verificar possíveis penalizações de desempenho que a arquitetura de microsserviços poderia ter sofrido em relação a versão monolítica. No segundo foi verificado aspectos de escalabilidade proporcionados pela nova versão em microsserviços. Os resultados indicaram que a nova versão, chamada então de CAOS Microservices (CAOS MS), apresenta ganhos de escalabilidade em relação à versão monolítica, sem também comprometer seu desempenho geral.

Palavras-chave: Arquitetura de Software. Migração para Microsserviços. Arquitetura Monolítica e Microsserviços. Mobile Cloud Computing. Offloading.

ABSTRACT

Mobile devices are becoming increasingly present in people's daily lives. However, the mobility provided by mobile devices imposes limitations such as less storage and processing capacity. Despite the substantial improvement of new generations of smartphones and other mobile devices, the amount of information and complexity of new applications created for these devices still impose certain restrictions on processing specific tasks, particularly concerning power consumption. This is a problem, specially for context-aware mobile applications, a particular class of mobile apps that use information gathered from the users' execution environment to adapt their behavior to improve the user experience while using such apps. A promisse approach to mitigate this issue is Mobile Cloud Computing (MCC). In the context of MCC, some solutions emerge to assist in the decentralization of data processing and operations, also reducing the energy consumption of devices. One is the technique known as offloading. Over the last few years, various platforms for supporting offloading have been proposed, among them, the Context Acquisition and Offloading System (CAOS). Currently, CAOS has problems due to its monolithic architecture, such as tight coupling and lack of scalability. These two aspects are strongly connected. A recent approach that has received much attention to address monolithic systems is the use of microservices. The present study aims at proposing the migration of the monolithic version of CAOS into a microservices architecture, and consequently, to achieve the benefits that this architecture provides. This new version is called CAOS Microservices (CAOS MS). We performed two experiments to evaluate the CAOS MS. The former measured possible performance penalties that the microservices architecture could have suffered concerning the monolithic version. The latter verified scalability aspects provided by CAOS MS. Our experiments show us that CAOS MS presents similar performance than its monolithic version, but with improved scalability support.

Keywords: Software architecture. Migration to Microservices. Monolithic Architecture and Microservices. Mobile Cloud Computing. Offloading.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CAOS	*Context Acquisiton and Offloading System*
MCC	*Mobile Cloud Computing*
GREat	Grupo de Redes de Computadores, Engenharia de Software e Sistemas
AMSC	Arquitetura Móvel Sensível ao Contexto
SOA	*Service oriented architecture*
CAOS MS	*CAOS Microservices*
REST	*Representational State Transfer*
API	*Application Programming Interface*
SDLC	*Software Development Lifecycle*
MQTT	*Message Queuing Telemetry Transport*
MBaaS	*Mobile Backend as a Service*
TCL	Teorema Central do Limite
CPU	*Central Processing Unit*
HPA	*Horizontal Pod Autoscaler*

SUMÁRIO

1 INTRODUÇÃO

A presente dissertação apresenta a refatoração de uma plataforma de *offloading* denominada *Context Acquisiton and Offloading System* (CAOS). Essa solução provê um conjunto mecanismos que permitem que dispositivos móveis migrem o processamento de aplicações nativas Android para computadores com maior poder computacional. Em experimentos realizados com o CAOS obteve-se expressivos ganhos no tempo de processamento em relação aos dispositivos móveis, além de viabilizar menor uso de bateria na realização de tais processos. No entanto, aspectos não observados em sua construção resultaram em uma arquitetura monolítica que requer grandes esforços para implantação, evolução e manutenção, com baixo suporte a escalabilidade. Visando propiciar maiores benefícios à solução CAOS, o presente trabalho propõe uma nova versão dessa plataforma baseada na arquitetura de microsserviços.

1.1 Contextualização

Os dispositivos móveis estão se tornando cada vez mais presentes no cotidiano das pessoas. Esses dispositivos possuem mais recursos de processamento, armazenamento, conexão e também capacidade de realizar o sensoriamento cada vez maiores. Através dos aplicativos, estes dispositivos auxiliam os usuários a realizar tarefas de rotina, como envio e recebimento de *e-mails* e mensagens (e.g., *WhatsApp*[1], *Telegram*[2] e *Gmail*[3]), os quais podem ser acessados a qualquer instante e em qualquer lugar. Segundo (KHALIDD, 2017), o número de *smartphones* vendidos ultrapassou o número de *laptops* em todo o mundo. Em vez de usar computadores pessoais, as pessoas estão usando *smartphones* para armazenar a maioria de seus dados pessoais, podendo ser acessados sem grandes esforços. De acordo com a (CISCO, 2019), até 2022 o número de *smartphones* será superior a 50% no total de aparelhos e conexões globais, sendo assim, haverão 11,6 bilhões de dispositivos móveis conectados à internet.

Todavia, a mobilidade proporcionada pelos dispositivos móveis traz consigo limitações. Os dispositivos móveis para serem portáteis precisam ser compactos e alimentados por bateria (SANTOS *et al.*, 2017). Com isso, estes dispositivos também possuem limitações, como menor capacidade de armazenamento e processamento. Apesar da substancial melhoria das novas gerações de dispositivos móveis (e.g., *smartphones*, *tablets* e *smartwatches*), a quantidade

[1] URL: https://whatsapp.com/
[2] URL: https://telegram.com/
[3] URL: https://gmail.com/

de informações e a complexidade dos procedimentos delegados a estes dispositivos ainda impõe restrições no processamento de certas tarefas, principalmente em relação ao consumo de energia. Isto é especialmente problemático para aplicações móveis sensíveis a contexto, uma classe particular de aplicações móveis que utiliza informações obtidas do ambiente de execução do usuário para adaptar seu comportamento em prol de benefícios para a experiência do usuário, ou mesmo do funcionamento do dispositivo móvel. Uma das possíveis abordagens para minimizar este problema é a *Mobile Cloud ComputingMobile Cloud Computing* (MCC).

De acordo com (DINH *et al.*, 2013), a MCC tem por objetivo provisionar um conjunto de serviços equivalentes aos da nuvem, adaptados à capacidade de dispositivos com recursos restritos, de modo a trazer melhorias de desempenho das aplicações ou economia de energia nos dispositivos. No contexto de MCC, surgem algumas soluções para auxiliar na descentralização do processamento de dados e operações, diminuindo o consumo energético dos dispositivos. Uma delas é a técnica conhecida como *Offloading* (KUMAR; LU, 2010). Essa técnica permite que dados e processamento possam ser realizados por dispositivos com maior poder computacional, propiciando maior desempenho e menor consumo energético do dispositivo móvel.

Ao longo dos últimos anos, várias plataformas de suporte ao *offloading* têm sido propostas ((XIA *et al.*, 2014); (CUERVO *et al.*, 2010); (ZHAO *et al.*, 2010); (QIAN; ANDRESEN, 2015); (KEMP *et al.*, 2010); e (CHUN *et al.*, 2011)). Esta dissertação tem especial interesse em um destes estudos: o *Context Acquisition and Offloading System* (GOMES *et al.*, 2017), uma solução desenvolvida pelo Grupo de Redes de Computadores, Engenharia de Software e Sistemas (GREat) da Universidade Federal do Ceará, que permite a implementação de uma Arquitetura Móvel Sensível ao Contexto (AMSC) com suporte ao *offloading* de processamento e dados contextuais na plataforma Android.

Apesar dos ganhos tanto em termos de tempo de processamento, quanto na economia de energia, o CAOS apresenta limitações. O projeto arquitetural da versão inicial do CAOS criou um conjunto de serviços para dar suporte às decisões sobre o *offloading* de dados e processamento. Porém, estes serviços foram implementados com forte interdependência entre eles, criando um sistema monolítico com problemas para manutenibilidade, configuração e implantação. Tais características também levaram a um problema no suporte à escalabilidade do CAOS.

1.2 Motivação

Os módulos de um software monolítico dependem de diversos recursos compartilhados (e.g., memória, banco de dados, arquivos), eles não são independentemente executáveis e escaláveis (DRAGONI *et al.*, 2017a). Para (TAIBI *et al.*, 2017b), os softwares monolíticos não escalam com rapidez e quanto maior a base de código, ficam mais complexos de compreender e manter. Geralmente as arquiteturas monolíticas não são adequadas para suporte a alta volatilidade e velocidade de entrega exigida nos últimos anos, e que para superar essa questão, novos estilos arquitetônicos estão sendo propostos. Segundo (TAIBI *et al.*, 2017b), softwares monolíticos tendem a demorar para serem escalados e a longo prazo tornam-se complexos de manter e compreender.

Nesse sentido, diversas abordagens e técnicas vem sendo aplicadas na criação de componentes e serviços reutilizáveis (e.g., *Service oriented architecture* (SOA)). Em SOA, a reutilização e robustez foi combinada com a obrigatoriedade de interoperabilidade entre os serviços, surgindo a ideia de um serviço como uma entidade de software via comunicação de mensagens, utilizando formatos e protocolos de dados padrão (e.g., XML, SOAP e HTTP) (DRAGONI *et al.*, 2017b). Nesse contexto, compartilhando princípios de design do SOA e trazendo benefícios como desenvolvimento e tempo de comercialização mais rápidos, reação rápida à mudanças, dinamismo, redução de custos econômicos e aplicações mais robustas, surge a arquitetura de microsserviços(RICHARDS, 2016).

Microsserviços é uma arquitetura oriunda da nuvem, através da qual um software pode ser desenvolvido como um conjunto de pequenos serviços. Cada um destes serviços pode ser desenvolvido, implantado e executado de forma independente e em plataformas distintas, possibilitando execução de seus próprios processos. Em geral, microsserviços utilizam mecanismos leves para realizar as comunicações entre eles, como *API RESTFull* (FOWLER; LEWIS, 2014). Dessa forma, os microsserviços podem ser programados em diversas linguagens de programação, viabilizando a evolução, manutenção e armazenamento individual.

Para tratar a questão da escalabilidade em sistemas monolíticos e flexibilizar a evolução e manutenção, diversos sistemas estão sendo construídos e migrados para arquitetura de microsserviços, objetivando transformar partes do software em serviços independentes que interagem entre si para realizar as funcionalidades do sistema (TAIBI *et al.*, 2017b). A arquitetura de microsserviços apresenta diversos benefícios em relação a monolítica, como (*i*) variedade na forma de gerenciamento de disponibilidade e escalabilidade de partes específicas do sistema, (*ii*)

maior capacidade de utilização de diferentes tecnologias, (*iii*) redução do tempo de entrega no mercado e (*iv*) melhor compreensão da base do código (FAN; MA, 2017).

1.3 Objetivos e Contribuições

Objetivando proporcionar uma arquitetura mais flexível e escalável à plataforma de *offloading* CAOS, este estudo tem como objetivo realizar um processo de refatoração da plataforma CAOS, a qual originalmente apresenta uma arquitetura monolítica e com problemas de suporte a escalabilidade e forte acoplamento, para uma arquitetura baseada em microsserviços, visando proporcionar a essa solução alguns benefícios que são possibilitados através da arquitetura de microsserviços, como: (*i*) divisão do sistema em pequenos serviços que possam ser mantidos, evoluídos e desenvolvidos com menor esforço; (*ii*) maior suporte a escalabilidade; (*iii*) independência dos serviços; (*iv*) tolerância a falhas; e (*v*) menor dependência tecnológica (FOWLER; LEWIS, 2014).

Este trabalho apresenta em detalhes o processo de refatoração utilizado para possibilitar a migração da arquitetura monolítica da plataforma CAOS para microsserviços, bem como a nova arquitetura concebida ao término desse processo. Em razão da arquitetura de microsserviços se tratar de um conceito relativamente novo, mencionado pela primeira vez em 2013 por (FOWLER; LEWIS, 2014), diversas técnicas e relatos de experiência vem contribuindo na construção de softwares baseados nesta arquitetura. De acordo com (BALALAIE ABBAS HEYDARNOORI, 2018), ainda existe grande carência de relatos de experiência sobre a migração e o uso de microsserviços na prática. Ressalta-se ainda que existem publicações sobre migrações realizadas pela indústria em blogs, no entanto, estes não possuem tantos detalhes sobre o processo de migração.

Dessa forma, acredita-se que a partir dos relatos de experiência apresentados e decisões arquitetônicas tomadas no processo de refatoração para microsserviços, possam contribuir na migração de outros desenvolvedores e interessados nesse contexto, além de Disponibilizar um conjunto de práticas e técnicas que foram adotadas para possibilitar a construção da solução de *offloading* batizada de *CAOS Microservices* (CAOS MS). Além disso, a partir da revisão de literatura realizada, percebeu-se que não há solução de suporte a *offloading* que tenham utilizado uma arquitetura baseada em microsserviços, nem que tenham investigado os benefícios que arquiteturas que possibilitem alto poder de escalabilidade, possam trazer para o contexto da técnica de *offloading*. Dessa forma, a própria solução concebida torna-se uma contribuição

no contexto da MCC. Além disso, acredita-se que com as diversas possibilidades que podem ser alcançadas com a utilização dessa arquitetura, a sociedade em geral consiga obter maiores benefícios que são oferecidos pelas plataformas de *offloading*.

1.4 Metodologia

A metodologia utilizada na construção desse trabalho, foi organizada seguindo as etapas:

- **Revisão da Literatura:** Através de uma revisão da literatura, foi investigado o estado da arte das arquitetura de software, buscando identificar e analisar as metodologias, desafios e motivações acerca das arquiteturas monolítica e microsserviços. Em conjunto, foram investigados conceitos sobre a *Mobile Cloud Computing*, bem como as práticas e soluções inerentes a técnica de *offloading*, bem como estudo sobre a atual arquitetura da plataforma CAOS.
- **Seleção e Análise dos Trabalhos Relacionados:** Após a revisão de literatura, buscou-se trabalhos que se relacionassem mais especificamente ao contexto de migração do presente estudo. Dessa forma, foram selecionados estudos relatassem processos, técnicas e tecnologias utilizadas nos processos de migração de arquiteturas monolíticas para a abordagem de microsserviços.
- **Processo de Refatoração:** A partir da revisão de literatura e os relatos de migração presentes nos trabalhos relacionados, foram selecionadas as tecnologias, práticas e abordagens que poderiam ser utilizadas na construção da nova solução baseada em microsserviços.
- **Implementação:** Após a definição do processo de migração, a solução CAOS foi refatorada para uma arquitetura baseada em microsserviços, onde, nessa nova arquitetura os componentes presentes no CAOS foram divididos em microsserviços individualmente escaláveis e mantidos.
- **Experimentos:** Com a nova arquitetura baseada em microsserviços, foram realizados testes de desempenho para verificar possíveis penalizações que a nova arquitetura poderia ter trazido. Em seguida, foram realizados testes de escalabilidade em ambas arquiteturas, a fim de verificar quais contribuições a solução baseada em microsserviços poderia trazer em relação a versão monolítica, bem como o seu comportamento diante de um ambiente com uma relativa quantidade de requisições.

1.5 Organização da Dissertação

Esta dissertação dissertação está organizada em 06(seis) capítulos. Este capítulo apresentou uma breve descrição sobre o paradigma *Mobile Cloud Computing* e a técnica *offloading*, bem como os motivos que levaram a migração da solução CAOS para microsserviços. Também são apresentados os objetivo, motivações, contribuições e metodologia que foram utilizadas para construção da presente pesquisa.

O Capítulo 2 apresenta uma revisão de literatura acerca dos conceitos sobre arquiteturas de software, bem como desafios, motivações, técnicas e práticas utilizadas nas arquiteturas monolíticas e microsserviços. Também é apresentado conceitos inerentes a *Mobile Cloud Computing* e *offloading*. Neste capítulo também é apresentada a atual arquitetura da solução de *offloading* a qual é abordada neste estudo.

O Capítulo 3 apresenta trabalhos que objetivaram a migração de softwares com arquiteturas monolíticas para microsserviços, apresentando um relato sobre as técnicas e ferramentas utilizadas nesse processo, bem como os desafios e motivações presentes nas arquiteturas de microsserviços.

O Capítulo 4 apresenta o processo utilizado para refatoração da solução monolítica para microsserviços. Também é apresentada a nova arquitetura concebida e as decisões arquitetônicas tomadas no decorrer do processo de migração.

O Capítulo 5 mostra os resultados obtidos através dos experimentos que objetivaram verificar os aspectos de desempenho e escalabilidade da solução baseada em microsserviços. Mostra também uma comparação entre esses dois aspectos em relação a versão monolítica.

Por fim, o Capítulo 6 descreve os resultados alcançados através do presente estudo, as limitações presentes na solução concebida e lições aprendidas no processo de migração. Além disso, são apresentadas as publicações realizadas a partir dos resultados deste estudo, bem as possíveis evoluções que podem ser adquiridas em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta definições e conceitos relacionados a tópicos que fundamentam o presente estudo. A seção 2.1 apresenta uma visão geral de arquiteturas de software, bem como seus principais elementos. Essa seção também aborda conceitos, motivações e desafios acerca das arquiteturas monolíticas e microsserviços. A seção 2.2 apresenta conceitos sobre a área de *Mobile Cloud Computing*, evidenciando as principais técnicas e desafios presentes nesta área. Por fim, na seção 2.3 é apresentada a arquitetura, características, funcionamento e organização da plataforma de *offloading* CAOS.

2.1 Arquiteturas de Software

Durante a história e evolução da computação, a construção de softwares tornou-se algo bastante complexo. Desde 1970 várias abordagens vem sendo propostas para tratar essa complexidade em diferentes níveis, como "programação estruturada"e a ideia de "integridade conceitual"(VALIPOUR *et al.*, 2009). Nesse contexto, o conceito de arquitetura de software surgiu como um projeto de solução de alto nível para problemas relacionados a complexidade.

Arquitetura de software é a estrutura ou as estruturas do sistema, que é constituída de elementos de software, das características visíveis externamente de tais elementos, e dos inter-relacionamentos entre eles, ou seja, pode ser entendido como a abstração do sistema (CLEMENTS, 2002). Para (PRESSMAN, 2009), arquitetura de software é uma ilustração que permite avaliar a eficácia do sistema comparando-o com os requisitos elicitados, e garantindo estudar alternativas de mudanças, caso necessário, antes que se inicie o desenvolvimento, em um período onde ainda é relativamente fácil e barato corrigir erros que tenham surgido durante o percurso. (PRESSMAN, 2009) também refere-se a arquitetura relatando sobre os benefícios de utilizá-la como um facilitador na comunicação entre os interessados no desenvolvimento de um software. A arquitetura de software evidencia as decisões que terão impacto em todo o desenvolvimento e no sucesso final do processo de construção do software.

Em conformidade, (SOMMERVILLE, 2011) determina como sendo arquitetura de software, um elo crítico entre o processo e os requisitos, pois estipula quais os componentes o software deverá ter e os relacionamentos que existirão entre eles. Na visão de (PFLEEGER, 2004), a descrição segue o mesmo padrão, quando diz que a arquitetura é responsável por associar o que foi identificado na especificação de requisitos e determinar quais os componentes que o

sistema irá possuir, e ainda, é responsável por descrever a intercomunicação entre os mesmos.

O estudo da arquitetura de software é conhecer como os softwares são projetados e construídos. No centro de qualquer software bem projetado há uma arquitetura de software adequada, ou seja, o conjunto de boas decisões de projeto, o que pode ser entendido como a união das principais soluções tomadas para o projeto e realizadas durante o seu desenvolvimento, como também em qualquer evolução do sistema. A arquitetura está em todas as principais propriedades de um software, incluindo seus elementos estruturais, tais como: componentes, conectores e configurações (MEDVIDOVIC; TAYLOR, 2010).

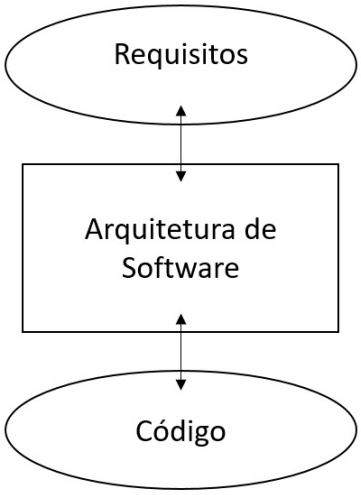
A arquitetura de software pode ser projetada em dois níveis de abstração designados como sendo de pequeno e grande porte. A arquitetura de pequeno porte está preocupada apenas com a arquitetura de um sistema individual, identificando somente os componentes e relações que existirão neste sistema em si, enquanto a de grande porte deve ilustrar a arquitetura de sistemas complexos, onde pode existir interoperabilidade com outros sistemas, ou estar distribuídos em várias empresas. A arquitetura está ligada diretamente aos requisitos não funcionais, determinando o seu desempenho, e por isso é tão importante, pois afeta diretamente a robustez, e também a manutenibilidade (SOMMERVILLE, 2011).

Para (MEDVIDOVIC; TAYLOR, 2010), a arquitetura deve estar em primeiro plano, mais do que os processos, mais do que a análise, e seguramente mais que a programação do sistema. Para qualquer software de tamanho ou complexidade significativa, sua arquitetura deve ser vista e avaliada com antecedência. O autor compara a arquitetura de software com a arquitetura de uma edificação quando menciona que assim como a criação eficiente de um grande edifício requer a observação de sua arquitetura antes da construção, o software também necessita. É importante mencionar que existem arquiteturas boas, ruins, elegantes e deselegantes. Então, a depender do projeto e do sistema que será criado, uma determinada arquitetura pode ser útil para um sistema e para outro não. A finalidade da pesquisa em arquitetura é fornecer técnicas, ferramentas e princípios de design cuja aplicação proporcione de maneira confiável arquiteturas boas e elegantes.

Embora existam inúmeras definições de arquitetura de software. No centro de todas elas está a noção de que a arquitetura de um sistema descreve sua estrutura bruta. Essa estrutura apoia as decisões de projeto de nível superior, incluindo aspectos como a forma como o sistema é composto de partes em interação, onde os principais caminhos da interação e quais são as principais propriedades das partes. Além disso, uma descrição da arquitetura inclui informações

suficientes para permitir análises de alto nível e avaliação crítica. Como pode ser percebido na figura 1, a arquitetura de software normalmente desempenha um papel fundamental como ponte entre requisitos e implementação. Ao fornecer uma descrição abstrata de um sistema, a arquitetura expõe certas propriedades, enquanto oculta outras (VALIPOUR *et al.*, 2009).

Figura 1 – Arquitetura de software como uma ponte



Fonte: Adaptado de (VALIPOUR *et al.*, 2009)

Como destacado, a arquitetura de software é essencial para entendimento sobre o funcionamento e comunicação dos módulos do sistema, minimizando possíveis erros de projeto que possam ocorrer ainda na fase de análise de requisitos. De acordo com (VALIPOUR *et al.*, 2009), a arquitetura de software pode desempenhar papel importante em pelo menos seis aspectos do desenvolvimento de software, sendo eles:

- Compreensão: A arquitetura de software simplifica a capacidade de compreender sistemas grandes, apresentando-os em um nível de abstração no qual o design de alto nível de um sistema pode ser entendido com mais facilidade;
- Reuso: as descrições arquitetônicas suportam a reutilização em vários níveis. O design arquitetônico suporta a reutilização de componentes grandes e também estruturas nas quais os componentes podem ser integrados;
- Construção: Uma descrição da arquitetura fornece um plano parcial para o desenvolvimento, indicando os principais componentes e dependências entre eles;
- Evolução: a arquitetura de software pode expor as dimensões ao longo das quais se espera que um sistema evolua. Ao tornar explícitos os componentes e partes do software, os mantenedores do sistema podem entender melhor as ramificações das mudanças e, assim,

estimar com mais precisão os custos das modificações.

- Análise: As descrições arquitetônicas fornecem novas oportunidades para análise, incluindo verificação de consistência do sistema, conformidade com restrições impostas por um estilo arquitetônico, conformidade com atributos de qualidade, análise de dependência e análise específica de domínio para arquiteturas construídas em estilos específicos.
- Gerenciamento: A experiência mostrou que projetos bem-sucedidos veem a conquista da arquitetura viável de software como um marco importante em um processo de desenvolvimento de software industrial. A avaliação crítica da arquitetura geralmente leva a um entendimento muito mais claro dos requisitos, estratégias de implementação e riscos potenciais

Dentre as características mais importantes de uma boa arquitetura de software está a modularidade. Para (RICHARDSON, 2018), modularidade significa projetar os componentes da aplicação separadamente e independentemente. Segundo (LINDVALL *et al.*, 2002), modularização é fundamental em arquiteturas de software. A proposta da modularização é dividir o sistema em pequenas partes, proporcionando um sistema mais legível e com maior manutenibilidade. Para (BALDWIN; CLARK, 2000), uma arquitetura modular multiplica e descentraliza as partes principais de um software, fazendo com que tenha um acréscimo de valor em um projeto que utiliza a modularidade.

Conforme (SANT'ANNA *et al.*, 2007), a complexidade de um projeto de arquitetura de software, em sua maior parte, é proveniente da modularização indevida dos principais componentes do sistema, (e.g., tratamento de exceções e persistência). Para (BALDWIN; CLARK, 2000), um sistema que possua módulos independentes pode ser mantido da forma que está, podendo ser feita a alteração individual ou em todos os módulos do sistema. As decisões de alterações são descentralizadas do ponto de vista de que os projetistas encarregados pelos módulos possam tomar decisões de alteração de forma autônoma. Portanto, um projeto modular diminui a complexidade, simplifica as eventuais mudanças e retorna uma implementação mais fácil ao incentivar o desenvolvimento simultâneo de inúmeras partes de um sistema.

Nos últimos anos, vários modelos e estilos arquiteturais foram propostos (e.g., Cliente-Servidor, Ponto a ponto, *Pipes and filters*, Orientada a Serviços, Monolítica, MVC, Distribuída e Microsserviços) (MARCOS, 1994); (SHAW *et al.*, 1996); (PUTMAN; BARRY, 2000); (TRIEBEL, 2000); e (BUSCHMANN, 1996); . Apesar de alguns serem bastante parecidos, todos possuem suas próprias características e aplicabilidades. Este estudo tem especial interesse

em 02(dois) desses modelos arquiteturais, o monolítico e microsserviços. Nas seções 2.1.1 e 2.1.2, são apresentadas características sobre o funcionamento e organização de ambas arquiteturas.

2.1.1 Arquitetura Monolítica

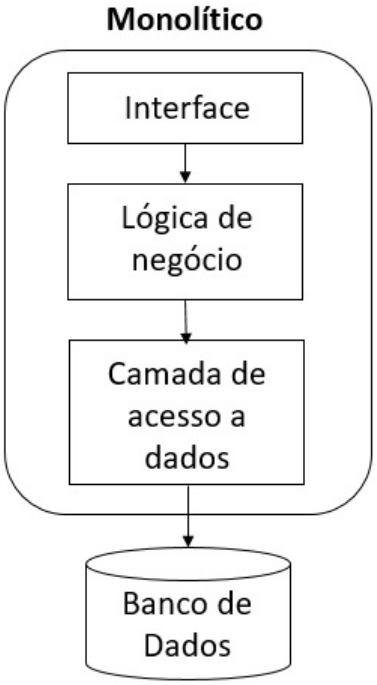
A arquitetura monolítica é considerada como a maneira padrão de começar a desenvolver aplicações. Um software que possui este tipo de arquitetura, tem uma única base de código para atender a vários serviços e interfaces diferentes, como *Representational State Transfer* (REST), *Application Programming Interface* (API) e páginas web. Uma única base de código facilita o desenvolvimento, a implantação e a escalabilidade do aplicativo, desde que o tamanho da base de código seja relativamente pequena. Uma base de código monolítica é uma boa opção no início do projeto devido às qualidades mencionadas acima e porque não há distribuição de código que possa adicionar complexidade (KALSKE, 2019). Em consonância, (PARMAR, 2014) descreve um software monolítico como uma aplicação monolítica tem uma base de código única, podendo possuir vários módulos. Esses módulos são divididos como recursos de negócios ou recursos técnicos, com um único binário executável ou implantável.

Uma aplicação monolítica geralmente usa uma única base para manipular todos os dados. O banco de dados pode ser dimensionado para diferentes partições por *sharding*, mas ainda assim, as partições usam o mesmo esquema. Com um único banco de dados, as transações geralmente são fáceis de manipular, pois a maioria dos sistemas de banco de dados fornece transações ACID (Atomicidade, Consistência, Isolamento, Durabilidade). Os desenvolvedores podem definir facilmente transações e se concentrar mais no fornecimento de novos recursos para os usuários finais. O banco de dados único tem suas limitações. Um sistema monolítico também pode ter vários tipos diferentes de dados. Alguns dados podem ser mais adequados para serem armazenados em um banco de dados NoSQL e outros em um banco de dados relacional. No entanto, com a abordagem monolítica, os desenvolvedores geralmente precisam escolher apenas um mecanismo de banco de dados e usá-lo para todos os tipos de dados(KALSKE, 2019).

Para (JÚNIOR, 2017), o uso da arquitetura monolítica traz consigo vantagens que podem beneficiar o desenvolvimento de sistemas. Tal arquitetura disponibiliza uma maior facilidade na construção da aplicação, devido a assistência proveniente das ferramentas de desenvolvimento tradicionais para esse tipo de arquitetura. Ainda, é percebido que existe facilidade no *deploy* da aplicação, pois a utilização dessa arquitetura, acarreta o desenvolvimento e execução de um único executável.

Nas arquiteturas monolíticas, o desenvolvimento do software pode ser bastante simples no início, mas à medida que o sistema cresce, sua complexidade também aumenta. Uma maneira típica de lidar com a complexidade em uma aplicação que possui uma arquitetura monolítica é dividir o aplicativo em diferentes camadas (FOWLER, 2002). De acordo com (KALSKE, 2019), a abordagem em camadas é amplamente usada em redes e sistemas operacionais. A arquitetura monolítica em camadas exibida na figura 2 é muito conhecida entre os desenvolvedores e todos estão familiarizados com esse tipo de abordagem arquitetural. O software é dividido em uma camada de interface do usuário, uma camada de serviço e uma camada de acesso a dados. A camada de acesso a dados geralmente acessa um banco de dados que lida com todos os dados relacionados a este aplicativo.

Figura 2 – Exemplo comum de uma arquitetura monolítica em camadas



Fonte: Adaptado de (KALSKE, 2019)

Assim como boa parte das arquiteturas escaláveis, a escalabilidade de um software monolítico é feita adicionando novos nós do sistema. Embora isso simplifique a escalabilidade, também limita este recurso. Os componentes do software que precisam de mais recursos computacionais precisam ser escalados juntamente com os componentes que podem preencher sua carga de trabalho com menos recursos. Isso significa custos adicionais para a organização, já que os recursos necessários para instanciação do software crescem de acordo com o tamanho da aplicação. Além disso, (DRAGONI *et al.*, 2017b) afirma que geralmente, no momento em que

arquiteturas monolíticas são submetidas a cargas sucessivas de dados, existe uma dificuldade acentuada em identificar quais dos componentes do sistema estão sendo realmente afetados, visto que a execução do sistema é em um único processo. Isto implica diretamente na escalabilidade, pois mesmo que apenas um único componente esteja sofrendo sobrecarga, todos os módulos do *software* terão que ser escalados. Ainda que for sabido qual o componente está sofrendo sobrecarga, não é possível escalar de forma isolada.

A arquitetura monolítica necessita lidar com circunstâncias de utilização do sistema em que o número de usuários supera a capacidade do servidor. Além do mais, arquiteturas monolíticas são difíceis de gerenciar e manter, em consequência da falta de ferramentas que visem a modularização (FAN; MA, 2017). Para (VILLAMIZAR *et al.*, 2015), escalar sistemas monolíticos é um desafio, tendo em vista que eles frequentemente apresentam diversos serviços, alguns deles mais utilizados que outros. Se os serviços utilizados com maior frequência necessitarem ser redimensionados devido ao fato de serem altamente exigidos, todo o conjunto de serviços também terá que ser redimensionado simultaneamente, o que acarreta que os serviços que não são utilizados com tanta frequência no sistema, passe a consumir grande quantidade de recursos do servidor, mesmo quando não estão em uso.

De acordo com (RICHARDSON, 2018), à medida que o tamanho da base de código aumenta, fica mais difícil adicionar novas funcionalidades e modificar as funcionalidades antigas, porque o desenvolvedor precisa encontrar o local correto para aplicar essas alterações, resultando em ciclos de desenvolvimento mais lentos.

A arquitetura monolítica é indicada na criação de pequenos sistemas, já que apesar da facilidade de implementação, a longo prazo tende a tornar-se complexa de manter e evoluir. Essa abordagem cria grande dependência tecnológica e de acordo com o crescimento do sistema, o tempo necessário para imersão de novos desenvolvedores a equipe aumenta exponencialmente. Para (KALSKE, 2019), uma das desvantagens de uma aplicação monolítica com base de código extensa é que leva muito tempo para se familiarizar com a grande base de código. Para ele, leva tempo para os novos desenvolvedores se atualizarem, pois se sentem perdidos na grande base de código e não conseguem encontrar o local correto para aplicar as alterações. Além disso, os desenvolvedores podem ter medo de refatorar a base de código, pois suas alterações podem afetar vários locais e testar manualmente todos esses locais é grande parte das vezes muito demorada.

Os módulos de um software monolítico dependem de diversos recursos compartilhados (e.g., memória, banco de dados, arquivos), eles não são independentemente executá-

veis (DRAGONI *et al.*, 2017a). Para (TAIBI *et al.*, 2017b) os softwares monolíticos não escalam com rapidez e quanto maior a base de código, ficam mais complexos de compreender e manter. Ressaltam que geralmente as arquiteturas monolíticas não são adequadas para suporte a alta volatilidade e velocidade de entrega exigida atualmente, e que para superar essa questão, novos estilos arquitetônicos foram propostos nos últimos anos.

Existem diversas abordagens que podem ser aplicadas na criação de componentes e serviços reutilizáveis. *Service oriented architecture* (SOA) pode ser visto como um exemplo, onde a necessidade de reutilização e robustez foi combinada com a obrigatoriedade de interoperabilidade entre os serviços, surgindo a ideia de um serviço como uma entidade de software via comunicação de mensagens, utilizando formatos e protocolos de dados padrão (e.g., XML, SOAP e HTTP) (DRAGONI *et al.*, 2017b). Outra maneira de desenvolver softwares com alto poder de manutenibilidade, reusabilidade e escalabilidade, e que recentemente tem ganhando muito espaço entre os desenvolvedores e instituições é a arquitetura baseada em microsserviços (FOWLER; LEWIS, 2014).

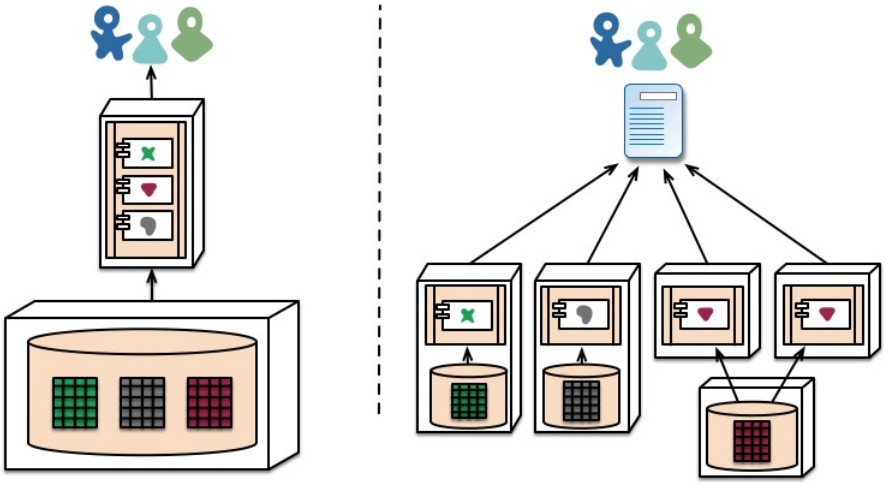
2.1.2 Arquitetura de Microsserviços

Para (SOUSA *et al.*, 2016), o estilo arquitetônico de microsserviços é cada vez mais utilizado na computação em nuvem. Essa abordagem promove a decomposição de aplicações monolíticas em pequenos serviços que podem ser independentemente escalados, otimizando a utilização dos recursos. Em consonância, (THÖNES, 2015) diz que os microsserviços representam uma pequena aplicação que possibilita o desenvolvimento, implantação, elasticidade e teste de forma independente. Relata também, que um microsserviço deve possuir uma única responsabilidade executada de forma independente, possibilitando um código base simples e de fácil compreensão. Na figura 3 é apresentada uma breve comparação de como os serviços de uma aplicação monolítica atua em uma arquitetura baseada em microsserviços. Cada módulo atua de forma independente e possui suas próprias responsabilidades, podendo possuir base de dados individual ou compartilhada.

De acordo com (LEVCOVITZ *et al.*, 2016), o padrão arquitetural "microsserviços" é uma abordagem interessante para o desenvolvimento incremental de aplicativos corporativos. Os autores defendem que novas funcionalidades podem ser desenvolvidas como microsserviços ao invés de incrementar novos módulos na base do código monolítico, diminuindo o esforço necessário em uma futura migração para a arquitetura de microsserviços. Além disso, os

componentes existentes podem ser transformados do monolítico para microsserviços, onde tal processo pode contribuir com a redução do tamanho da aplicação, propiciando um código menor e fácil de manter.

Figura 3 – Diferença entre os componentes de uma aplicação monolítica e microsserviço



Fonte: (FOWLER; LEWIS, 2014)

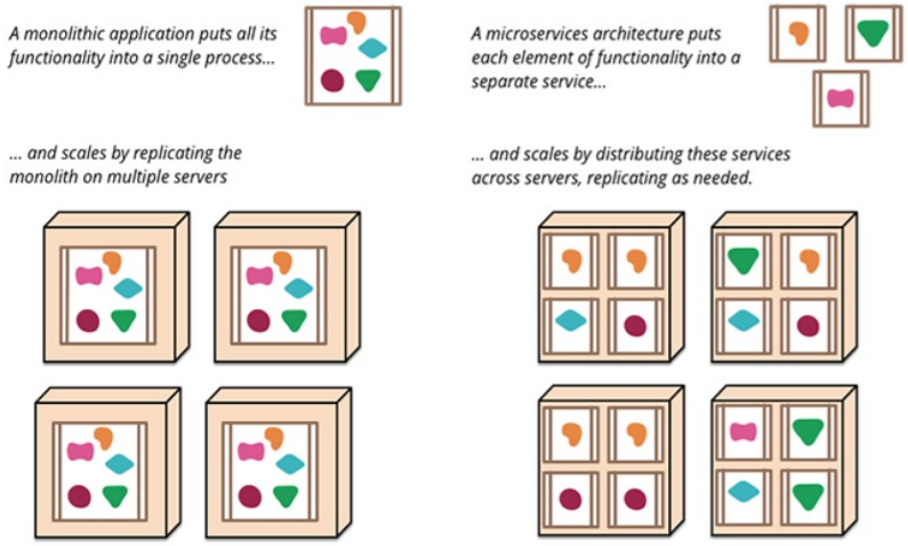
Devido ao seu tamanho reduzido e ao foco em apenas um contexto do negócio, os microsserviços possibilitam uma boa modularidade em sua base de código. A modularidade facilita o desenvolvimento porque faz alterações em um módulo independentes de outros módulos. A modularidade é facilmente mantida em microsserviços, porque existem limites claros entre cada um dos serviços. Um microsserviço pode apenas ver a interface de outros microsserviços, impedindo chamadas para métodos internos de outros microsserviços. Isso significa que a violação acidental da modularidade é mais difícil e manter a modularidade clara não requer disciplina dos desenvolvedores (KALSKE, 2019).

Ainda de acordo com (KALSKE, 2019), os microsserviços devem sempre cumprir o SRP (do inglês, *Single Responsibility Principle*). SRP significa "reunir as coisas que mudam pela mesma razão e separar as coisas que mudam por razões diferentes"(FOWLER, 2009). Com a arquitetura de microsserviço, o esforço para manter o SRP é consideravelmente menor, porque o estilo da arquitetura incentiva a criação de pequenas unidades. A modularidade também ajuda com o SRP, porque quando os limites são claros, é improvável a quebra acidental do SRP. Ao desenvolver microsserviços, o objetivo é ter serviços que sejam fracamente acoplados e altamente coesos. Um fraco acoplamento significa que os serviços não devem possuir dependências de outros serviços (PAPAZOGLOU, 2003). Isso é alcançado com os microsserviços, pois eles se comunicam apenas por meio de interfaces fornecidas por cada microsserviço. A coesão pode

ser descrita como a responsabilidade única dos recursos relacionados em diferentes módulos e/ou classes (BRIAND *et al.*, 1996). Quando os microsserviços são separados corretamente, eles aderem ao SRP e têm um único contexto de negócios no qual operam. Se essas qualidades forem aplicadas aos microsserviços, isso significa que eles também são altamente coesos.

Pode-se perceber na Figura 4, que diferente das arquiteturas monolíticas, a escalabilidade proporcionada pelas arquiteturas de microsserviços, pode ser realizada por determinadas partes do sistema. Os serviços, assim como nas arquiteturas monolíticas, podem escalar horizontalmente, adicionando novas instâncias de serviços ou verticalmente, escalando serviços individuais na mesma instância. O grande diferencial está no tempo com que isso ocorre. Ao invés de fazer um *deploy* inteiro da aplicação, é feito somente da parte que está mais em uso naquele momento, podendo ser desligada após o pico de requisições daquele serviço. As novas funcionalidades que são desenvolvidas, podem ser adicionadas facilmente a arquitetura do sistema, bastando que o novo serviço tenha um mecanismo de comunicação com a API existente.

Figura 4 – Diferença entre escalabilidade de sistemas monolíticos e microsserviços



Fonte: (FOWLER; LEWIS, 2014)

De acordo com (SUN *et al.*, 2015), a arquitetura microsserviços desagrega softwares monolíticos complexos em um conjunto de serviços pequenos e autônomos que trabalham em conjunto. A decomposição permite que diferentes serviços sejam construídos, implantados, gerenciados e evoluídos de forma independente. Ao decorrer dessa desagregação, as chamadas das funções dos componentes são substituídas por comunicações leves entre serviços, podendo ser implementadas através de interfaces *API* bem definidas.

2.1.2.1 Desafios

Apesar das vantagens obtidas com a arquitetura de microsserviços, existem diversos fatores a serem levados em consideração antes de adotar essa arquitetura. Para (BALALAIE *et al.*, 2016), o projeto com microsserviços não é uma bala de prata. Os autores afirmam que a motivação a qual os levaram a migrar para essa arquitetura foi a alta flexibilidade obtida e auxílio das ferramentas *Spring Cloud*(SPRING, 2018) e *Netflix OSS*(NETFLIX, 2018). Contudo, ao adotar microsserviços, várias complexidades foram introduzidas no sistema, exigindo um esforço considerável para resolvê-las (e.g., serviço de registro e descoberta, padronização de protocolos de comunicação e gerenciamento de diversas base de dados).

Para (ESPOSITO *et al.*, 2016), (FAN; MA, 2017), (BALALAIE *et al.*, 2015), (RIBEIRO, 2017) e (TAIBI *et al.*, 2017a), a migração de uma aplicação monolítica para uma abordagem de microsserviços não é nada trivial. Nesse contexto de migração, são encontrados diversos desafios, sendo eles:

- Configuração do ambiente de produção: diferente das arquiteturas monolíticas, a configuração de softwares que utilizam microsserviços não é tão simples. Muitas ferramentas de automação devem ser cuidadosamente configuradas para alcançar os resultados desejados. Isso inclui ferramentas de monitoramento, serviços de descoberta, ferramentas para automatização de *deploy* e integração contínua;
- Redundância de dados: uma das propostas da arquitetura microsserviços é tornar os serviços o mais independente possível. Contudo, quando se trata de dados, fica complicado pensar em um serviço totalmente independente, visto que em algum momento, eles acabam precisando acessar dados de outros serviços. Por essa razão, para garantir essa governança de dados, alguns serviços podem ter informações de outros serviços em sua base dados, ocasionando uma redundância de informações;
- Maior utilização de recursos: Os microsserviços usam várias ferramentas para obter flexibilidade arquitetural, como a descoberta de serviço e o *API gateway*. Isso consome mais recursos e aumenta a complexidade do sistema;
- Desenvolvedores especializados: migrar para microsserviços requer desenvolvedores mais experientes, que devem ser capazes de decompor o sistema e desenvolver novos serviços desacoplados do monolítico, migrar dados para serviços isolados e incluir novos mecanismos de comunicação e orquestração entre serviços que não são necessários para sistemas monolíticos;

– A implantação no ambiente de desenvolvimento é difícil: apesar dos aplicativos terem o código isolado, eles dependem de diversos serviços para o seu funcionamento. Sendo assim, para manter determinado serviço, os desenvolvedores precisam instalar diversas dependências, fazendo com que o processo de manutenibilidade torne-se dispendioso;
– Atualização dos serviços individuais: apesar de ser uma vantagem, o processo de *Deployment* individual é bastante complexo, pois, os serviços ficam espalhados nos servidores na rede. Neste caso, no ato do *deploy*, a ferramenta deve localizar todos os serviços para que seja possível realizar a atualização destes.

Pode-se perceber que existem diversos pontos a serem observados em ambas arquiteturas. Por ser uma área nova, não há uma definição padrão do que pode, ou não, ser migrado para arquitetura de microsserviços. A adoção a essa abordagem, sempre vai depender do contexto e necessidades do software(DRAGONI *et al.*, 2017a).

2.2 Mobile Cloud Computing

Mobile Cloud Computing (MCC), na sua forma mais simples, refere-se a uma infraestrutura em que o armazenamento e o processamento de dados ocorrem fora do dispositivo móvel. Os aplicativos que utilizam a ideia de MCC movem o poder computacional e o armazenamento de dados dos *smartphones* para a nuvem, trazendo aplicativos e *mobile computing* para não apenas os usuários de *smartphones*, mas uma variedade muito mais ampla de dispositivos móveis (DINH *et al.*, 2013). (AEPONA, 2010) descreve a MCC como um paradigma para aplicativos móveis, no qual o processamento e o armazenamento de dados são movidos do dispositivo móvel para plataformas de computação poderosas localizadas na nuvem.

2.2.1 Offloading

Diversos temas vem sendo pesquisados no contexto de MCC, entre esses, o que mais se destaca é o *offloading* (FERNANDO *et al.*, 2013). *Offloading* é uma técnica que provisiona aumento do desempenho e redução do consumo de energia de dispositivos móveis por meio da migração de processamento e dados de dispositivos móveis para outras infraestruturas, com maior poder computacional e armazenamento. Diferente do modelo tradicional cliente-servidor, em que o *thin client* sempre delega a responsabilidade de realizar o processamento ao servidor remoto. No *offloading*, quando não há conexão entre o dispositivo móvel e servidor, o processamento é

realizado localmente, ou seja, no próprio dispositivo. Quando existe a conexão e existem ganhos na migração dos dados ou processamento, o *offloading* é realizado.

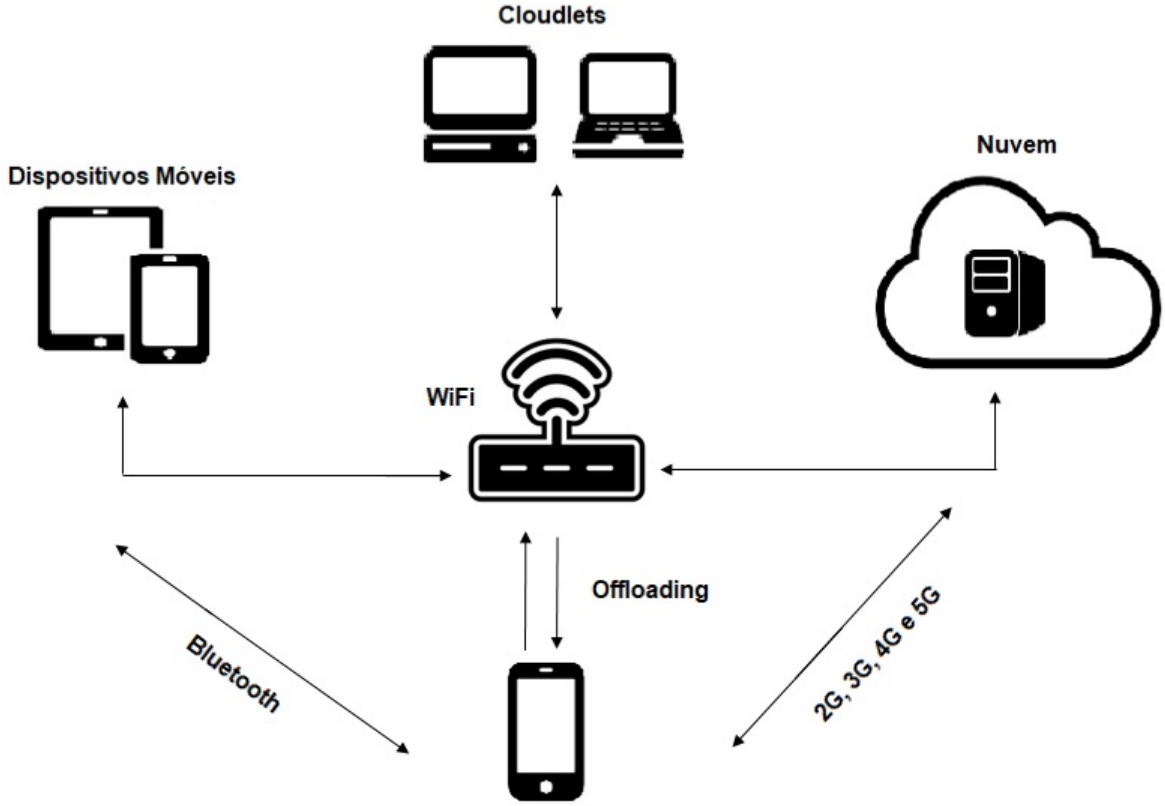
Em geral, existem 02(dois) tipos principais de *offloading*, sendo eles: processamento e dados (FERNANDO *et al.*, 2013). O *offloading* de processamento permite a migração do processamento que seria realizado no dispositivo móvel a outro ambiente de execução (e.g., *desktops*, máquina virtual na nuvem ou outro dispositivo móvel), provisionando a diminuição do consumo energético e aumentando a capacidade computacional. O *offloading* de dados tem por objetivo estender a capacidade de armazenamento do dispositivo móvel. Assim, é possível migrar os dados do dispositivo móvel para um ambiente com maior capacidade de armazenamento, viabilizando também de enviar dados processados de volta para o dispositivo móvel (GOMES *et al.*, 2016).

Em relação ao *offloading* de processos, um aspecto importante é a granularidade do que é migrado para ser executado fora do dispositivo móvel. De acordo com um estudo realizado por (KHALIDD, 2017), atualmente as soluções de offloading possuem diferentes níveis de granularidade, entre elas: *threads*, métodos, classes, partes da aplicação e a aplicação completa. Esse particionamento da aplicação pode ser realizado automaticamente pela infraestrutura de *offloading* ou pode ser transferida a responsabilidade para o desenvolvedor da aplicação, podendo utilizar uma abordagem para marcação (e.g., anotação em Java), para identificar quais componentes são candidatos a serem migrados para o ambiente remoto.

O ambiente remoto de execução é outro aspecto importante. Dentre as opções viáveis, pode-se citar a migração para (*i*) nuvens públicas, (*ii*) para outros dispositivos móveis ou (*iii*) para *cloudlets*. A figura 5 ilustra um típico cenário de *offloading*, apresentando os tipos comuns de ambientes de execução, bem como as formas de comunicação que são comumente utilizadas entre os dispositivos móveis e ambientes.

Conforme ilustrado na figura 5, na execução em plataformas de nuvem, os dispositivos podem utilizar a própria conexão ofertada pelas operadoras de telefonia móvel (e.g., 2G, 3G, 4G e 5G), bem como a conexão via rede *Wi-Fi*. Este tipo de abordagem possibilita que os recursos que são disponibilizados pela nuvem também estejam presentes nos dispositivos móveis, aumentando a gama de recursos disponíveis nestes aparelhos. No entanto, o *offloading* para a nuvem nem sempre é uma boa solução, devido às altas latências presentes na WAN (KHALIDD, 2017). Nesse sentido, os recursos remotos necessitam muitas vezes estar próximos do dispositivo móvel.

Figura 5 – Cenário de offloading



Fonte: Elaborado pelo autor

Os processos de *offloading* também podem ser executados em máquinas locais, ou seja, através de *cloudlets* (SATYANARAYANAN *et al.*, 2009). A utilização de *cloudlets* trás consigo vantagens ao oferecer uma melhor qualidade de serviço e menor tempo para recebimento da requisição e resposta. Esse fato ocorre em razão da velocidade de transmissão de dados dentro de uma rede wi-fi local, onde muitas vezes conta com a conexão direta entre o dispositivo e a máquina de execução do *offloading* (AKHERFI *et al.*, 2018).

Nas execuções em dispositivos móveis, os aparelhos com baixo poder computacional podem otimizar o desempenho na execução de suas tarefas migrando essa responsabilidade para outros dispositivos com maior capacidade dentro da mesma rede. Esse tipo de cenário tem algumas vantagens, como a potencial disponibilidade de uma nuvem móvel, tendo em vista a crescente popularização de dispositivos móveis; e o potencial de escalabilidade da nuvem móvel à medida que mais dispositivos se conectam à rede (SANTOS *et al.*, 2017).

2.2.1.1 Desafios

De acordo com (AKHERFI *et al.*, 2018), os desafios relacionados a MCC também se estendem para a área de *offloading*, abaixo são apresentados alguns desses desafios:

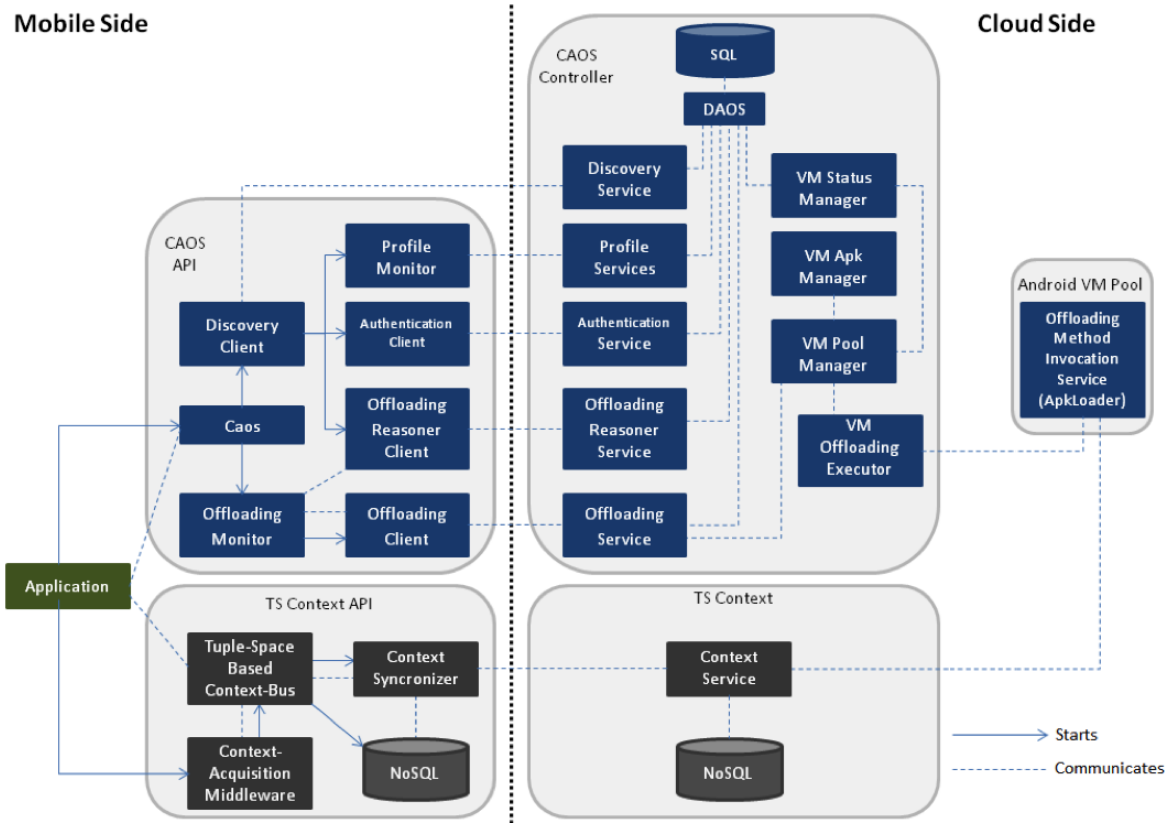
- **Heterogeneidade:** Um dos principais desafios nas atuais plataformas de *offloading* é a diversidade e a heterogeneidade das arquiteturas e sistemas operacionais presentes nos dispositivos móveis. É esperado um acesso consistente aos serviços em nuvem, independentemente do sistema operacional instalado ou do hardware usado. Uma solução de *offloading* compartível com diferentes plataformas de *smartphones* ainda é um problema desafiador no campo da MCC (CUERVO *et al.*, 2010).
- **Segurança e privacidade:** Segurança e privacidade são dois aspectos cruciais e precisam ser mantidos durante o processo de *offloading*. Além de todos os recursos tecnológicos, há um grande aumento na variedade de ataques aos dispositivos móveis, que atualmente é o principal alvo dos invasores. Em relação à segurança na nuvem, as ameaças estão relacionadas à transmissão de dados entre os diferentes nós na rede. Portanto, altos níveis de segurança são esperados pelos dispositivos móveis e pelos provedores de nuvem (DOU *et al.*, 2010).
- **Mecanismos para tomada de decisão:** Diversos fatores podem determinar quando compensa fazer ou não *offloading* de alguma instrução: Taxa de transmissão de rede, alta latência, nível de bateria, tarifas de uso de redes celulares(SANTOS *et al.*, 2017).
- **Serviço de Descoberta e Particionamento:** Em tempo de execução, é desafiador decidir quais componentes do aplicativo podem passar por processos de *offloading*, bem como localizar o servidor o qual executa o serviço adequado para isso. Os algoritmos que respondem a esse problema precisam de um esforço intensivo em recursos, o que pode afetar o tempo de execução do *offloading* (GIURGIU *et al.*, 2012).

A seguir é apresentada a plataforma de offloading CAOS (GOMES *et al.*, 2017), uma solução desenvolvida pelo Grupo de Redes de Computadores, Engenharia de Software e Sistemas (GREat). Essa solução foi utilizada como base para construção da versão baseada em microsserviços apresentada neste estudo.

2.3 CAOS

O CAOS é uma solução que permite o *offloading* de métodos e dados contextuais entre dispositivos móveis para *cloudlets* e nuvens públicas. O CAOS segue uma arquitetura Cliente/Servidor tradicional, na qual dispositivos móveis agem como clientes para serviços executados em infraestruturas de nuvem/*cloudlets*, conforme ilustrado na Figura 6.

Figura 6 – Arquitetura da plataforma CAOS



Fonte: (GOMES *et al.*, 2017)

CAOS é desenvolvido na linguagem Java e baseado na plataforma Android, sendo que os métodos do aplicativo móvel podem ser marcados por desenvolvedores com uma anotação `@Offloadable` para indicar que estes podem ser executados fora do dispositivo móvel. Durante a execução de uma aplicação, o *Offloading Monitor* é o responsável por interceptar as chamadas aos métodos anotados. Neste momento, o CAOS usa uma estratégia de decisão dinâmica baseada em diversas métricas como latência, vazão, número e tamanho dos argumentos do método dentre outras, para decidir se vale a pena ou não migrar o método. Esta decisão é baseada em uma árvore de decisão criada a partir das métricas coletadas pelo *Profile Monitor* e enviadas para a nuvem/*cloudlet* e armazenadas pelo *Profile Service*. Uma árvore de decisão é montada no ambiente da nuvem pelo *Offloading Reasoner Service*, enviada para o dispositivo móvel, e

armazenada localmente pelo *Offloading Reasoner Client*. Este último componente armazena as estruturas de decisão para os métodos anotados em cada aplicativo móvel instalado no dispositivo. Desta forma, a tomada de decisão sobre a realização ou não do *offloading* de processamento é feita localmente no dispositivo, sem necessidade de se consultar um serviço remoto.

Quando a decisão pelo *offloading* é positiva, o *Offloading Client* é o componente responsável por realizar toda a transferência do método e seus argumentos para o ambiente remoto, e depois receber os resultados do processamento. Do lado da nuvem, o *Offloading Service* recebe os pedidos de *offloading* e delega sua execução a uma Máquina Virtual Android gerenciada pelo CAOS. O processo de *offloading* de métodos utiliza um protocolo próprio de RPC (*Remote Procedure Call*) com objetivo de otimizar o tempo necessário para troca de mensagens entre o dispositivo móvel e o servidor de *offloading*.

No CAOS, os métodos migrados são delegados em máquinas virtuais Android que executam máquinas x86 tradicionais. O componente *VM Pool Manager* é responsável por fornecer um ambiente que redireciona as solicitações de *offloading* para uma VM Android adequada. Para executar um método a partir de um aplicativo específico, o CAOS exige que os pacotes de implantação correspondentes (conhecidos como APK) de aplicativos compatíveis com CAOS sejam armazenados em uma pasta especial. O gerenciador *VM Apk* envia todos os arquivos APK para todas as Máquinas Virtuais Android acessíveis listadas nesta pasta especial. O *VM Status Manager* é responsável por monitorar e manter informações (por exemplo, o número de solicitações de *offloading*) de cada máquina virtual gerenciada pelo *VM Pool Manager*. O componente *VM Offloading Executor* é responsável por solicitar a execução do *offloading* em uma VM Android chamando o *Offloading Method Invocation Service*, que é executado na máquina virtual e executa o método desejado.

Os demais componentes do lado cliente são o *CAOS*, *Authentication Client* e *Discovery Client*. O primeiro atua como gerenciador para iniciar os demais serviços da plataforma. Já o segundo usa um mecanismo baseado na comunicação UDP-*Multicast* para descobrir os servidores (cloudlets) em execução na rede local do usuário.

Já em relação ao *offloading* de dados, o CAOS dá suporte a que informações contextuais coletadas dos sensores de um dispositivo móvel possam ser migradas para a nuvem ou cloudlet, e disponibilizada para outros usuários. Para isso, o CAOS propõe políticas de sincronização para determinar quando as informações devem ser enviadas a nuvem/*cloudlet*, e um controle de privacidade para dados contextuais sensíveis que o usuário não deseje compartilhar.

É importante destacar que apesar dos testes relacionados ao consumo de energia realizados com o CAOS apresentarem ganhos neste aspecto, esta não é a principal motivação da solução. A plataforma CAOS centraliza seus esforços no aumento do desempenho em relação a execução nos dispositivos móveis.

Diversos experimentos foram realizados com o CAOS demonstraram como a solução melhora o desempenho de aplicações móveis, auxilia a disseminação de informação contextual e diminui o consumo de energia dos dispositivos móveis(GOMES *et al.*, 2017). Porém, um aspecto não investigado na solução foi sua escalabilidade diante da presença de vários dispositivos móveis. Na realidade, esta questão também não é investigada por boa parte dos trabalhos na área ((XIA *et al.*, 2014); (CUERVO *et al.*, 2010); (ZHAO *et al.*, 2010); (QIAN; ANDRESEN, 2015); (KEMP *et al.*, 2010); (CHUN *et al.*, 2011)).

2.4 Considerações Finais

Este capítulo apresentou definições e conceitos relacionados à arquiteturas de software, com maior foco em arquiteturas monolíticas e arquiteturas baseada em microsserviços. Em seguida, foram apresentadas definições sobre a *Mobile Cloud Computing*, dando ênfase na técnica de *offloading*, além de algumas dos principais desafios e questões de pesquisa relacionadas à essa técnica. Por fim, a plataforma de *offloading* CAOS foi apresentada, essa solução foi utilizada como base para construção da proposta utilizando uma arquitetura baseada em microsserviços.

O capítulo a seguir apresenta os trabalhos relacionados que abordaram a migração de softwares monolíticos para uma abordagem utilizando microsserviços.

3 TRABALHOS RELACIONADOS

Neste capítulo são apresentados alguns dos estudos encontrados na literatura que se relacionam com a proposta desse trabalho. As pesquisas encontradas visaram explanar os relatos de suas experiências na migração de aplicações monolíticas para microsserviços, bem como as abordagens, técnicas e desafios inerentes ao processo de migração. Para isso, na seção 3.1 é apresentado o metodologia que foi aplicada para obtenção das pesquisas relacionadas; na seção 3.2 foi realizada uma sintetização dos estudos relacionados; na seção 3.3 foi realizada uma comparação e discussão entre os trabalhos.

3.1 *Metodologia para Escolha dos Trabalhos Relacionados*

Para localização dos trabalhos relacionados, diversas plataformas e bibliotecas digitais de publicações de pesquisa na área de computação foram utilizadas, objetivando localizar trabalhos que apresentassem técnicas, ferramentas e metodologias que foram aplicadas nos seus trabalhos, para possibilitar a migração de softwares com arquitetura monolítica para microsserviços, bem como estudos que apresentassem relatos sobre o processo utilizado na migração. Na tabela 1, são apresentadas as plataformas/bibliotecas digitais que foram utilizadas para consulta dos estudos que pudessem se relacionar com essa temática.

Tabela 1 – Plataformas e bibliotecas digitais consultadas.

Plataforma/Biblioteca Digital	**Endereço Eletrônico**
IEEE Explorer	https://www.ieee.org
SciELO	https://scielo.org
ACM Digital Library	https://dl.acm.org
Springer Link	https://link.springer.com
Google Scholar	https://scholar.google.com
Science Direct	https://www.sciencedirect.com

Fonte: Elaborado pelo autor

Para realizar as consultas nas plataformas e bibliotecas digitais mencionadas acima, foram utilizadas as seguintes strings de busca:

(*"monolith application"* **OR** *"monolithic application"*) **AND** (*microservices*) **AND** (*"migration to microservices"*) **AND** (*"migration of monolithic applications to microservices"*) **AND** (*"technologies"* **OR** *"techniques"* **AND** *"for building microservices"*).

Como critérios para inclusão dos trabalhos, teve-se: estudos que relatassem processos, tecnologias ou abordagens que foram utilizados de forma prática na migração de aplicações

monolíticas para microsserviços; pesquisas completas e publicadas na língua inglesa; e trabalhos que apresentassem técnicas que foram utilizadas para definição, validação e construção dos microsserviços. Como critérios de exclusão, teve-se: estudos que não apresentassem relatos sobre a experiência de migração; e estudos que não tivessem sido publicados entre 2014 e 2019, já que de acordo com (PAHL; JAMSHIDI, 2016), por se tratar de um conceito novo, ainda não existiam iniciativas concisas na construção de microsserviços. É motivada a aplicação de tais critérios em razão da seleção de processos, técnicas, ferramentas e tecnologias que pudessem contribuir no processo de migração proposto no presente estudo. Após aplicação dos critérios de inclusão e exclusão, foram selecionados 07(sete) estudos relacionados a essa pesquisa, os quais são sintetizados na seção 3.2 e comparados na seção 3.3.

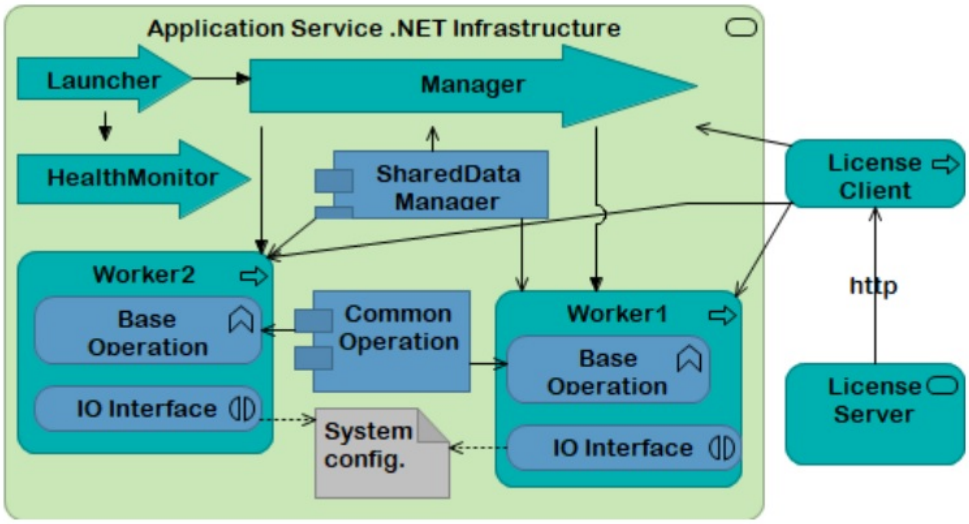
3.2 *Sintetização dos Estudos Relacionados*

3.2.1 Industrial Automation Application

No estudo conduzido pelos autores (SARKAR *et al.*, 2018), foi proposta uma arquitetura para migração de um software monolítico voltado a automação industrial de vários domínios, como automotivas e fábricas químicas e elétricas, para uma arquitetura baseada em microsserviços. Os autores relatam que a aplicação era fortemente acoplada, aumentando consideravelmente o processo de evolução e correção de *bugs*, pois uma mínima alteração impactava em diversas partes da aplicação. A princípio, eles tinham o objetivo de realizar a separação em diferentes contêineres, para isso, dividiriam a aplicação em diferentes módulos. No entanto, conforme a refatoração acontecia, optaram por transformar em uma arquitetura de microsserviços. Conforme ilustrado na Figura 7, a arquitetura monolítica toda a aplicação possuía um único código fonte. Todavia, já era possível perceber alguns serviços que poderiam ser refatorados para atuarem de forma individualizada.

Os autores descrevem que um dos maiores desafios enfrentados foi por conta da aplicação monolítica ter sido construída para atuar em um ambiente *windows*, no entanto, a ferramenta que estavam pretendendo utilizar para conteinerização dos microsserviços (docker), não funcionava da forma esperada neste sistema operacional. Desta forma, a refatoração deveria possibilitar também a compatibilidade do software deles com um ambiente linux. Para realizar a definição dos microsserviços foram realizadas reuniões com pessoas que entendiam do negócio e com aqueles que fizeram parte da construção da aplicação inicial. Após isso, foi iniciado o

Figura 7 – Arquitetura monolítica da aplicação de automação industrial



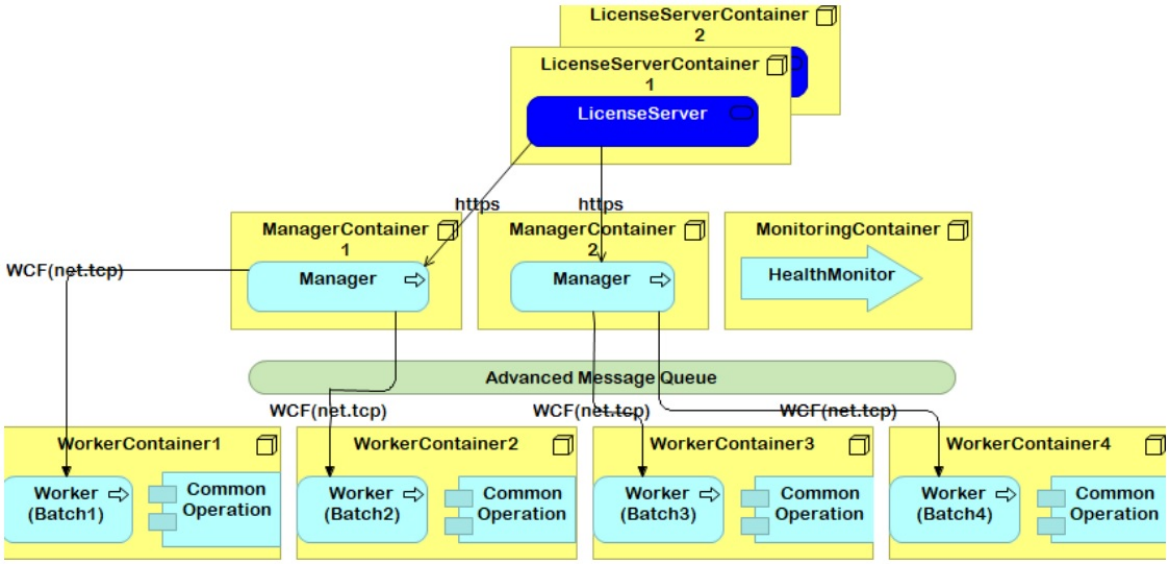
Fonte: (SARKAR *et al.*, 2018)

processo de refatoração e percebeu-se que boa parte do acoplamento presente em sua aplicação, era por conta de bibliotecas externas. Nesse sentido, um forte acoplamento neste aspecto, tornava dificultosa a ideia de separação dos módulos. Outro problema destacado pelos autores, se deu em direção aos estados dos microsserviços. Segundo eles, um microsserviço quando iniciado a partir de recursos de escalabilidade, inicia zerado, ou seja, sem estado algum. A aplicação que foi migrada possuía cálculos que eram divididos em diversos microsserviços, e se a requisição realizada não fosse para o mesmo microsserviços o qual começou a operação, gerava erros em tempo de execução. Em razão disso, autores explicam que arquiteturas baseadas em microsserviços podem perder benefícios como escalabilidade e confiabilidade de acordo com o contexto do software.

Na Figura 8 é ilustrada a arquitetura concebida após o processo de migração para microsserviços. Os autores relatam que os componentes *Worker*, *Manager* e *HealthMonitor* foram refatorados para se tornarem microsserviços. Autores também relatam que utilizaram as tecnologias (*docker* e *docker swarm*(HAT, 2019)) para implementar o ambiente microsserviços. O problema referente as bibliotecas utilizadas foi sanado a partir da disponibilização de todas em um *container* separado, tornando ele também um microsserviço escalável. Para tratar o problema referente ao estado da aplicação, foi implementado um mecanismo para troca de mensagens entre os microsserviços, possibilitando que o estado pudesse ser replicado de acordo com a necessidade.

Por fim, os autores afirmam que a migração de uma arquitetura monolítica para um aplicativo distribuído baseado em microsserviço é possível, desde que o aplicativo seja bem

Figura 8 – Arquitetura microsserviços da aplicação de automação industrial



Fonte: (SARKAR *et al.*, 2018)

modularizado. Aspectos como plataforma de execução também podem impactar no esforço para realizar a migração, bem como particularidades no funcionamento de microsserviços dependentes, principalmente quando relacionados com memória compartilhada e estados. A partir das decisões arquitetônicas tomadas no processo de migração, foi possibilitado ganhos consideráveis de desempenho, escalabilidade e tolerância a falhas, diminuindo o esforço necessário para evolução e manutenção do software.

3.2.2 EasyLearn

Na pesquisa realizada pelos autores (FAN; MA, 2017), foi proposto um processo para migração de aplicações monolíticas para microsserviços. Eles afirmam que apesar de existirem muitos livros e pesquisas que visaram apresentar padrões e práticas voltadas a arquitetura de microsserviços, ainda faltam métodos sistemáticos e controlados por *Software Development Lifecycle* (SDLC) que facilitem a migração para microsserviços. O processo proposto por eles inclui mecanismos para *design*, desenvolvimento e operações. Para isso, os autores utilizaram uma aplicação móvel denominada EasyLearn. Esse aplicativo é utilizado para facilitar a criação, compartilhamento, leitura e discussão de artigos de diversas áreas.

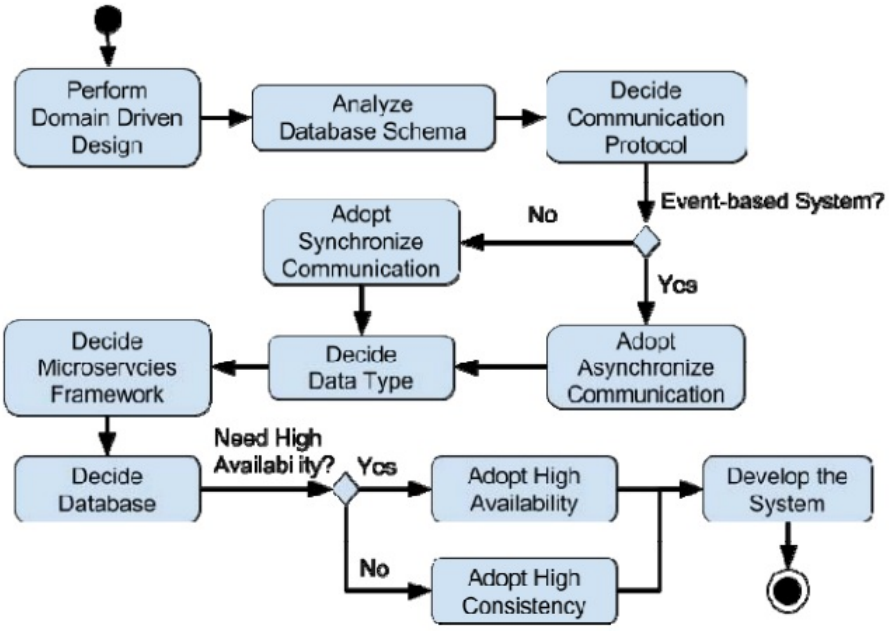
Os autores descrevem que o aplicativo foi projeto e dividido em 03(três) subsistemas, todos monolíticos, sendo: (*i*) uma aplicação android; (*ii*) uma aplicação web; e serviços *RESTfull*. Existiam 02(dois) grandes problemas presentes na versão monolítica, (*i*) a evolução dos esquemas de bancos de dados para adaptação em novas funcionalidades exigia modificação

das funcionalidades relacionadas. Apesar do *design* e interfaces bem definidas, ainda existia alto acoplamento entre os módulos; (*ii*) o serviço que fazia sincronização processava uma grande quantidade de strings, aumentando consideravelmente o consumo de recursos como processador e memória, levando a longas esperas pelo serviços, onde em alguns casos a aplicação também era comprometida. Para superar essas dificuldades, foi proposta a migração do aplicativo EasyLearn para uma abordagem de microsserviços, esperando também que a arquitetura aumentasse a escalabilidade e capacidade de manutenção do sistema.

Para realizar o processo de migração, os autores optaram por uma abordagem iterativa, a fim de manter a estabilidade da versão atual. Segundo eles, essa abordagem possibilitou lidar com problemas que ocorreram durante a migração, como a definição incorreta do serviço, atribuição inadequada de responsabilidade e incompatibilidade de interfaces. Conforme ilustrado na Figura 9, na etapa inicial do processo foi realizada uma análise da arquitetura interna utilizando o *Domain Driven Design*, visando extrair os candidatos a microsserviços da arquitetura original. Na próxima etapa, foi verificado se o esquema de banco de dados era consistente o suficiente para os candidatos a microsserviços, eliminando os candidatos considerados inadequados nesta etapa. Na etapa seguinte, foram extraídos e organizados os códigos referentes aos candidatos a microsserviços mencionados na etapa anterior. Na próxima etapa, foi definido o protocolo padrão de comunicação e a estrutura dos microsserviços. Por fim, todo o código foi refatorado e transformado em microsserviços, onde a comunicação era feita via REST e *Message Queuing Telemetry Transport* (MQTT).

Os autores definem que dois métodos podem ser utilizados na migração de um software monolítico para microsserviços. O primeiro refere-se a aplicação do DDD para analisar vários aspectos da arquitetura do software. De acordo com eles, não existe uma maneira canônica para dividir serviços, podendo variar muito em relação aos requisitos do sistema. O DDD é um método utilizado na análise de sistemas complexos, que possuam as características: (*i*) cada projeto é de um domínio específico; (*ii*) sistema complexos são compostos por vários módulos de domínio; e (*iii*) os problemas do domínio são analisados pelos desenvolvedores em conjunto com os profissionais do domínio. Cada microsserviço deve ser projetado como um componente operável independentemente, o uso do DDD contribui na extração de serviços do domínio, promovendo microsserviços de baixo acoplamento. O segundo método seria analisando a estrutura de banco de dados. Segundo eles, a camada de repositório é geralmente utilizado por diversos módulos, resultando em um alto grau de acoplamento entre os módulos. Para eles, um

Figura 9 – Processo de migração proposto



Fonte: (FAN; MA, 2017)

microsserviço deve ter um banco de dados único, pois o compartilhamento pode impactar no acoplamento da aplicação.

Por fim, são apresentados 03(três) relatos sobre a experiência que obtiveram na migração da ferramenta EasyLearn. O primeiro é em relação a automatização de teste, pois após o melhoramento ou correção de algum serviço, o mesmo pode ser testado individualmente. No segundo relatam que os microsserviços devem ser projetados para lidar com problemas em um único domínio, e a comunicação entre eles deve ser unificada. No terceiro afirmam que os microsserviços são projetos para serem tolerante a falhas. Quando os microsserviços param de funcionar da forma que foram projetados, eles são tratados automaticamente por mecanismos de tratamento de falhas, ao invés de parar o software por completo.

Após os relatos de experiência, eles afirmam que apesar dos benefícios apresentados, as arquiteturas de microsserviços não permitem a configuração de forma simplificada como a de uma arquitetura monolítica, e que são utilizadas muitas ferramentas para tornar possível o ambiente de microsserviços. Além disso, os microsserviços utilizam muitos recursos para possibilitar a flexibilidade dessa arquitetura, como *service discovery* e *api gateway*. Por fim, afirmam que a abordagem utilizado e o estudo realizado pode servir como um *case* para outras equipes de desenvolvimento migrar aplicações semelhantes para microsserviços.

3.2.3 Danske Bank's

No estudo conduzido pelos autores (BUCCHIARONE *et al.*, 2018), é apresentada uma arquitetura que foi elaborada e utilizada na migração de um software bancário denominado Danske Bank's, o qual foi havia sido construído utilizando uma arquitetura monolítica. No entanto, por questões de alto acoplamento e grande esforço na manutenção e evolução, decidiram migrar para uma arquitetura baseada em microsserviços.

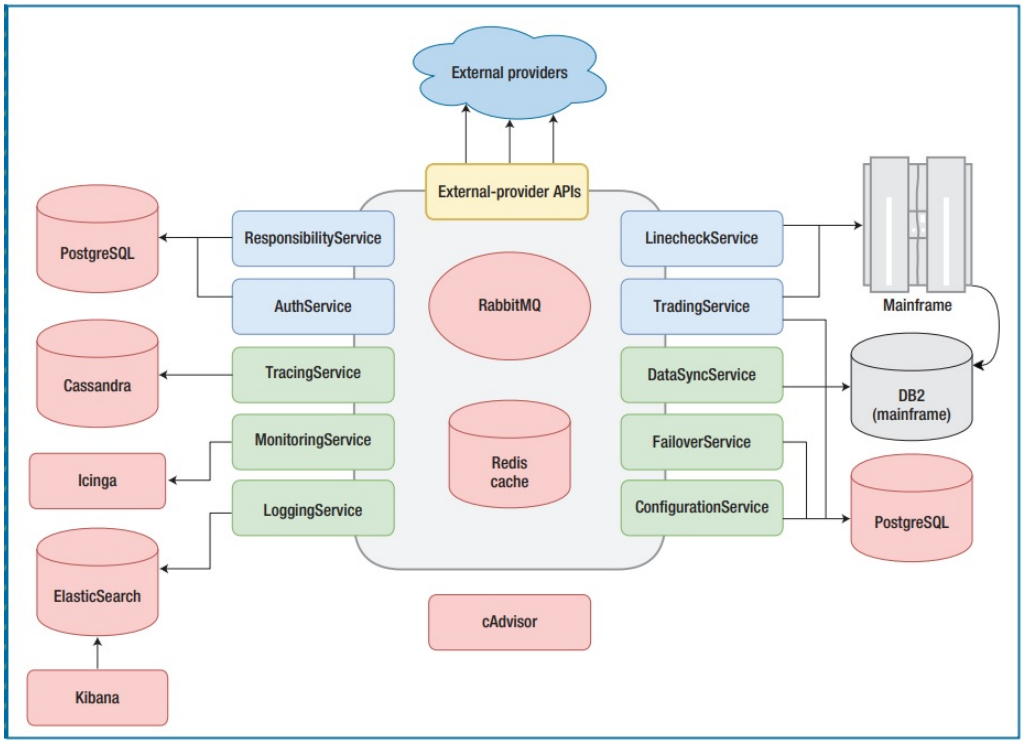
Os autores utilizaram uma abordagem incremental para realizar a migração, ou seja, o desenvolvimento das funcionalidades da versão monolítica ocorreu de forma separada, de modo que possibilitava que a versão antiga continuassem funcionando, enquanto somente alguns módulos estavam sendo substituídos pela versão em microsserviços. Para definir qual seria a ordem das funcionalidades que seriam migradas, os autores realizaram reuniões com os interessados no sistema. A partir destes encontros, foi decidido quais funcionalidades ficariam unidas em um único microsserviço e quais seriam seriam migradas primeiro para a nova arquitetura.

A arquitetura concebida no processo de migração da aplicação bancária Danske Bank's é ilustrada na Figura 11. Segundo os autores, a migração trouxe modernização para o software de forma a possibilitar benefícios tanto a nível de processo como a nível de aspecto arquitetural. Relatam que entre os benefícios adquiridos nesta nova versão, se destacam: processo de integração e implantação contínua dos microsserviços, através do uso de pipelines de automação; alta disponibilidade dos microsserviços obtidos pelo uso de orquestradores dos *containers* (Docker Swarm); e integração e sincronização entre os microsserviços, através da troca de mensagens por coreografia (RabbitMQ).

Os autores relatam que diversos desafios e problemas que enfrentavam com a aplicação monolítica foram resolvidos ou melhorados com o uso da arquitetura baseada em microsserviços, entre eles: a flexibilidade para definição de uma tecnologia específica para resolução de determinado problema que uma funcionalidade/serviço possuía; menor esforço na implantação de serviços ao utilizar a ideia de *containers docker*, bem como a centralização de logs e métricas de monitoramento dos serviços. Para eles, a centralização das métricas e logs de monitoramento permitiram que as equipes de desenvolvimento obtivessem uma melhor percepção acerca dos microsserviços em funcionamento, tendo como consequência uma atuação proativa das equipes, sempre que algum microsserviços estivesse atuando de forma incorreta.

Por fim, os autores descrevem que a arquitetura baseada em microsserviços possui

Figura 10 – Arquitetura Microsserviço da Aplicação Dranske Bank's



Fonte: (BUCCHIARONE *et al.*, 2018)

desafios que as aplicação monolíticas não apresentam, como: maior uso de recursos e esforço na elaboração de um padrão para comunicação. No entanto, relatam que os microsserviços tiveram grande valia e trouxeram grandes ganhos a empresa, visto que conseguiam resolver problemas relacionados ao monitoramento, tolerância à falha e as questões relacionadas a concorrência.

3.2.4 Backtory

Na pesquisa realizada por (BALALAIE *et al.*, 2016), são relatadas as experiências e lições que foram percebidas no decorrer do processo de migração incremental e, também, as lições aprendidas na refatoração e migração da arquitetura da solução Backtory, uma aplicação monolítica baseada em *Mobile Backend as a Service* (*Mobile Backend as a Service* (MBaaS)) para uma arquitetura baseada em microsserviços. A principal motivação para migrar a ferramenta Backtory para uma abordagem de microsserviços foi a necessidade de resolver um problema relacionado ao requisito de disponibilizar um chat como serviço. Além disso, relatam que com esse novo modelo arquitetural, seria possível alcançar aspectos como: reusabilidade no uso dos serviços, governança dos dados de forma descentralizada, implantação automatizada dos microsserviços e melhor suporte a escalabilidade aos microsserviços oferecidos.

De acordo com os autores, para definição e concepção dos microsserviços foram

utilizadas as técnicas *Domain Driven Design* e o padrão de definição de contexto (*Bounded Context*). Oa autores relatam que para realizar a migração foram utilizadas as tecnologias e *frameworks* comumente presentes no contexto de desenvolvimento de microsserviços, como: Spring Boot, Spring Cloud Server e Netflix OSS (NETFLIX, 2018)(SPRING, 2018), fornecendo um conjunto de mecanismos e componentes para implementação da arquitetura microsserviços, como: serviço de registro e descoberta (Eureka), serviço de balanceamento de carga (Ribbon), serviço de *circuit breaker* (Hyxtris) e serviço *edge server* (Zuul).

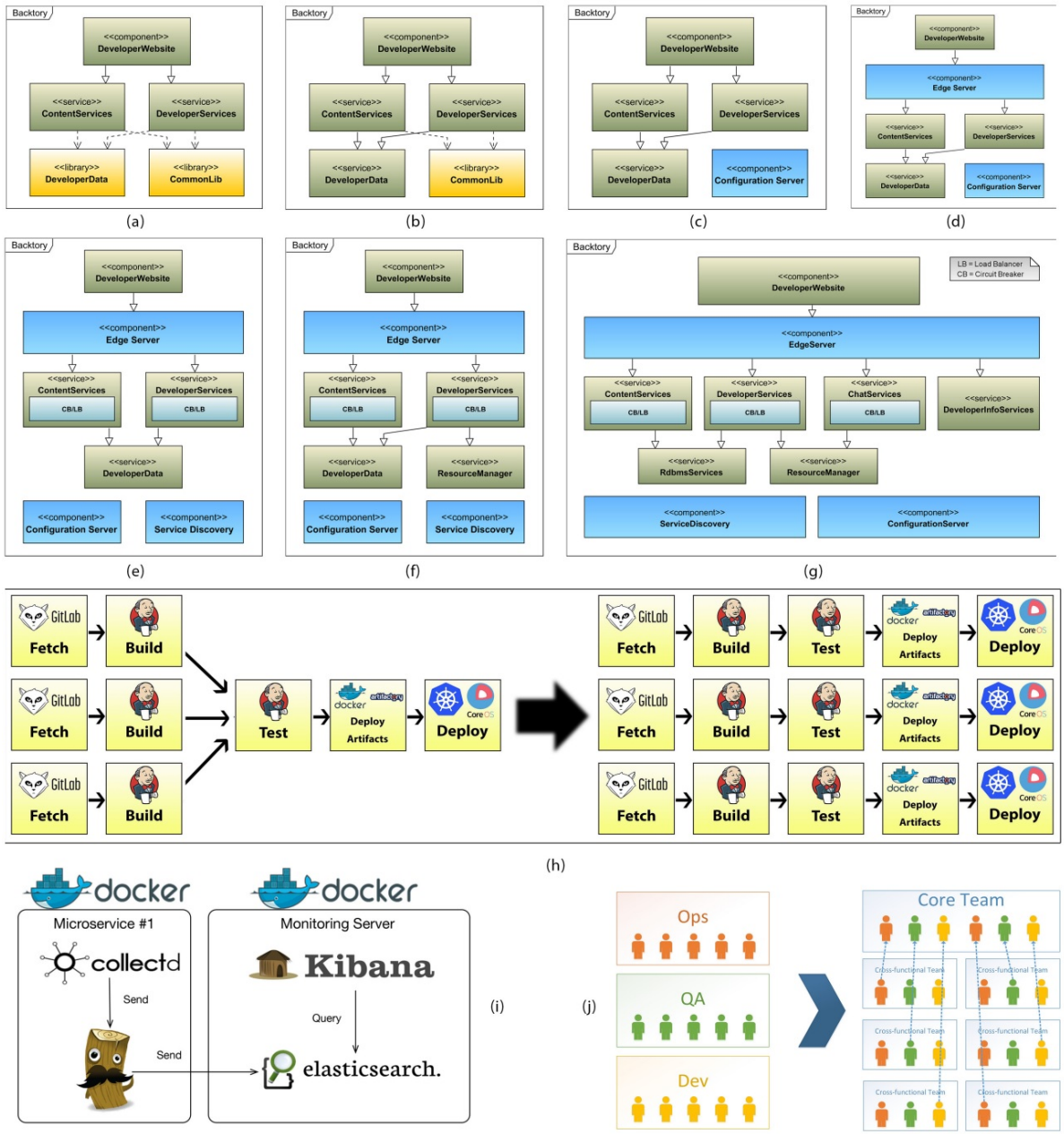
Conforme é ilustrado na Figura 11, após definição das tecnologias e técnicas que seriam utilizadas, os autores definiram 10(dez) etapas para realizar a refatoração e migração da arquitetura monolítica Backtory para microsserviços, sendo eles: preparação do pipeline de integração contínua; transformando o componente DeveloperData para serviço; introdução da entrega contínua; introdução do serviço de Edge Server; introdução de colaboração de serviço dinâmico; introdução do componente ResourceManager; introdução do serviço de chat e componente DeveloperInforService; clusterização dos microsserviços; interconectando a equipe de desenvolvimento.

Por fim, ao finalizar o processo de migração os autores descrevem 05(cinco) lições aprendidas que podem auxiliar a interessados na migração para microsserviços. A primeira lição refere-se ao esforço necessário para implantar microsserviços em ambiente de desenvolvimento, visto a quantidade de ferramentas necessárias para possibilitar este ambiente; a segunda refere-se à manutenção dos contratos do serviço; a terceira refere-se à necessidade de possuir desenvolvedores especializados em computação distribuída/microsserviços; a quarta refere à necessidade de um padrão para definição de candidatos a microsserviços; na quinta e última lição os autores afirmam que microsserviços não são uma "bala de prata"e que para utilizar este tipo de abordagem, diversos desafios e complexidades são enfrentados.

3.2.5 Client Management

No estudo realizado pelos autores (KNOCHE; HASSELBRING, 2018) foi proposto um processo de modernização manual para possibilitar a migração de um software monolítico para microsserviços. Para isso, utilizou-se como base um sistema de gerenciamento de clientes denominado *Client Management* que havia sido desenvolvido inicialmente na linguagem de programação Cobol. Os autores projetaram um processo de 05(cinco) etapas para realizar a modernização, conforme ilustrado na Figura 12.

Figura 11 – Etapas para migração do software Backtory

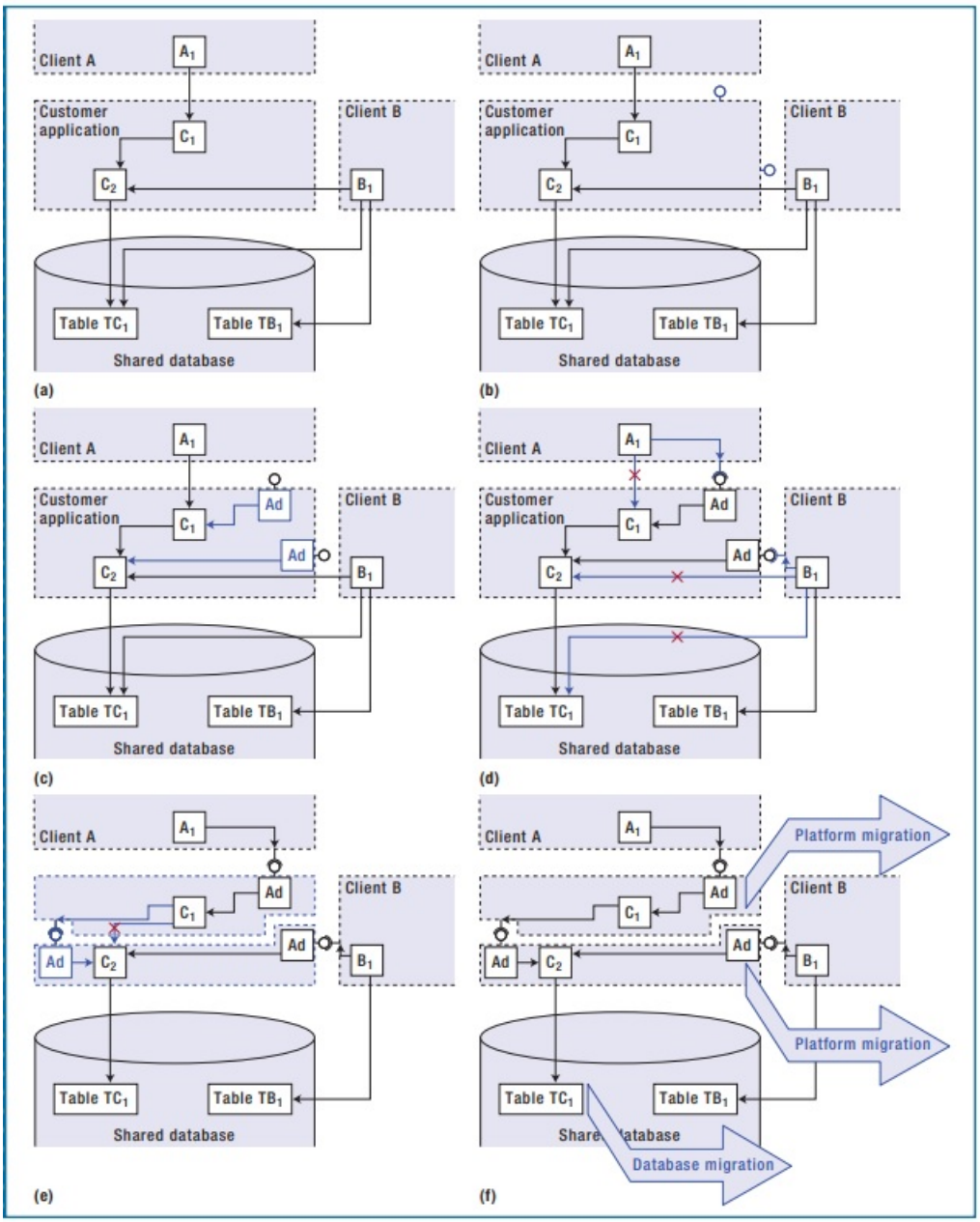


Fonte: (BALALAIE *et al.*, 2016)

Na primeira etapa foi definido um serviço de fachada externo, possibilitando definir as operações dos serviços que serão disponibilizados para as aplicações clientes consumirem. Na segunda etapa foram implementados os serviços de fachada, objetivando garantir que os serviços de fechada possuíssem o mesmo comportamento das operações existentes na aplicação monolítica. Nessa etapa ainda foram realizados testes para garantir que os microsserviços teriam o mesmo resultado da aplicação monolítica. Na terceira etapa foi realizada a migração das aplicações clientes, objetivando a utilização dos serviços implementados, deixando de usar os recursos da aplicação monolítica (e.g., banco de dados) e diminuindo os pontos de entrada da

aplicação monolítica.

Figura 12 – Processo para migração e modernização da arquitetura monolítica



Fonte: (KNOCHE; HASSELBRING, 2018)

A quarta etapa do processo buscou estabelecer os serviços de fechada internos, conforme ilustrado na Figura 12. O objetivo dessa etapa era organizar a aplicação monolítica internamente, fazendo com que os módulos acessassem os serviços de fachada interno. A última etapa objetivou a troca das implementações dos componentes de fachada por microsserviços. Sendo assim, os serviços que outrora haviam sido separados, foram transformados em microsserviços.

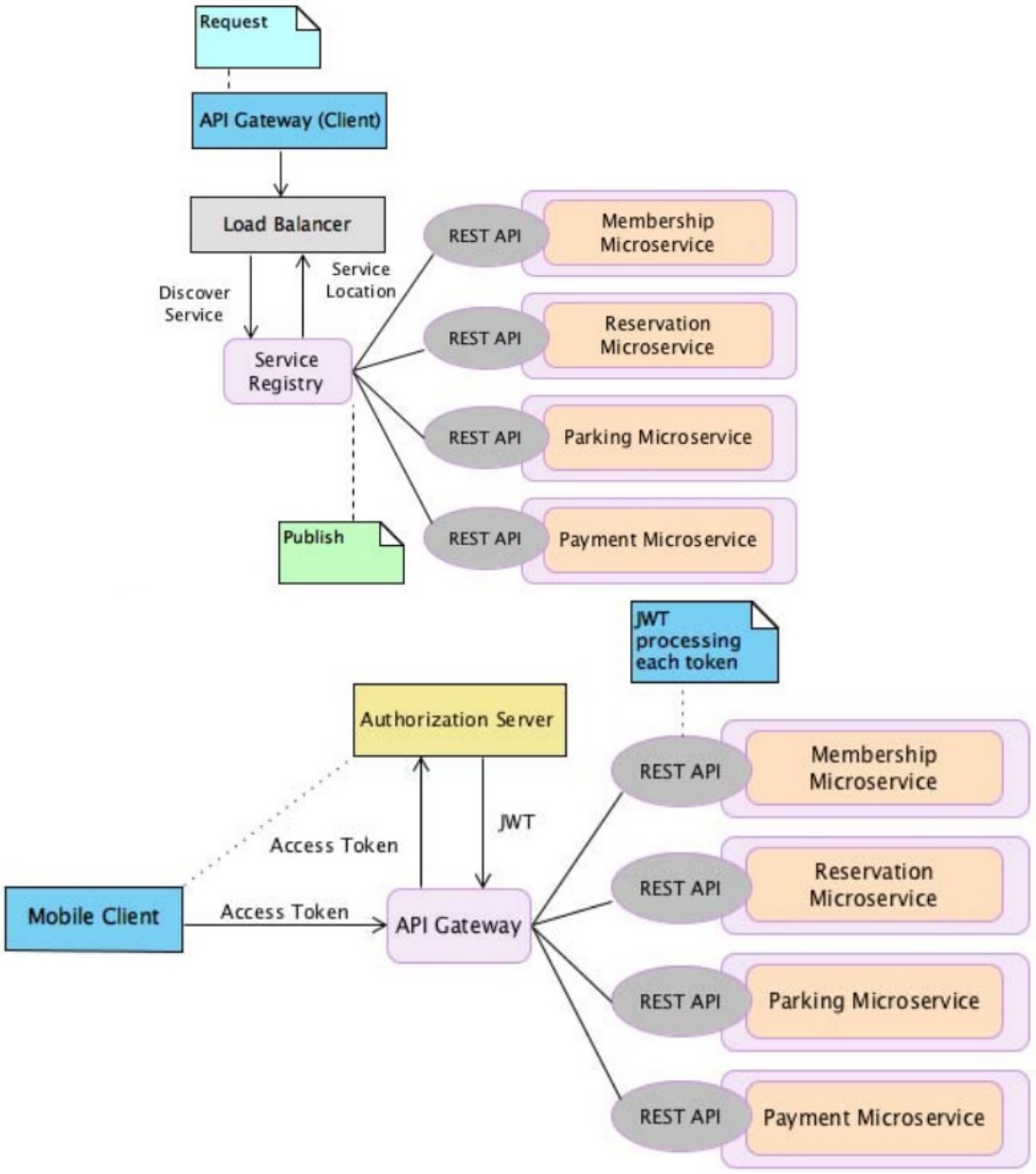
Por fim, os autores descrevem que o processo de migração da aplicação monolítica foi migrada para microsserviços dentro de um período de 04(quatro) anos. A aplicação foi dividida em 23(vinte e três) microsserviços. Relatam que todos foram desenvolvidos utilizando Java, fazendo uso também das tecnologias docker e *docker compose* para construção do ambiente de microsserviços. Apesar do esforço realizado, algumas funcionalidades continuaram em uma versão monolítica. Esse fato ocorreu por conta de problemas relacionados a controle de transação em um ambiente distribuído. O trabalho também indica que o controle de transação distribuída é um dos principais desafios e ainda precisa ser estudado nas arquiteturas de microsserviços, já que diversos aplicações precisam garantir a integridade dos seus dados.

3.2.6 Vacancy Reservation

Na pesquisa realizada pelos autores (YUGOPUSPITO *et al.*, 2017), foi proposta uma arquitetura para servir como base para migração de uma aplicação monolítica para uma arquitetura baseada em microsserviços. O software migrado se tratava de uma aplicação voltada a reserva de vagas para um estacionamento de supermercado. Os autores tinham como premissa possibilitar melhor suporte a escalabilidade e viabilizar a evolução e evolução com menor esforço. A aplicação em questão possuía quatro domínios principais: Sócios, Reserva, Estacionamento e Pagamento, armazenando suas informações em uma única base de dados. Com objetivo de construir uma aplicação menos acoplada, foi proposto dividir todos esses domínios em microsserviços independentemente mantidos e evoluíveis. Conforme ilustrado na Figura 13, optou-se por cada banco de dados possuir sua própria base de dados, e toda a comunicação entre eles ocorreria através de *APIs REST*.

De acordo com os autores, os serviços foram implementados seguindo boas práticas presentes na engenharia de software, como: segregação de interfaces; princípio da responsabilidade única; e utilização da metodologia *Domain Driven Design* para definição dos microsserviços. No início do desenvolvimento das integrações entre os serviços, foram identificados alguns problemas para comunicação direta via *API REST*, pois essa comunicação realizada de forma direta, estava gerando novamente um alto acoplamento, podendo dificultar a manutenção a medida que novos microsserviços fossem desenvolvidos ou evoluídos. Nesse sentido, foi implementado um *API Gateway* para possibilitar uma única forma de entrada para os clientes que desejassem consumir os microsserviços. Através disso, afirmam que resolveram os problemas inerentes à segurança e monitoramento dos microsserviços, diminuindo também o acoplamento apresentado

Figura 13 – Arquitetura microsserviços utilizada no sistema de reserva de vagas



Fonte: (YUGOPUSPITO *et al.*, 2017)

anteriormente.

Para possibilitar uma maior segurança entre os microsserviços e os dados que estavam sendo consumidos, os autores construíram uma estrutura utilizando protocolos de autorização *OAuth2* e *OpenID*. Tais protocolos foram utilizados para viabilizar a segurança no consumo dos dados via protocolo HTTP, sendo expostos de forma segura pelas *API REST* dos microsserviços. Conforme ilustrado na Figura 13, os autores utilizaram o serviço de registro e um balanceador de carga para permitir que os microsserviços fossem escalados automaticamente, de acordo com os níveis de demanda aos microsserviços. Para isso, foram utilizadas as ferramentas docker e

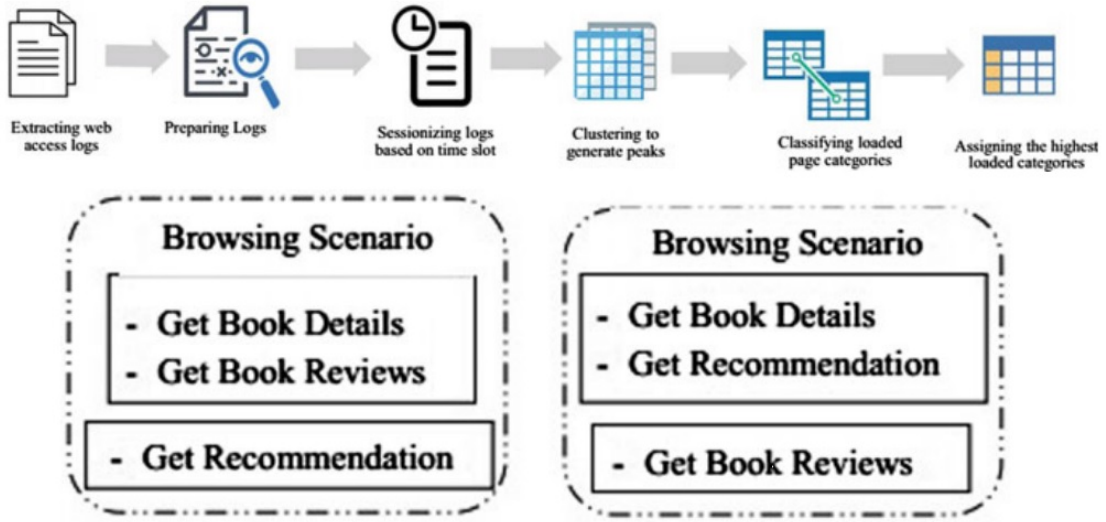
kubernetes. Por fim, concluem que a migração da arquitetura monolítica para microsserviços, simplificou o processo de negócio, facilitando também o processo de manutenção e evolução do software.

3.2.7 Micro-BookShop

No estudo realizado pelos autores (MUSTAFA *et al.*, 2018), foi apresentada a decomposição de uma arquitetura monolítica com intuito de gerar um processo manual que possibilite a definição de microsserviços. A técnica construída e intitulada por eles de Gran-Micro objetiva encontrar uma melhor granularidade de microsserviços a partir de logs de acesso a uma determinada aplicação web denominada Micro-BookShop. O processo proposto é composto por 06(seis) passos.

No primeiro é realizada uma mineração dos logs de acesso ao servidor web onde a aplicação é executada. No segundo passo são removidas as URLs inválidos dos logs de acesso. No proximo passo devem ser encontrados os principais períodos de acesso, localizando os períodos de tempo que ocorreram a maior sobrecarga da aplicação. O quarto passo agrupa e define os períodos de tempo de acesso, gerando os picos de acesso em um determinado período. Conforme ilustrado na Figura 14, no quinto passo são localizadas e classificas as páginas da aplicação que foram mais acessadas de acordo com o pico estruturado da etapa anterior. Por fim, no sexto e último passo é gerada uma recomendação de possíveis microsserviços, utilizando como base o maior percentual de acesso.

Figura 14 – Processo proposto para decomposição do software Micro-BookShop



Fonte: (MUSTAFA *et al.*, 2018)

No estudo é relatado que para chegar a granularidade ideal de microsserviços, os requisitos funcionais (e.g., desempenho) devem ser levados em consideração. Os autores alertam dizendo que a boa parte das técnicas utilizadas em decomposições de microsserviços apresentam um modelo de domínio, não observando aspectos relacionados a desempenho. Ainda no estudo, também é feita uma comparação utilizando a técnica proposta por eles e sem a técnica proposta. Na Figura 14, são ilustrados ambos cenários, o que fica localizado na esquerda ilustra a decomposição da aplicação deles sem a técnica proposta e o segundo ilustra a decomposição utilizando a técnica proposta por eles.

Os autores concluem afirmando que a técnica Gran-Micro apresenta 02(dois) aspectos bastante positivos. O primeiro é não utilizar um modelo de negócio específico e o segundo é levar em consideração requisitos não funcionais como o de desempenho. Por fim, finalizam com a proposta de utilizar o mesmo processo em uma aplicação web de grande porte.

3.3 Comparativo Entre os Trabalhos

A partir dos trabalhos coletados e analisados é possível perceber que apesar da área de atuação dos softwares serem completamente distintos, a motivação o qual levaram eles a migrar suas aplicações monolíticas para microsserviços são bastante semelhantes. Os autores (SARKAR *et al.*, 2018); (FAN; MA, 2017); e (BUCCHIARONE *et al.*, 2018) relatam que uma das principais motivações o qual lhes levaram a migrar para essa arquitetura foi o forte acoplamento de seus sistemas e a necessidade de evolução com menor esforço. Enquanto os autores (BALALAIE *et al.*, 2016); (KNOCHE; HASSELBRING, 2018); e (YUGOPUSPITO *et al.*, 2017) descrevem que a motivação além da facilidade para evolução, foi possibilitar melhor suporte à escalabilidade ao seus sistemas. Os autores (FAN; MA, 2017) e (MUSTAFA *et al.*, 2018) procuraram possibilitar melhor desempenho em determinadas funcionalidades do seu software.

Através do relato de experiência apresentado pelos autores, pode-se coletar diversas contribuições para interessados em migrar para a abordagem de microsserviços. Entre as principais contribuições, podem ser citados: 05(cinco) processos de migração e (03) três modelo de arquitetura microsserviços. Os estudos apresentados também mostram que a plataforma onde o software é executado o software também pode variar (e.g., Móvel, Desktop e Web). Todos apresentam a refatoração de pelo menos um tipo de plataforma, com exceção dos autores (FAN; MA, 2017), que apresentam a refatoração dos três tipos de plataforma citados. As

tecnologias utilizadas também são bastante semelhantes, diferindo algumas vezes por conta da especifidade de cada software. Entre as tecnologias mais utilizadas para possibilitar a criação de contêineres e construção de um ambiente microsserviços, pode ser citado o Docker e as ferramentas NetflixOSS.

A forma como os microsserviços foram definidos também variam em alguns trabalhos, todavia, existe certa semelhança. Os autores (YUGOPUSPITO *et al.*, 2017); e (BALALAIE *et al.*, 2016); propõem a definição de microsserviços utilizando a técnica DDD. Os autores (FAN; MA, 2017) também propõem o uso do DDD, todavia, utilizam outra abordagem que consiste na análise das estruturas do banco de dados da aplicação. Os autores (MUSTAFA *et al.*, 2018) propõem uma abordagem de definição a partir de observação de logs de acesso gerados pelo software. Já os autores, (SARKAR *et al.*, 2018); (BUCCHIARONE *et al.*, 2018); e (KNOCHE; HASSELBRING, 2018); partiram para uma abordagem empírica, ou seja, os microsserviços foram definidos a partir de reuniões com os envolvidos na construção e/ou negócio do software.

No quadro apresentado na Figura 15, são apresentadas as principais caraterísticas dos estudos encontrados, bem como uma breve comparação entre as diversas abordagens e tecnologias utilizadas em seus contextos de migração.

Figura 15 – Principais características dos trabalhos relacionados

Trabalho	Área de Atuação do Software	Principal Contribuição	Plataforma	Técnica para Definição dos Serviços	Tecnologias
(SARKAR et al., 2018)	Automação Industrial	Arquitetura	Desktop	Empírico	Docker e Docker Swarm
(FAN; MA, 2017)	Gerenciamento de Trabalhos Científicos	Processo	Móvel, Desktop e Web	DDD e Análise das Estruturas do Banco	Docker, Docker Compose, Grafana e Kubernetes
(BUCCHIARONE et al., 2018)	Aplicação Bancária	Arquitetura	Web	Empírico	Docker, Docker Swarm e RabbitMQ
(BALALAIE et al., 2016)	Softwares como Serviço	Processo	Móvel	DDD	Spring Boot/Cloud e NetflixOSS
(KNOCHE; HASSELBRING, 2018)	Gerenciamento de Clientes	Processo	Web	Empírico	Docker e Docker Compose
(YUGOPUSPITO et al., 2017)	Supermercado	Arquitetura e Processo	Web	DDD	NetflixOSS, Docker e Kubernetes
(MUSTAFA et al., 2018)	Venda de Livros	Processo	Web	Observação de Logs	NetflixOSS, Docker e Kubernetes

Fonte: Elaborado pelo autor

As abordagens, técnicas, tecnologias e relatos apresentado nos trabalhos, contribuíram na migração da plataforma CAOS. A partir deles que foi definida a forma como os microsserviços seriam definidos (DDD e definição empírica), bem como tecnologias que seriam utilizadas para possibilitar um ambiente de microsserviços (Docker e Kubernetes).

3.4 Considerações Finais

Este capítulo apresentou os trabalhos que se relacionavam com o objetivo do presente estudo. Para tanto, definiu-se a metodologia para seleção dos trabalhos, bem como as bases de dados utilizadas. Em seguida, os estudos foram sintetizados de forma a apresentar as principais características e decisões tomadas durante a migração apresentada pelos autores. Por fim, foi realizada uma comparação entre os trabalhos, através da apresentação de suas características e especificidades.

O próximo capítulo apresenta o processo utilizado para migração da solução de *offloading* CAOS para microsserviços, bem como as características da nova arquitetura concebida após essa migração.

4 CAOS MICROSERVICES

Este capítulo apresenta o processo utilizado na migração e concepção da arquitetura do CAOS baseada em microsserviços. O processo de migração é apresentado na seção 4.1. A arquitetura do CAOS MS é apresentada na seção 4.2.

4.1 Processo de Migração

Para realizar o processo de migração do CAOS foram seguidos 04(quatro) fases. Cada etapa gerou resultados, sendo alguns deles não tangíveis (como documentos ou relatórios), mas que contribuíram no conhecimento do autor desta dissertação, possibilitando uma melhor tomada de decisão nas fases seguintes. Na Figura 16 são apresentadas as fases seguidas, a saber: **(i)** Concepção e refinamento da proposta; **(ii)** Definição dos microsserviços; **(iii)** Definição das tecnologias; e **(iv)** Desenvolvimento, testes, melhorias e concepção da nova arquitetura. Estas fases foram divididas em 11(onze) etapas, onde estas são detalhadas a seguir.

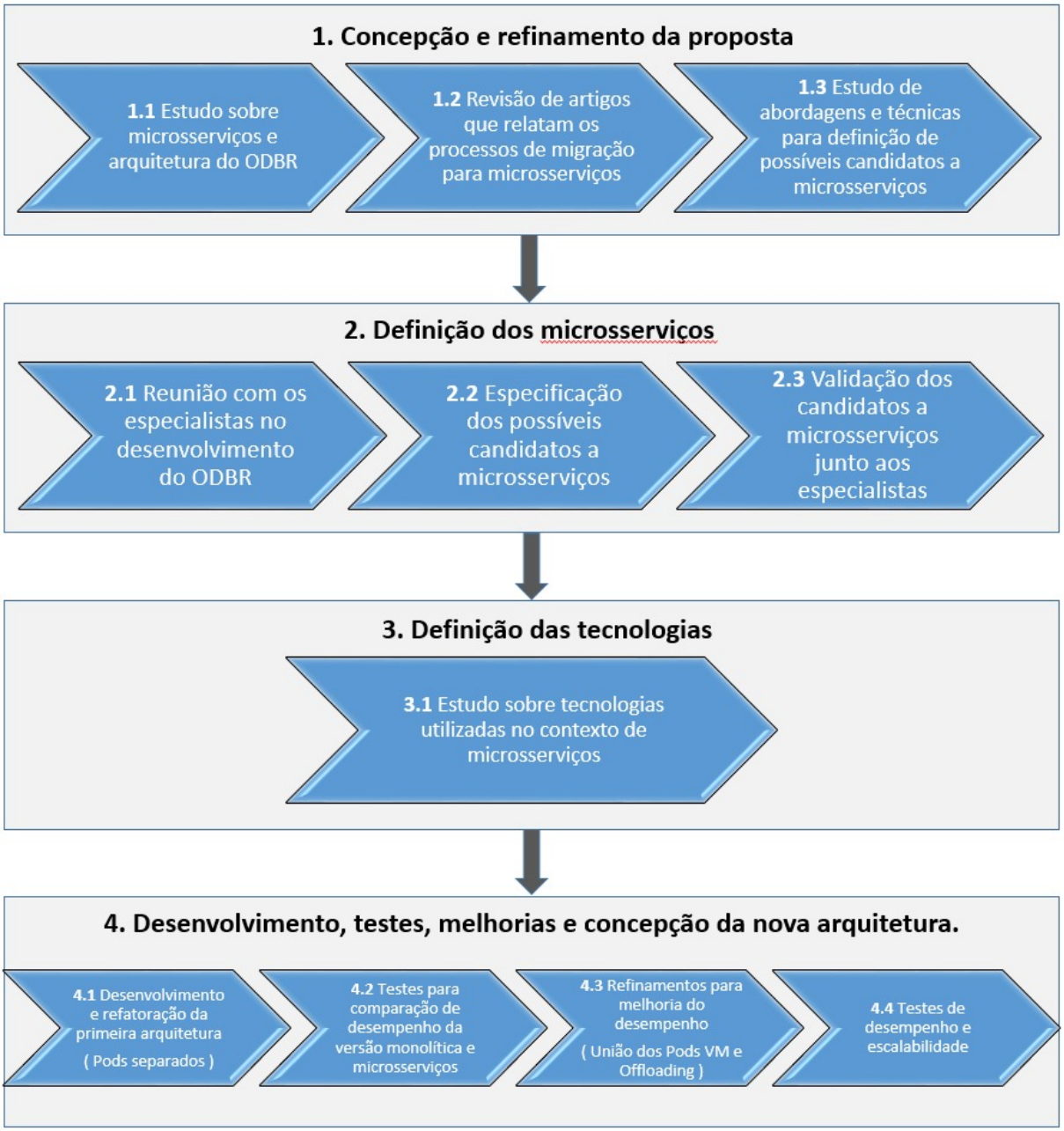
Nas subseções a seguir são detalhadas as fases e etapas seguidas durante o processo de migração, bem como os resultados gerados e sua contribuição no contexto geral de migração.

4.1.1 Fase 1 - Concepção e refinamento da proposta

A etapa inicial do processo foi destinada à análise da versão inicial do CAOS. Nesta etapa, a partir dos estudos publicados acerca da solução monolítica, foram analisadas as funcionalidades, documentação, tecnologias, módulos e dependências que compõem a solução monolítica do CAOS. Em conjunto, foram estudados conceitos e fundamentos da arquitetura microsserviços, buscando assimilar os benefícios que essa abordagem poderia proporcionar à nova versão da plataforma.

Com isso, foi possível obter uma visão inicial acerca dos benefícios que poderiam ser adquiridos com essa migração. Dentre eles, destacava-se uma escalabilidade mais eficiente, pois em situação de altas requisições, seria possível escalar somente os recursos que estavam sendo sobrecarregados. Além disso, foi percebido que poderia ser diminuía a complexidade para evolução da solução atual, visto que possibilitaria que novos microsserviços que porventura viessem a ser desenvolvidos, pudessem ser adicionados a nova arquitetura sem muito esforço, não impactando nos demais serviços. Com a evolução e manutenção individual dos microsserviços, a chance de problemas gerados durante a correção e incremento de funções aos serviços que

Figura 16 – Processo utilizado na migração do CAOS



Fonte: Elaborado pelo autor

compõem a solução tende a ser minimizada.

Na segunda etapa foram revisados diversos artigos que relatavam a experiência de migração para a arquitetura de microsserviços, possibilitando uma visão geral das técnicas e práticas que poderiam ser utilizadas no contexto de migração proposto ((SARKAR *et al.*, 2018); (FAN; MA, 2017); (BUCCHIARONE *et al.*, 2018); (BALALAIE *et al.*, 2016); (KNOCHE; HASSELBRING, 2018); (YUGOPUSPITO *et al.*, 2017); e (MUSTAFA *et al.*, 2018)). Durante a revisão, foi observado que os benefícios da arquitetura de microsserviços não são garantidos automaticamente, e só podem ser alcançados através de uma cuidadosa decomposição do software

existente (RICHARDSON, 2018). Logo, para iniciar a migração para microsserviços, deveriam ser identificados os possíveis candidatos a microsserviços e que comumente é utilizada uma abordagem empírica para definição dos serviços da aplicação. Em outras palavras, os serviços são definidos a partir do conhecimento dos *stakeholders* (desenvolvedores, projetistas e arquitetos) do software. No entanto, para realizar tais tarefas podem ser utilizadas técnicas, modelos e padrões de reengenharia já existentes na engenharia de software (DRAGONI *et al.*, 2017b).

A partir dos resultados encontrados anteriormente, a terceira etapa teve como objetivo a escolha de uma abordagem a ser usada na refatoração do CAOS. Foram identificadas algumas abordagens, como: Definição de Serviços a partir da análise da base de dados; Definição a partir da observação de logs; *Empirical Definition* (ED); ***S**ervice-oriented **P**rocess for **Re**engineering **a**nd **D**evops (SPReaD)*; e *Domain Driven Design* (DDD).

A técnica baseada na análise da base de dados, consiste em definir microsserviços a partir de possíveis dependências e relacionamentos presentes nas estruturas do banco de dados da aplicação. Dessa forma, o objetivo é dividir a aplicação do ponto de vista dos dados, procurando manter o mínimo de dependência possível(FAN; MA, 2017). A definição baseada na observação de logs, consiste em definir microsserviços a partir dos logs gerados pela aplicação em determinado período, possibilitando que através dos picos de acesso, seja realizado um mapeamento das funcionalidades da aplicação que são mais utilizadas. De acordo com essa técnica, as funcionalidades com essa características são possíveis candidatas a microsserviços(MUSTAFA *et al.*, 2018).

A técnica ED permite a definição dos microsserviços baseado no entendimento acerca da arquitetura do sistema, possibilitando identificar os serviços que poderiam funcionar de forma individual (TAIBI *et al.*, 2017b). A abordagem utilizando SPReaD possibilita a reengenharia de sistemas legados, integrando os aspectos de DevOps para o direcionamento e concepção de serviços (JUSTINO, 2018). O DDD entende o sistema como um grande domínio que pode ser dividido em subdomínios. Cada subdomínio corresponde a uma parte diferente do negócio. Os serviços devem ser desenvolvidos de acordo com os subdomínios da aplicação (RICHARDSON, 2018). Este estudo permitiu ter base para a escolha de uma estratégia, que é detalhada na próxima fase.

Baseado nos resultados gerados nas etapas anteriores, na quarta etapa foi realizada uma reunião com três dos especialistas que participaram do desenvolvimento da plataforma CAOS (dois deles atuaram diretamente na construção da solução, enquanto o outro atuou na

parte conceitual do projeto). Nesta reunião, foi realizada uma entrevista semiestruturada na modalidade presencial e durou cerca de uma hora (as perguntas realizadas estão dispostas no Apêndice A). Durante o encontro procurou-se identificar as dependências que haviam entre os componentes e quais as funcionalidades que a plataforma ofertava, possibilitando com isso a elaboração de uma breve concepção dos subdomínios da aplicação. Além disso, a entrevista viabilizou uma melhor compreensão sobre a forma como o CAOS foi projetado e desenvolvido.

4.1.2 Fase 2 - Definição dos microsserviços

Na quinta etapa foram discutidas as metodologias encontradas na etapa 1.3. Dentre as estudadas, a que mais se adequou ao contexto de migração proposto foi a *Domain Driven Design*. A escolha dessa técnica é justificada em razão da pré-divisão dos componentes da solução a ser migrada. Ainda em sua versão monolítica já era possível perceber claramente os subdomínios que representavam a ferramenta como um todo (e.g., autenticação, descoberta, offloading de dados e processamento, monitoramento e tomada de decisão).

Nesse sentido, o uso do DDD se mostrou bastante eficaz, pois diversos aspectos que essa metodologia possibilita eram almejados na nova solução. DDD tem como premissa a divisão do sistema em pequenos módulos ou subdomínios de um domínio principal, obedecendo os seguintes princípios: (*i*) Favorecimento do reuso; (*ii*) Alinhamento do código com o negócio; (*iii*) Mínimo de acoplamento e (*iv*)Independência da Tecnologia. O favorecimento do reuso indica que módulos construídos pudessem ser utilizados em contextos futuros do CAOS. O alinhamento do código com o negócio indica a necessidade de contato próximo entre desenvolvedores e os analistas do domínio, no caso, a proximidade com os especialistas no desenvolvimento do CAOS monolítico. O mínimo de acoplamento está relacionado com o modo a desenvolver os serviços da nova versão do CAOS em modelos bem definidos e organizados, sem que houvesse grande dependência entre os demais serviços, módulos e classes. Por fim, a Independência da Tecnologia indica que não deve-se ater a nenhuma solução específica e ter o foco no domínio principal. Com isso, foram selecionadas as tecnologias que pudessem atender o contexto da migração sem limitar o domínio (detalhes sobre essas tecnologias são apresentadas na seção 4.1.3). Após esse processo, foram definidos os possíveis candidatos a microsserviços.

Na sexta etapa também foi realizada uma nova reunião com os mesmos especialistas citados na etapa 2.1. Esse encontro visou apresentar e validar os candidatos a microsserviços obtidos com o uso do DDD no contexto da plataforma CAOS. Os candidatos a microsserviços

foram validados de acordo com as funcionalidades que eles exerciam e em consonância com a função de cada um no contexto do domínio principal. É preciso fazer aqui uma observação em relação ao processo de identificação dos microsserviços a serem migrados na plataforma CAOS. Como já relatado, a versão original da plataforma CAOS também permite o *offloading* de dados contextuais. Entretanto, como o esforço técnico no processo de migração era alto, este trabalho focou apenas na migração dos microsserviços que possuíam relação com o *offloading* de processamento, não abrangendo as funcionalidades de aquisição e gerenciamento contextual. Sendo assim, os microsserviços propostos para *offloading* de processamento são :

- **Microsserviço de Autenticação e Descoberta**, responsável por identificar os dispositivos que estão se conectando a plataforma, e servir de ponto de conexão com os demais serviços da solução;
- **Microsserviço de Monitoramento**, responsável por monitorar e registrar diversas métricas dos dispositivos móveis, além de receber e registrar as informações dos processos de *offloading* local e remoto. Essas métricas são utilizadas posteriormente para construção da árvore de decisão que é utilizada para se decidir sobre a viabilidade ou não do *offloading* de um método;
- **Microsserviço de Tomada de Decisão**, que é responsável pela construção das árvores de decisão de cada método de aplicações suportadas pelo CAOS, levando em consideração as métricas dos *smartphones*, bem como os tempos de execução local e remota. Este microsserviço envia as árvores de decisão para cada dispositivo conectado à plataforma sempre que a estrutura da árvore é modificada. O dispositivo móvel é responsável por utilizar a estrutura localmente para decidir sobre a execução ou não do *offloading*;
- **Microsserviço de *Offloading***, responsável por receber os dados do método de um dispositivo e enviá-lo ao serviço de Máquina Virtual Android. Ao término do procedimento, o mesmo devolve o resultado ao dispositivo o qual solicitou o processamento, mantendo o mesmo funcionamento da versão monolítica;
- **Microsserviço de Android Virtual Machine (VM)**, é o responsável por instanciar uma máquina virtual Android para realizar os processos de *offloading* requisitados pelo serviço de *Offloading*.
- **Microsserviço de Banco de Dados** encapsula um repositório responsável por registrar, fornecer e manter as informações geradas pelos demais microsserviços (e.g., dados do das métricas gerados pelo serviço de monitoramento, dados dos dispositivos gerados pelo

serviço de autenticação e descoberta e dados de realização de *offloading* local e remoto;

4.1.3 Fase 3 - Definição das tecnologias

Na sétima etapa foram estudadas e selecionadas ferramentas que pudessem auxiliar no desenvolvimento e implantação da nova arquitetura de microsserviços. Segundo (BALALAIE *et al.*, 2015), para viabilizar e obter vantagens com a arquitetura de microsserviços é necessária a implementação de diversos mecanismos que possibilitem o acesso, registro, descoberta, balanceamento de carga e escalonamento dos microsserviços.

Na refatoração da nova versão foi mantida a linguagem de programação Java, visto que o esforço necessário para alteração seria grande e sem motivos justificáveis, já que linguagem não era o a causadora dos problemas de forte acoplamento e baixo suporte à escalabilidade. Apesar disso, algumas funcionalidades foram construídas utilizando também a linguagem de programação PhP (e.g., repositório de aplicações). Por razões de desempenho foram utilizadas as ferramentas *Docker* para conteinerização dos microsserviços e *Kubernetes* para orquestração dos contêineres. A escolha destas ferramentas se deu pelo fato da grande comunidade e popularidade de ambas ferramentas. Mais detalhes sobre as ferramentas utilizadas na migração são apresentadas na seção 4.2.1.

4.1.4 Fase 4 - Desenvolvimento, testes, melhorias e concepção da nova arquitetura

Na oitava etapa foi concebida a primeira versão do CAOS em microsserviços, batizada de CAOS Microservices (CAOS MS). Nessa versão, a solução contava com todos os microsserviços separados e isolados em diferentes Pods[1] no Kubernetes. Os módulos do lado cliente (dispositivo móvel) permaneceram inalterados, exceto por alguns ajustes na comunicação por conta de particularidades do Docker. Na versão monolítica, o servidor onde executava os serviços do CAOS sempre assumia que quem realizava a requisição era o dispositivo, no entanto, no ambiente Docker, a requisição que chega aos contêineres é realizada a partir de mecanismos da própria ferramenta, fazendo com que a solução achasse que quem está requisitando é o Docker e não um dispositivo móvel. Dessa forma, apenas a maneira como a solução recebia a requisição do dispositivo foi alterada. Sendo assim, na nova versão, sempre que é realizada a requisição, também é informado o IP do dispositivo o qual requisitou o *offloading*. Tal fato garante uma

[1] Pod é a menor unidade dentro de um cluster Kubernetes. Esse mecanismo permite a execução dos contêineres construídos no Docker(CONCRETE, 2018)

compatibilidade entre aplicações desenvolvidas para ambas versões da plataforma, a monolítica e a baseada em microsserviços. Ressalta-se que somente os componentes do lado servidor passaram pelo processo de migração.

Na nona etapa foram realizados experimentos de desempenho e escalabilidade entre as versões monolítica e microsserviços. O experimento de desempenho visou verificar se a versão em microsserviços tinha sofrido penalização no tempo de realização do *offloading* quando comparada a versão monolítica. Já o experimento de escalabilidade visou verificar como a nova versão em microsserviços se comportaria durante as requisições e processos de escalabilidade horizontal, característica impossibilitada na versão monolítica. Após os testes de desempenho foi percebido que o tempo para realização de *offloading* havia aumentado muito quando comparados a versão monolítica, invalidando a nova solução, visto que desempenho é requisito base para o *offloading*. Mais detalhes sobre os resultados obtidos com estes experimentos são apresentados no capítulo 5.

Na décima etapa, foram realizados novos procedimentos de testes para identificar a causa da queda de desempenho encontrada na nona etapa. Nestes testes foi capturado o tempo de execução em diferentes momentos do processo de *offloading* de modo a se determinar os gargalos da execução remota dos métodos migrados. Então foi possível perceber que a separação dos microsserviços *Android VM* e *Offloading* em diferentes Pods no Kubernetes aumentava consideravelmente o tempo da execução do processo, pois após o microsserviço *Offloading* receber os argumentos da aplicação, era necessário acessar o serviço de descoberta do Kubernetes e transferir os dados recebidos para o Pod no qual serviço de *Android VM* era executado. Esse mesmo processo era repetido após o término da execução do processo do *offloading*, praticamente duplicando o tempo necessário para devolver o resultado ao dispositivo que havia requisitado o processamento. Diante desse fato, foi realizada a união dos microsserviços *Android VM* e *Offloading* em um único Pod. Essa alteração impossibilitou o escalonamento individual desses microsserviços. No entanto, ambos continuam com processos únicos e independentes (i.e., as alterações realizadas em qualquer um deles não impactam no funcionamento, manutenção e evolução do outro).

Na última etapa, após a união dos microsserviços *Android VM* e *Offloading*, os mesmos experimentos de desempenho realizados na etapa 4.2 foram repetidos. Com os resultados adquiridos foi possível perceber que o tempo para realização do *offloading* se manteve muito próximo à arquitetura monolítica, sendo que em alguns casos, o tempo do CAOS MS foi

levemente menor. No entanto, apesar do CAOS MS apresentar desempenho similar que o CAOS, essa não é a principal motivação da nova arquitetura. Nos experimentos de escalabilidade, o CAOS MS apresentou ganhos significativos em relação à arquitetura monolítica. Neste novo testo, uma maior quantidade de dispositivos conseguiu executar as tarefas de *offloading* com êxito e em menor tempo, comprovando a eficácia na escalabilidade da nova arquitetura. Mais detalhes sobre os resultados obtidos com estes experimentos são apresentados no capítulo 5.

4.2 Arquitetura CAOS Microservices

Nesta seção será apresentada a nova arquitetura do CAOS, bem como decisões e modificações realizadas para possibilitar a sua implementação em microsserviços. A nova versão do CAOS foi implementada com auxílio de *frameworks* e plataformas específicas para o desenvolvimento de microsserviços, o qual são descritas a seguir.

4.2.1 Tecnologias Utilizadas

Assim como o CAOS original, a plataforma CAOS MS foi implementada em quase toda sua totalidade na linguagem de programação Java. A exceção é o sistema Web para controle do repositório para aplicações que possuem métodos anotados para realização de *offloading*, sendo desenvolvido em PHP.

No início da implementação do CAOS MS, cogitou-se a possibilidade de utilização das ferramentas `NetflixOSS`[2] (*Netflix Open Source Software Center*). Entretanto, percebeu-se que essas ferramentas criam muitas dependências(JAMSHIDI *et al.*, 2018), com impacto diretamente no desempenho, requisito chave para uma solução de *offloading*. Diante disso, diversas ferramentas comumente utilizadas no desenvolvimento de microsserviços foram analisadas a fim de identificar as que mais se adequavam ao cenário da plataforma CAOS e que pudessem fornecer as funcionalidades que são necessárias no ambiente de microsserviços. Entre as soluções encontradas, foram escolhidas o `Docker`[3] para conteinerização dos microsserviços e `Kubernetes`[4] para gerenciamento dos contêineres.

A tecnologia Docker usa o kernel do Linux e recursos do kernel como Cgroups e namespaces para segregar processos, trabalhando com a ideia de conteinerização. Assim,

[2] URL: https://netflix.github.io, último acesso em 10 de Julho de 2019.
[3] URL: https://www.docker.com, último acesso em 10 de Julho de 2019.
[4] URL: https://kubernetes.io, último acesso em 10 de Julho de 2019.

possibilita que os serviços sejam executados e inicializados de maneira independente e com maior velocidade(HAT, 2019). A tecnologia Kubernetes foi utilizada como orquestrador dos contêineres construídos com o Docker. O Kubernetes é uma plataforma *open source* que automatiza as operações de gerenciamento dos contêineres Linux. Essa plataforma elimina grande parte dos processos manuais necessários para implantar e escalar as aplicações em contêineres. O Kubernetes oferece os recursos de orquestração e gerenciamento necessários para implantar contêineres em escala para essas cargas de trabalho, com facilidade e eficiência(HAT, 2019). Com isto, o projeto de refatoração do CAOS fez com que os microsserviços do lado servidor fossem transformados em imagens/contêineres do Docker.

4.2.2 Visão arquitetural CAOS MS

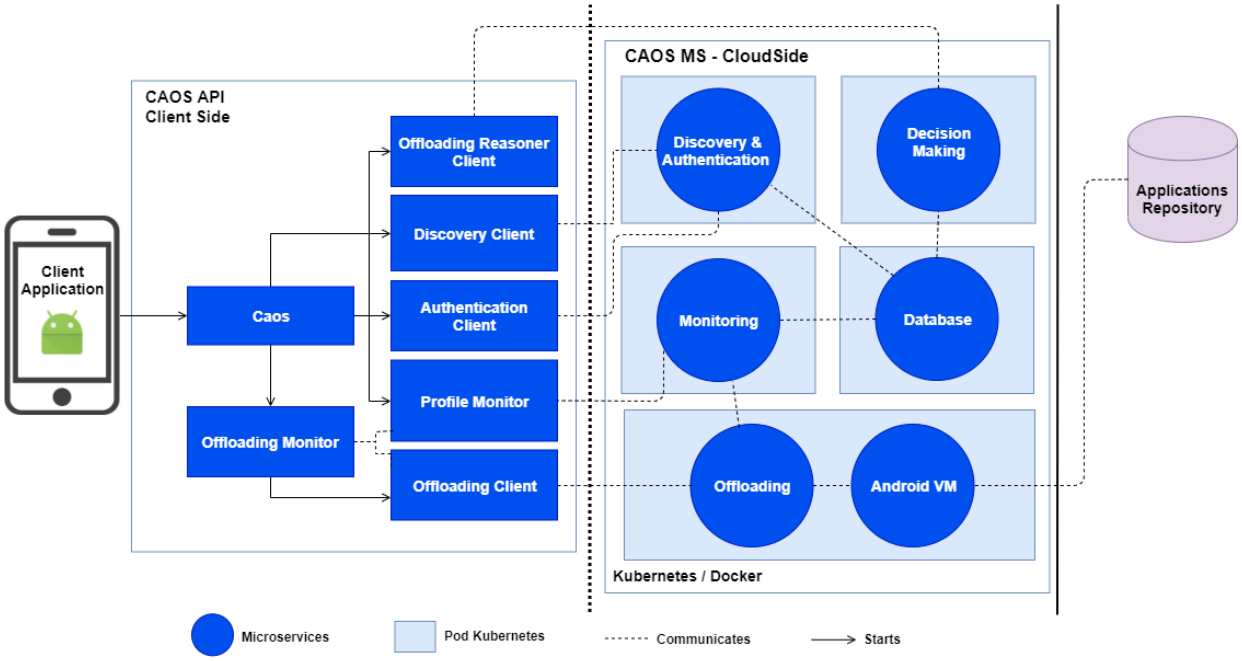
A arquitetura da plataforma CAOS MS é ilustrada na Figura 17. Pode ser observado que toda a comunicação entre o dispositivo móvel e os microsserviços ocorre de forma individual. Cada Pod do Kubernetes representa um processo separado e individualizado dos microsserviços, que podem ser escalados e encerrados isoladamente. Um Pod é a menor unidade dentro de um cluster Kubernetes. Esse mecanismo permite a execução dos contêineres construídos no Docker(CONCRETE, 2018). Vale ressaltar que em caso de falhas em algum microsserviço, os demais não serão afetados, e uma nova instancia do serviço onde ocorreu a falha será iniciada automaticamente pelo Kubernetes.

Do lado servidor, os serviços de autenticação e descoberta foram unificados em um único microsserviço, em razão da baixa quantidade de funções executadas e a semelhança de suas ações. A autenticação mantém uma lista de dispositivos que estão conectados à plataforma, enquanto a função do serviço de descoberta é fornecer aos dispositivos que se autenticaram no CAOS MS, informações sobre as portas dos demais microsserviços do lado servidor. Na Figura 18 é ilustrado como é realizado o processo de funcionamento e troca de informações entre os serviços e dispositivos.

O microsserviço de *Tomada de Decisão* assume a responsabilidade de criar as árvores de decisão e atualizar os dispositivos conectados. Essa ação ocorre sempre que houver mudanças entre as árvores geradas no microsserviço e a existente na aplicação em execução. Na versão monolítica, essas funcionalidades eram exercidas pelo módulo *Offloading Reasoner*. A Figura 19 ilustra o fluxo de execução desse processo de atualização de árvore de decisão.

O microsserviço *Banco de Dados* executa uma instância com o banco de dados

Figura 17 – Arquitetura da plataforma CAOS MS.



Fonte: Elaborado pelo autor

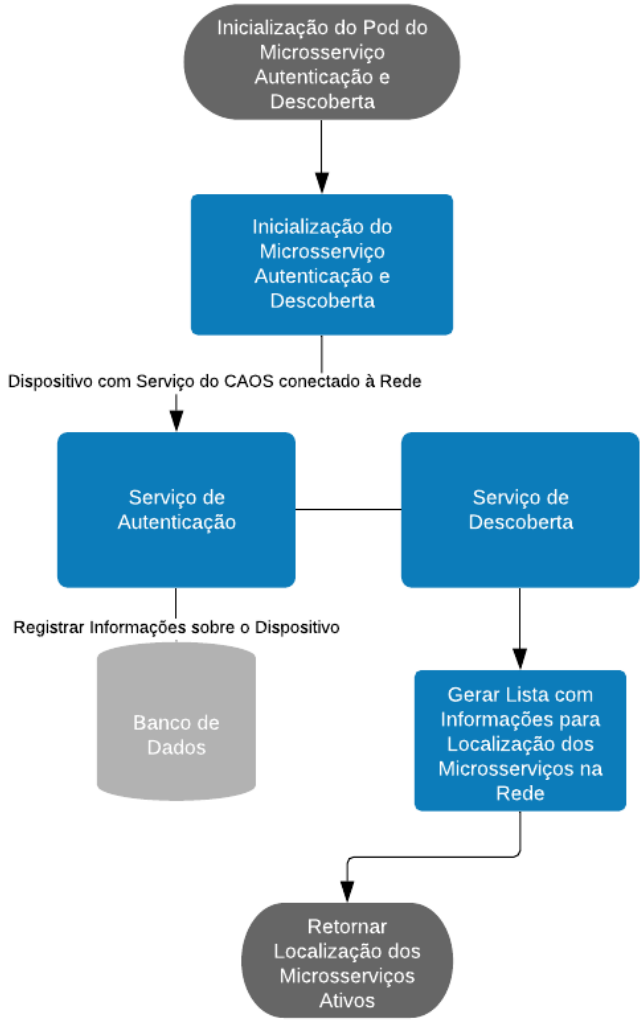
`Postgres`,[5] fornecendo mecanismos para persistência de dados aos microsserviços de *Autenticação e Descoberta*, *Monitoramento* e *Tomada de Decisão*. Os dados neste microsserviço incluem as métricas de monitoramento, informações sobre os métodos que foram executados na plataforma e informações sobre o histórico de execução de *offloading* dos dispositivos conectados ao CAOS MS.

A decisão de manter e isolar os dados da solução em um único microsserviço, ou seja, de manter os dados em uma base de dados fora dos microsserviços se deu em razão de algumas funcionalidades que necessitavam das informações que são geradas pelos diversos processos dentro dos microsserviços da solução. Dois exemplos clássicos disso são: só ocorre o processo de monitoramento se houver um dispositivo autenticado e o microsserviço de *Tomada de Decisão* só gera a árvore de decisão de forma precisa, se acessar todos os dados de *offloading* gerados pelo monitoramento. Foi cogitada a possibilidade de ser desenvolvido um mecanismo que realizasse essa comunicação. No entanto, a demora que poderia ser causada para execução de alguns processos que requerem os dados gerados por outros microsserviços impactaria diretamente no desempenho da solução.

Como já explanado, no CAOS monolítico, a execução do método migrado ocorre em uma *Virtual Machine* (VM) Android hospedada na plataforma. Para viabilizar esse processo, era necessário a instalação do sistema operacional (SO) Android na VM, além de algumas configura-

[5] URL: https://www.postgresql.org/, último acesso em 13 de Julho de 2019.

Figura 18 – Fluxo do Funcionamento do Microsserviço Autenticação e Descoberta.

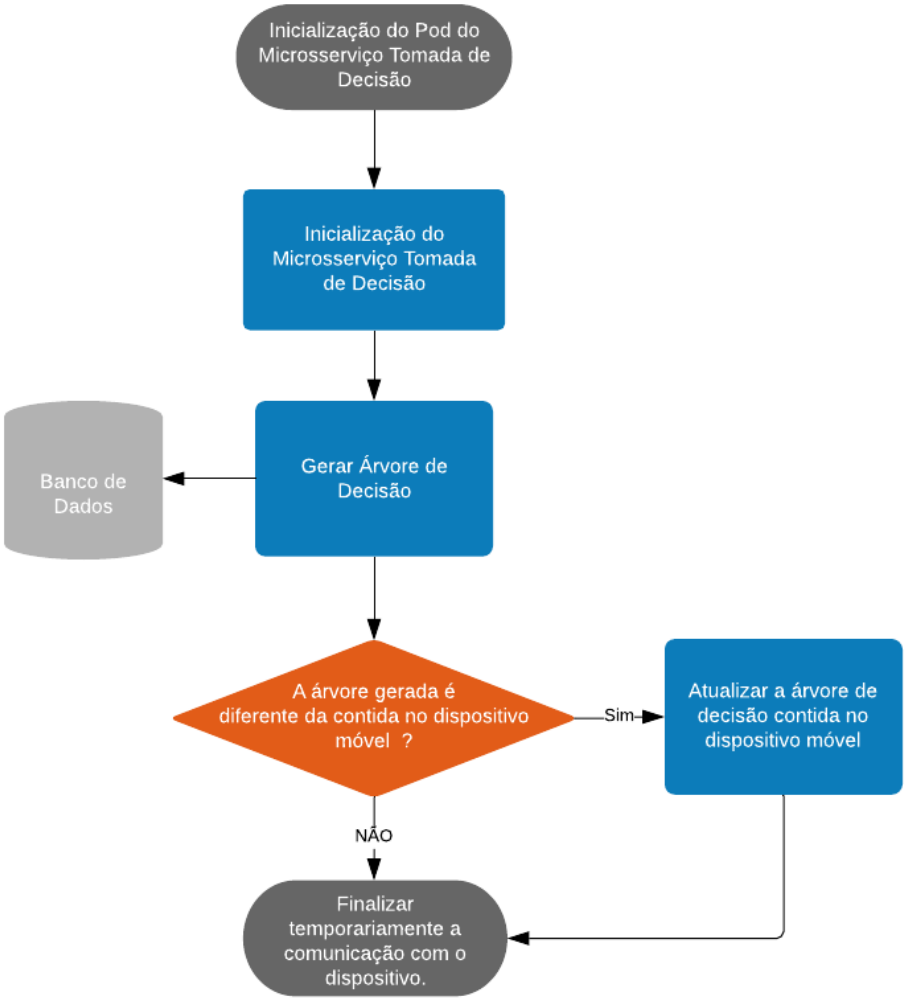


Fonte: Elaborado pelo autor

ções de rede para possibilitar o acesso do CAOS à VM. Na inicialização dos componentes da solução, a VM pré-configurada era acessada via comandos *Android Debug Bridge* (ADB) para instalação das aplicações que possuíam métodos marcados com a anotação @*Offloadable*. No entanto, a instalação dos aplicativos via ADB geralmente gerava muitos erros, principalmente quando a VM estava instanciada há a algum tempo. Nesse sentido, como a proposta era dividir ambos (Serviço de *Offloading* e VM) em dois microsserviços distintos, foi percebido que em um ambiente de microsserviços, o processo de instanciação e configuração da VM, bem como a instalação dos aplicativos não poderia ocorrer da mesma forma, já que poderia haver muitas instancias de ambos os microsserviços.

Toda aplicação compatível com o CAOS MS deve disponibilizar o pacote (APK) de sua aplicação para poder ser implantada nas máquinas virtuais Android da infraestrutura Kubernetes. Para melhorar o processo de implantação de novas aplicações, foram implementados

Figura 19 – Fluxo do Funcionamento do Microsserviço Tomada de Decisão.

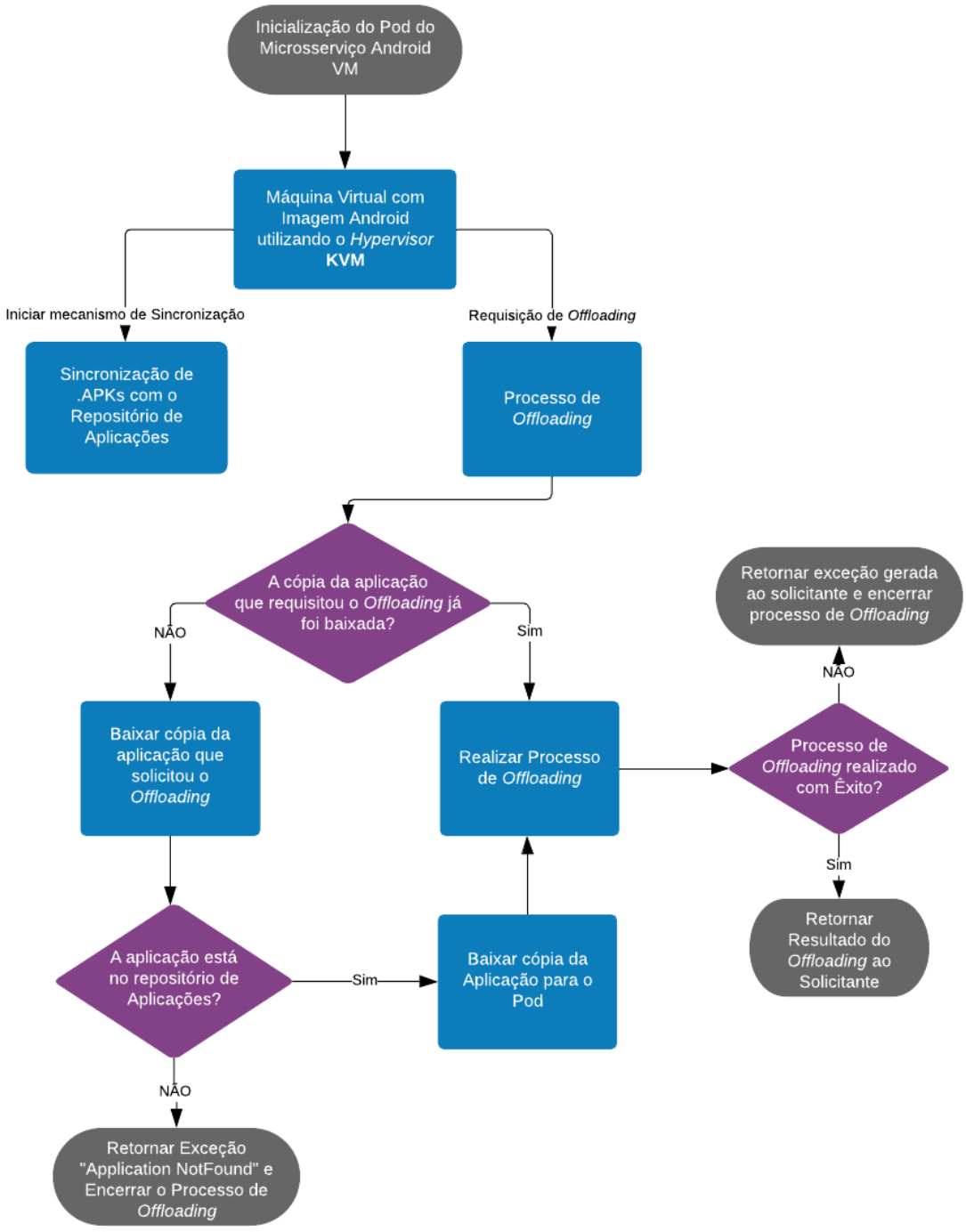


Fonte: Elaborado pelo autor

dois módulos para gerenciamento e sincronização das aplicações na VM. Um dos módulos é executado fora da rede Kubernetes, por exemplo em um provedor público de nuvem, e permite o registro e gerenciamento das aplicações enviadas pelos desenvolvedores. Na arquitetura ilustrada na Figura 17, esse módulo fica dentro do repositório de aplicações. Outro módulo fica interno ao microsserviço Android VM e possibilita que as aplicações que foram enviadas pelos desenvolvedores sejam sincronizadas com cada Pod que agrega um microsserviço de Offloading e uma VM Android. Além disso, o componente interno possui um mecanismo que verifica e atualiza as aplicações baixadas, caso tenham sido modificadas. Essas alterações trouxeram ganhos expressivos em relação ao processo de instalação de aplicações adotado na solução monolítica, pois além de pouco eficiente, esta demandava um esforço de configuração maior. A Figura 20 ilustra o funcionamento do microsserviço Android VM.

Por razões de desempenho, os microsserviços de *Offloading* e Android VM precisa-

Figura 20 – Fluxo de Funcionamento do Microsserviço Android VM.



Fonte: Elaborado pelo autor

ram ser implantados em um único Pod. Durante os experimentos, percebeu-se que, por conta do tamanho dos parâmetros passados em alguns métodos, o tempo para realizar o processo de *offloading* aumentava consideravelmente quando estavam em Pods separados. Mais informações sobre essa decisão são apresentadas no capítulo 5.

O módulo *Profile* Services foi migrado para o microsserviço de Monitoramento (*Monitoring*), sendo que não houve modificações na forma de funcionamento e comunicação entre o

serviço de Monitoramento e a API cliente. No entanto, como esse microsserviço também é responsável por monitorar os processos executados pelo microsserviço *Offloading*, foi desenvolvido um mecanismo de notificação para permitir que o microsserviço de *Offloading* envie as métricas de processamento dos pedidos dos clientes para o microsserviço de Monitoramento. Todavia, é importante frisar que não existe obrigatoriedade dessa comunicação para funcionamento do serviço de *offloading*.

O microsserviço *Android VM* inicia automaticamente uma imagem com o sistema operacional Android configurada com todas as dependências e configurações necessárias para realização dos processos de *offloading*. Após a inicialização do Android ou atualização das aplicações, ocorre uma sincronização automática das aplicações contidas no repositório de aplicações.

4.3 Considerações Finais

Neste capítulo foi apresentada a nova arquitetura baseada em microsserviços da plataforma CAOS e um relato sobre o processo de migração utilizado. O CAOS MS foi construído utilizando como base a versão monolítica e possibilitou que os mesmos recursos contidos na versão anterior, ficassem disponíveis nesta nova arquitetura.

A nova versão buscou possibilitar uma escalabilidade mais eficaz a plataforma CAOS, além de maior flexibilidade para evolução, manutenção e imersão de novos desenvolvedores. O processo de refatoração apresentado, mostrou todas as fases e etapas que foram seguidas desde a concepção da ideia até experimentos e melhorias.

5 EXPERIMENTOS

Para avaliar a arquitetura do CAOS MS, dois tipos de experimentos foram realizados. O primeiro buscou avaliar se a abordagem baseada em microsserviços trouxe algum prejuízo em relação ao desempenho das operações de *offloading* quando comparadas à arquitetura monolítica. Além disso, foram realizados experimentos para avaliar a escalabilidade do CAOS MS em comparação à versão original da solução. Para viabilizar tais experimentos, foi organizado um ambiente que possibilitasse a escalabilidade e execução distribuída dos microsserviços.

Nos experimentos realizados utilizou-se 02(duas) aplicações que já haviam sido utilizadas como objetos de teste por (GOMES *et al.*, 2017) para validação da versão original da plataforma. A primeira é denominada `BenchImage` e permite a aplicação de diferentes filtros em imagens. A segunda aplicação é denominada `MatrixOperations` e possibilita a multiplicação e soma de matrizes randômicas de diferentes dimensões.

A seção 5.1 apresenta o ambiente o qual ocorreu o teste de desempenho, os detalhes sobre a aplicação BenchImage, bem como os resultados adquiridos após o teste de desempenho em ambas arquiteturas. A seção 5.2 mostra como foi construído e configurado o ambiente do teste de desempenho, detalhes de funcionamento da aplicação MatrixOperations e os resultados obtidos com os testes de escalabilidade realizados com versão original e com a versão baseada em microsserviços.

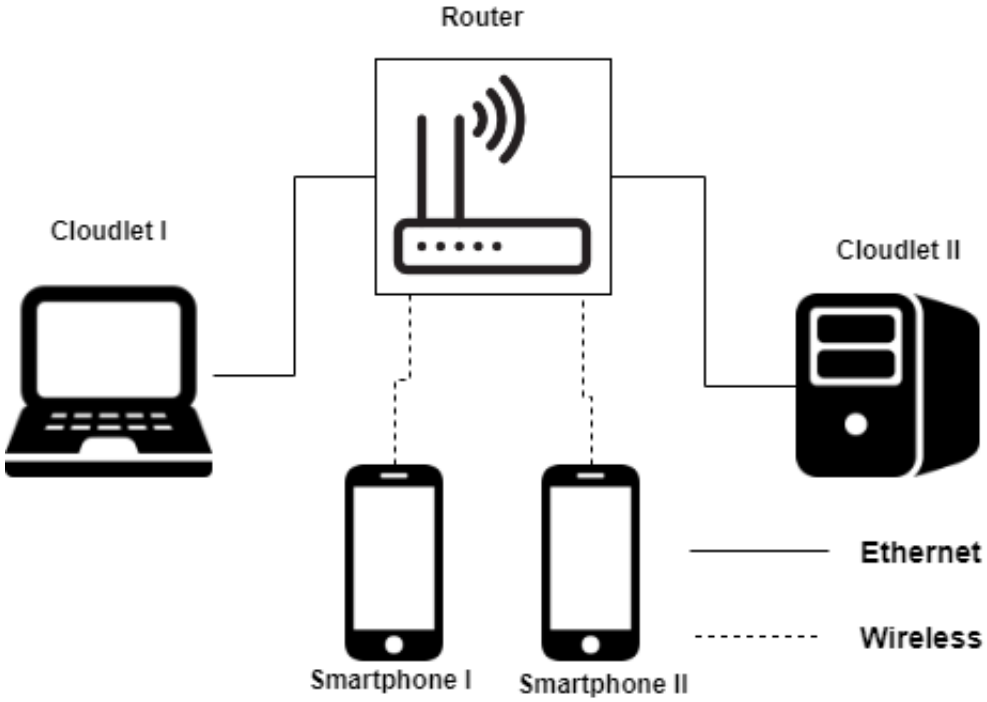
5.1 Teste de Desempenho

5.1.1 Descrição do Experimento

Para realizar os testes de desempenho de forma distribuída foi preparado um ambiente que contou com 02 (dois) computadores, 01 (um) Roteador *Wireless* TP-Link TL-WR940N com velocidade de banda de até 450MB e 02 (dois) *smartphones* com sistema operacional Android. Conforme é apresentado na Figura 21, por questões de desempenho, a conexão e a transmissão de dados entre os computadores foi realizada através da rede Ethernet, enquanto os *smartphones* ficaram conectados através da rede *wireless*.

Na Tabela 2 são apresentadas as especificações dos computadores e dispositivos móveis utilizados nos testes. A separação entre os *cloudlets* I e II indica que o nó *master* ficou responsável pelo gerenciamento e distribuição das requisições no *host*. Sendo assim, todos

Figura 21 – Ambiente utilizado para os testes de desempenho.



Fonte: Elaborado pelo autor

os microsserviços da solução foram executados no nó *host*. A distribuição foi realizada dessa maneira em razão dos recursos disponíveis nos computadores utilizados. Por possuir recursos computacionais superiores ao nó *master*, o *cloudlet* II foi escolhido para ser o responsável pela execução dos microsserviços. A motivação para utilização de dois cloudlets neste ambiente foi verificar como a plataforma iria se comportar diante de um ambiente distribuído, onde os serviços e requisições não estariam em um único nó da rede (como em sua versão monolítica).

Tabela 2 – Especificação dos dispositivos utilizados nos testes de desempenho.

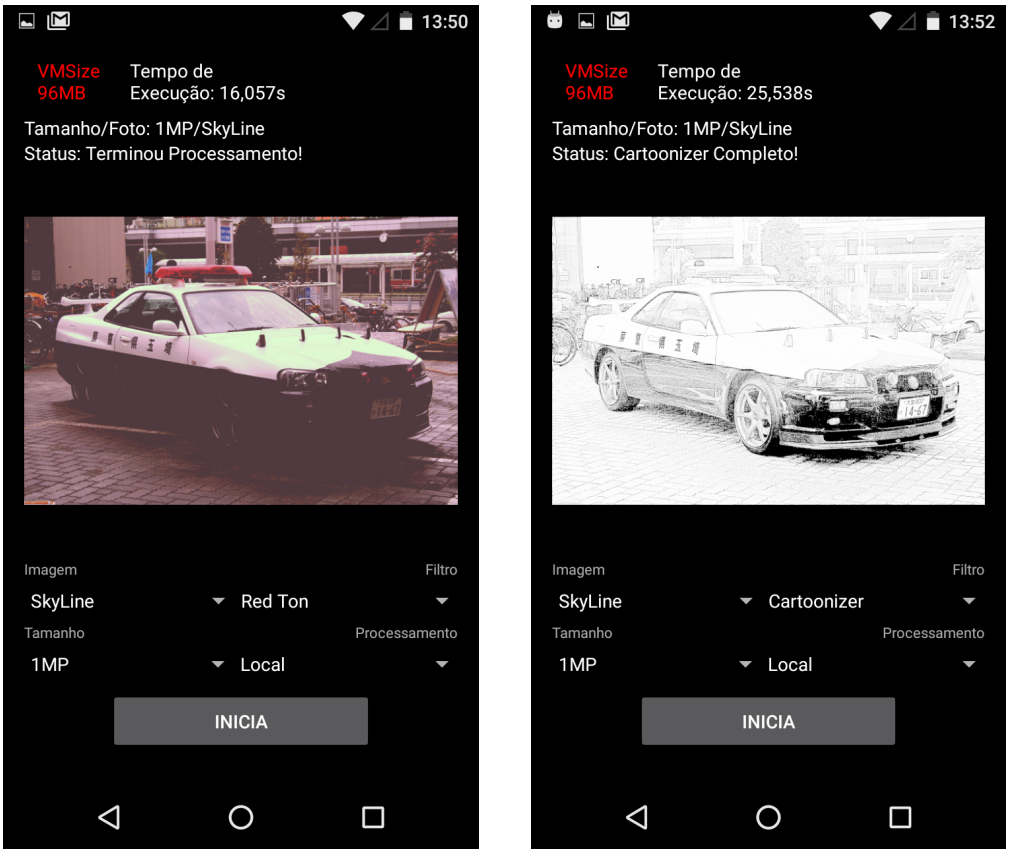
Equipamento	**Função**	**Configuração**
Smartphone I	Nó Cliente	Android Oreo 8.1, Qualcomm Snapdragon, 4 cores, 1.4GHz e 2GB RAM.
Smartphone II	Nó Cliente	Android Oreo 8.1, Qualcomm Snapdragon, 8 cores, 2.0GHz e 2GB RAM.
Cloudlet I	Nó Master	Linux Ubuntu 18.04 64-bits, Intel Core i5, 2 cores, 2.7GHz e 8GB RAM.
Cloudlet II	Nó Host	Linux Ubuntu 18.04 64-bits, Intel Core i7, 8 cores, 3.4GHz, NVIDIA GeForce GTX 750 e 24GB RAM.

Fonte: Elaborado pelo autor

5.1.2 Descrição da Aplicação BenchImage

Para a realização dos testes de desempenho, foi instalada a aplicação BenchImage em ambos *smartphones*. A aplicação BenchImage permite a aplicação de diversos filtros em imagens de diferentes resoluções, os quais demandam um razoável nível de processamento por parte dos *smartphones*. Dois tipos de filtros são fornecidos pela aplicação: `Cartoonizer` e `RedTone`. O primeiro aplica um filtro que transforma a imagem em uma versão `Cartoon`, enquanto o segundo aplica um tom avermelhado à imagem. O filtro `Cartoonizer` requer mais processamento que o filtro `RedTone`, de modo a exigir mais recursos do dispositivo móvel, e em geral, isso demandar uma ação de *offloading*. Além disso, as imagens disponibilizadas na aplicação variam de 0,3 megapixels à 8,0 megapixels, de modo que quanto maior a resolução, mais recursos computacionais são exigidos para realizar o processamento do filtro. A figura 22 apresenta a interface da aplicação BenchImage.

Figura 22 – *Screenshots* da Aplicação BenchImage



(a) (b)

Fonte: Elaborado pelo autor

A figura 22 (a) exemplifica a aplicação do filtro `RedTone`, enquanto a figura 22(b)

ilustra o filtro `Cartoonizer`. Na parte superior da aplicação é exibido o tempo em milissegundos que durou o processo de aplicação do filtro, possibilitando que o usuário altere o local onde a aplicação do filtro será executada. Os locais de execução podem variar entre local, nuvem e *cloudlet*.

5.1.3 Resultados Obtidos

Nos experimentos com a aplicação BenchImage, foi utilizado o filtro `RedTone` em imagens com diferentes resoluções (0,3MP, 2MP e 8MP). Os experimentos foram repetidos 30 (trinta) vezes com todos os parâmetros citados e com ambos *smartphones*. Já que segundo o Teorema Central do Limite (TCL) (FISCHER, 2010), quando se tem uma amostra suficientemente grande, a distribuição de probabilidade da média amostral pode ser aproximada por uma distribuição normal. Essa aproximação é valida a partir de 30 médias amostrais.

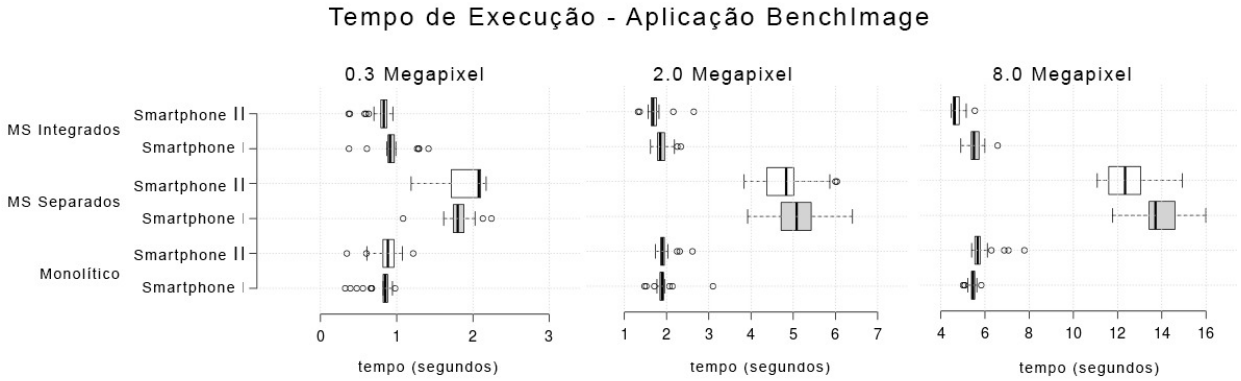
Para permitir uma comparação mais justa, e levando em consideração que o maior estresse de recursos computacionais ocorre na VM, os mesmos recursos destinados a virtualização Android na versão monolítica foram mantidos para versão microsserviços. As configurações aplicadas à virtualização foram *Hypervisor* KVM, Android x86 4.4.2 com 4 GB RAM e 2 Cores. Para garantir a realização do *offloading*, a aplicação BenchImage teve seus métodos anotados como estáticos, indicando que sempre o *offloading* será realizado, sem necessidade de uma tomada de decisão.

Os testes foram realizados com duas configurações do CAOS MS, cujos resultados são apresentados na Figura 23 por meio de gráficos *Box Plot*. A Figura 23 apresenta o desempenho da aplicação em termos do tempo total de execução (comunicação + processamento remoto) nos dois *smartphones* utilizados nos experimentos.

Além das versões com microsserviços, os mesmos testes de desempenho também foram realizados na arquitetura monolítica original para efeito de comparação. Os valores obtidos para a versão original do CAOS são referenciados na Figura 23 como cenário “Monolítico”.

A primeira avaliação do CAOS MS foi feita utilizando uma distribuição completa de todos os microsserviços em Pods distintos (referenciado na Figura 23 como “Microsserviços Separados”). Nesta configuração percebeu-se que o tempo de execução da solução teve um aumento considerável em relação ao desempenho da versão monolítica. Com estes resultados, mesmo com os benefícios relacionados à flexibilidade e à escalabilidade propiciados pela arquitetura de microsserviços, a solução tornava-se inviável, já que uma das premissas da técnica

Figura 23 – Comparação entre os tempos de execução da aplicação BenchImage nas versões monolítica e microsserviços da plataforma CAOS.



		0.3 Megapixel	2.0 Megapixel	8.0 Megapixel
Monolítico	Smartphone II	0,88 ± 0,16	1,95 ± 0,17	5,82 ± 0,53
	Smartphone I	0,80 ± 0,16	1,91 ± 0,26	5,44 ± 0,18
MS Separados	Smartphone II	1,93 ± 0,25	4,74 ± 0,55	12,47 ± 0,97
	Smartphone I	1,81 ± 0,19	5,07 ± 0,58	13,86 ± 1,04
MS Integrados	Smartphone II	0,79 ± 0,14	1,72 ± 0,22	4,72 ± 0,22
	Smartphone I	0,95 ± 0,19	1,91 ± 0,17	5,57 ± 0,29

Fonte: Elaborado pelo autor

de *offloading* é a melhoria do desempenho no processamento de funções.

Como já relatado na seção 4.1, diante dos primeiros resultados foi capturado o tempo de execução em diferentes momentos do processo de *offloading*, objetivando identificar os gargalos da execução remota dos métodos migrados. Foi então possível perceber que a separação dos microsserviços *Android VM* e *Offloading* em diferentes Pods no Kubernetes aumentava consideravelmente o tempo da execução do processo, pois após o microsserviço *Offloading* receber os dados dos parâmetros da aplicação, era necessário acessar o serviço de descoberta do Kubernetes e transferir os dados recebidos para o Pod o qual possuía o serviço de *Android VM*. Esse mesmo processo era repetido após o término da execução do processo do *offloading*, praticamente duplicando o tempo necessário para devolver o resultado ao dispositivo que havia requisitado o processamento.

Com estas informações, uma nova configuração do CAOS MS foi realizada, implantando-se os microsserviços *Android VM* e *Offloading* em único Pod. Esta configuração é referenciada na Figura 23 como “Microsserviços (MS) Integrados”. Os mesmos testes de desempenho foram refeitos. De acordo com os dados obtidos, é possível notar que o tempo de execução se manteve muito próximo em ambas arquiteturas, monolítica e microsserviços, e em alguns momentos a solução baseada em microsserviços obteve desempenho melhor que a versão monolítica.

É importante mencionar que a implantação dos microsserviços em um único Pod impossibilita o escalonamento individual desses microsserviços. No entanto, ambos continuam com processos únicos e independentes, ou seja, as alterações realizadas em qualquer um deles

não impactam no funcionamento, manutenção e evolução do outro.

5.2 Teste de Escalabilidade

5.2.1 Descrição do Experimento

Uma das grandes motivações do CAOS MS era prover uma maior escalabilidade, dado que a versão monolítica só permitia uma abordagem vertical, ou seja, aumentando recursos do cloudlet onde a plataforma servidora seja implantada.

Nesse sentido, foram realizados testes de escalabilidade para verificar a eficácia do CAOS MS nesse aspecto. O ambiente no qual ocorreram os testes é semelhante ao apresentado na Figura 21. No entanto, neste novo cenário foram utilizados 20 (vinte) *smartphones* reais de voluntários, objetivando verificar como a solução se comportaria diante de um ambiente com usuários reais. Esse fato ocorre em razão de toda a comunicação entre estes microsserviços e os dispositivos ocorrer de maneira única e automática, ou seja, por dispositivo. Antes de iniciar os testes, foram instaladas em todos os *smartphones* as aplicações BenchImage e MatrixOperations. Mais detalhes sobre o funcionamento da aplicação MatrixOperations são apresentados na seção 5.2.2.

Em razão do tempo necessário para executar os testes com diferentes valores de parâmetros, a utilização de dispositivos reais e o recurso computacional exigido para processos mais complexos, limitou-se em ambas aplicações o uso de um único parâmetro. Nos experimentos com a aplicação BenchImage foi utilizado o filtro `RedTone` com a resolução (2.0 MP). Já na aplicação MatrixOperations foram utilizada matrizes com dimensão 100x100. Além disso, o experimento com cada parâmetro de ambas aplicações foi repetido 03 (três) vezes com todos os *smartphones*. Esse processo ocorreu de forma simultânea, ou seja, todos os dispositivos realizavam a requisição de *offloading* ao mesmo tempo.

Na Tabela 3 são apresentados os recursos de *Central Processing Unit* (CPU) e memória configurados para cada microsserviço nos testes de escalabilidade. No caso dos recursos configurados para a versão monolítica, foram mantidas as configurações descritas na subseção 5.1.

Inicialmente foi especificado que cada microsserviço só iniciaria com uma única instância. No entanto, por limitações de recursos computacionais disponíveis no nó host, ficou limitada a escala de até no máximo 03 (três) instâncias por microsserviço. Para proporcionar a

Tabela 3 – Especificação dos recursos de CPU e memória configurados para cada microsserviço para o experimento de Escalabilidade.

Microsserviços	Configuração
Discovery & Authentication	0,5 Core e 1GB de RAM
Monitoring	0,5 Core e 1GB de RAM
Decision Making	0,5 Core e 1GB de RAM
Offloading & Android VM	2 Cores e 4GB de RAM
Database	0,5 Core e 1GB de RAM

Fonte: Elaborado pelo autor

escalabilidade de forma automática foi utilizado o serviço *Horizontal Pod Autoscaler* (HPA)[1] do Kubernetes. Optou-se por configurar esse serviço para iniciar uma nova instância sempre que algum Pod estivesse com o uso de CPU e/ou memória superior a 80%, de acordo com os recursos especificados na Tabela 3.

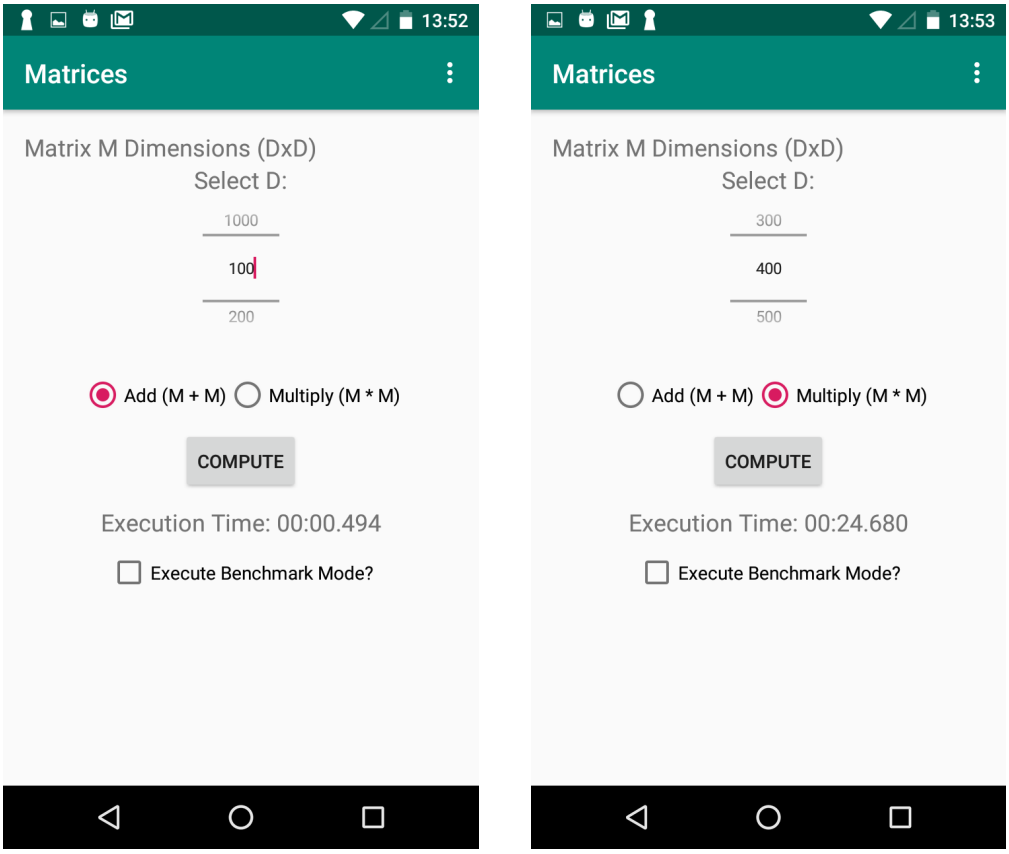
5.2.2 Descrição da Aplicação MatrixOperations

A aplicação MatrixOperations permite a realização das operações de adição e multiplicação de matrizes. Para isso, a aplicação possibilita a geração de uma randômica com dimensões 100x100 até 1000x1000 e realiza a operação que foi determinada pelo usuário (multiplicação ou soma). A operação de soma normalmente não requer muito recurso computacional. Em testes realizados por (GOMES *et al.*, 2017), as operações de soma sempre eram mais rápidas quando executadas no próprio dispositivo móvel, enquanto a multiplicação quase sempre é melhor ser executada remotamente (exceto para matrizes pequenas). Semelhante a aplicação BenchImage, a MatrixOperations possibilita a execução de operações que requer um relativo poder computacional dos dispositivos, possibilitando realizar comparações entre o tempo de execução local, ou seja, no próprio dispositivo, e a execução remota. Essa aplicação também já foi utilizada por (GOMES *et al.*, 2017) em testes na validação da versão original do CAOS.

Na figura 24 é apresentada a interface da aplicação MatrixOperations. Na figura 24(a) é ilustrada a operação de soma de matriz com dimensão 100x100 e a figura 24(b) ilustra a operação de multiplicação com uma matriz com dimensão 400x400. Na parte inferior da aplicação, fica o tempo da execução do procedimento em milissegundos. Diferente da BenchImage, não é possível parametrizar quando a execução será local ou remota, sempre que ela conseguir estabelecer conexão com o servidor que executa os serviços do CAOS, ocorrerá o processo de *offloading* remoto (exceto quando a estrutura de decisão dinâmica decide que a execução local é

[1] URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. último acesso em 13 de Julho de 2019.

Figura 24 – *Screenshots* da aplicação MatrixOperations



(a) (b)

Fonte: Elaborado pelo autor

melhor).

5.2.3 Resultados Obtidos

Durante a execução dos testes foram monitorados os recursos que estavam sendo utilizados pelos microsserviços e o tempo de processamento dos *offloadings*. Nessa ocasião foi percebido que os recursos disponíveis na instância onde estavam os microsserviços *Android VM Service* e *Offloading Service* ilustrados na tabela 3 não eram suficientes para realizar o processamento de todos os dispositivos, causando *TimeoutException* em alguns *smartphones*. Sendo assim, além do tempo de execução das requisições, foi também aferido a porcentagem de dispositivos que conseguiam efetuar o *offloading* com êxito.

A Tabela 4 apresenta a taxa de sucesso e o tempo de execução na realização do *offloading* obtidos com a aplicação BenchImage. Nesse cenário, somente o Pod no qual estavam os microsserviços *Android VM Service* e *Offloading Service* sofreram sobrecarga a ponto de serem iniciadas novas instâncias. É possível perceber que conforme o número de instâncias

crescia, a taxa de sucesso na execução do *offloading* aumentava e o tempo médio de execução diminuía. O mesmo resultado não foi observado com a versão monolítica, pois como já relatado, o projeto de sua arquitetura não permite a escalabilidade individual dos componentes e nem a criação de novas instâncias. Estes resultados mostram eficiência na escalabilidade do CAOS MS, pois somente os recursos que estavam sendo sobrecarregados foram escalados, enquanto os demais continuaram com uma única instância.

Tabela 4 – Resultados dos testes com a aplicação BenchImage.

Arquitetura	Quantidade de Instâncias	Taxa de Sucesso Médio na Execução	Tempo Médio da Execução
Monolítica	01	63,60%	22,95
Microsserviços	01	58,82%	13,47
Microsserviços	02	98,02%	13,15
Microsserviços	03	100,00%	8,23

Fonte: Elaborado pelo autor

Os resultados obtidos nos testes com a aplicação MatrixOperations são apresentados na Tabela 5. Em comparação aos testes com a aplicação BenchImage, percebe-se que a taxa de sucesso foi superior, enquanto o tempo de execução inferior. Esse fato ocorreu em razão dos parâmetros utilizados, pois a multiplicação de Matrizes com dimensão 100x100 consome menos recursos que o filtro `RedTone` de 2MP da aplicação BenchImage.

É possível notar ainda que, assim como nos resultados obtidos com o uso da aplicação BenchImage, conforme o número de instâncias subia, a taxa de sucesso aumentava e o tempo de execução diminuía consideravelmente. Tais resultados comprovam uma melhoria no funcionamento da plataforma com a nova versão em microsserviços, já que foi possível reduzir o tempo necessário para realizar *offloading* mesmo com o aumento de usuários. Estes resultados, quando comparado com a versão monolítica, mostram um ganho significativo no quesito tempo de execução e taxa de sucesso.

Tabela 5 – Resultados dos testes com a aplicação MatrixOperations.

Arquitetura	Quantidade de Instâncias	Taxa de Sucesso Médio na Execução	Tempo Médio da Execução
Monolítica	01	83,30%	8,65
Microsserviços	01	86,27%	7,50
Microsserviços	02	100,00%	5,41
Microsserviços	03	100,00%	3,57

Fonte: Elaborado pelo autor

Adicionalmente, são mostrados na Tabela 6 os dados acerca do monitoramento de recursos que estavam sendo utilizados em cada microsserviços durante todo o experimento. A

tabela apresenta o nome do microsserviço, a quantidade de Pods máximo que o microsserviço atingiu durante o teste, e a média de uso de memória RAM e CPU.

Tabela 6 – Resultados do uso de recursos por Microsserviço.

Microsserviço	Quantidade de Pods	Média de Uso da Memória	Média de Uso da CPU
Discovery & Authentication	01	347MB	26,40%
Monitoring	01	541MB	57,95%
Decision Making	01	714MB	39,74%
Offloading & Android VM	03	2,14GB	78,60%
Database	01	371MB	9,34%

Fonte: Elaborado pelo autor

De acordo com os dados apresentados na Tabela 6, nota-se que o uso de memória RAM foi baixo em todos os microsserviços e que a maior sobrecarga de fato ocorreu nos microsserviços *Offloading* e *Android VM*, pois mesmo com o uso de 03 instâncias, estes ainda apresentaram um uso de CPU bastante elevado. Esse resultado confirma a hipótese levantada na seção 5.1, quando afirmou-se que o maior consumo de recursos ocorreria nos pods que executavam os serviços *Offloading* e *Android VM*.

5.3 Considerações Finais

Este capítulo apresentou uma comparação entre a versão monolítica da plataforma CAOS e a nova versão que foi desenvolvida utilizando uma arquitetura baseada em microsserviços. Para isto, foram realizados 02(dois) experimentos, sendo: teste de desempenho e teste de escalabilidade.

No teste que objetivou verificar o aspecto de desempenho, percebeu-se que não houveram perdas em relação a versão monolítica, bem como foi verificado aspectos de escalabilidade da nova versão. Algumas ameaças podem comprometer a validade do teste de escalabilidade, já que foram utilizados apenas 20(vinte) *smartphones*, bem como não foi investigado como a arquitetura se comportaria diante de cenários com poucas e repentinas requisições.

O orquestrador de contêineres Kubernetes foi essencial no teste de escalabilidade, visto que ele proporcionou a escala correta dos microsserviços de acordo com as métricas de hardware previamente configuradas.

6 CONCLUSÃO E TRABALHOS FUTUROS

Este capítulo sintetiza os principais resultados alcançados e discussões realizadas ao decorrer deste estudo. Na seção 6.1 são apresentados os resultados alcançados com a nova solução de *offloading* baseada em microsserviços. Na seção 6.2 são apresentadas as limitações inerentes à solução concebida. Na seção 6.3 são exibidas as publicações realizadas a partir do estudo proposto nesta dissertação. Por fim, na seção 6.4 apresentam os trabalhos futuros, visando possíveis extensões da solução.

6.1 Resultados Alcançados

Esta dissertação de mestrado apresentou o desenvolvimento e concepção do CAOS MS, uma plataforma com suporte a *offloading* para dispositivos móveis baseada na arquitetura de microsserviços. O CAOS MS evolui uma solução monolítica previamente construída, porém com restrições de escalabilidade e com forte acoplamento. Neste estudo foi detalhado o processo de refatoração e concepção do CAOS MS a partir de sua versão monolítica, além de experimentos comparativos de desempenho e escalabilidade entre a arquitetura original e a nova arquitetura proposta.

De acordo com os resultados obtidos e apresentados nessa pesquisa, a solução CAOS MS mostrou-se eficaz nos quesitos escalabilidade e desempenho. A princípio era cogitada a hipótese de perda de desempenho por conta de componentes necessários na comunicação e ambiente de microsserviços. No entanto, após melhorias realizadas na primeira versão de microsserviços concebida, o teste de desempenho mostrou que não houveram perdas significativas em relação a versão monolítica. Já nos testes de escalabilidade houve o escalonamento correto dos microsserviços de modo a distribuir a carga entre vários nós, fazendo também com que o tempo na realização dos processos de *offloading* diminuísse. Também verificou-se que por diminuir o tempo na execução do *offloading*, mais dispositivos móveis conseguiam finalizar o processamento com êxito.

A partir dos resultados coletados durante os experimentos com o CAOS MS, percebeu-se que a escalabilidade viabilizada pela arquitetura de microsserviços tornou a execução do *offloading* de processamento consideravelmente mais rápida, enquanto possibilitava melhor utilização dos recursos computacionais, visto que somente os microsserviços que estavam recebendo muitas requisições eram escalados. Desta forma, acredita-se que esse tipo de abordagem abra

um leque de possibilidades aos pesquisadores no contexto de MCC e *offloading*.

As lições aprendidas na migração da versão monolítica do CAOS para a versão baseada em uma arquitetura microsserviços pode resultar em um catálogo de padrões e decisões de projeto na identificação, delimitação de escopo e contextos, refatorações e decisões de projeto que podem ser úteis em outros projetos de modernização de software com características similares para uma arquitetura microsserviços.

6.2 Limitação

A solução CAOS MS limitou-se somente aos componentes referentes ao *offloading* de processamento presentes na plataforma CAOS. Desta forma, o *offloading* de dados, bem como a aquisição contextual presentes na versão original, não foram migradas para a nova versão baseada em microsserviços. A princípio, foi cogitada toda a migração de toda a plataforma, todavia, por razões de complexidade e possíveis problemas que viessem a ocorrer durante a migração de ambas funcionalidades, decidiu-se que somente os componentes relacionados ao *offloading* de processamento seriam refatorados nessa primeira migração, ficando o *offloading* de dados como trabalhos futuros.

6.3 Produção Bibliográfica

Durante o decorrer do curso de mestrado, foram realizadas 02(duas) publicações em eventos nacionais, ambas relacionadas ao estudo apresentado nesta dissertação. Mais detalhes sobre essas publicações são apresentadas logo a seguir:

- ***A Microservice Based Architecture to Support Offloading in Mobile Cloud Computing*:** Publicado como artigo completo no XIII *Brazilian Symposium on Software Components, Architectures, and Reuse* (SBCARS 2019), com Qualis B3, o estudo apresentou a arquitetura concebida após a refatoração do CAOS para microsserviços, bem como resultados obtidos em experimentos com essa nova arquitetura. Autores: Adriano L. Cândido (UFC), Fernando A. M. Trinta (UFC), Lincoln S. Rocha (UFC), Paulo A. L. Rego (UFC), Nabor C. Mendonça (UNIFOR), and Vinicius C. Garcia (UFPE);
- **Um relato sobre a migração de uma plataforma de *offloading* para microsserviços:** Publicado como artigo completo no VII *Workshop on Software Visualization, Evolution and Maintenance* (VEM 2019), com Qualis B5, o estudo apresentou o processo e nova

arquitetura concebida após a refatoração do CAOS. Autores: Adriano L. Cândido (UFC), Fernando A. M. Trinta (UFC), Lincoln S. Rocha (UFC), Paulo A. L. Rego (UFC), Nabor C. Mendonça (UNIFOR), and Vinicius C. Garcia (UFPE);

6.4 Trabalhos Futuros

Como trabalhos futuros, propõem-se extensões e evoluções da solução CAOS MS, sendo elas:

- Desenvolver um mecanismo que possibilite realizar o processamento das funções de forma distribuída, de modo que uma determinada requisição seja dividida e processada em todos os microsserviços de *offloading* disponíveis, levando em consideração métricas como seu poder de processamento e o tempo de resposta do microsserviço, podendo aumentar o desempenho no processamento das requisições. Acredita-se que apesar das limitações que essa abordagem poderia agregar, pode ser diminuído consideravelmente o tempo na realização do processo de *offloading* de processamento, visto a requisição seria dividida em partes e seria processada de forma distribuída, seguindo a ideia "dividir para conquistar".
- Conduzir novos experimentos relacionados aos aspectos relacionados a qualidade da arquitetura da versão em microsserviços, como grau de coesão e acoplamento entre os serviços.
- Realizar a refatoração dos módulos de *offloading* de dados e aquisição contextual presentes na versão monolítica, de forma a possibilitar seu funcionamento também na arquitetura de microsserviços.

REFERÊNCIAS

AEPONA. Mobile cloud computing solution brief. White Paper, 2010.

AKHERFI, K.; GERNDT, M.; HARROUD, H. Mobile cloud computing for computation offloading: Issues and challenges. **Applied computing and informatics**, Elsevier, v. 14, n. 1, p. 1–16, 2018.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Migrating to cloud-native architectures using microservices: an experience report. In: SPRINGER. **European Conference on Service-Oriented and Cloud Computing**. *[S. I.]*, 2015. p. 201–215.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: migration to a cloud-native architecture. **IEEE Software**, IEEE, v. 33, n. 3, p. 42–52, 2016.

BALALAIE ABBAS HEYDARNOORI, P. J. D. A. T. e. T. L. A. **Microservices migration patterns**. 2018. Disponível em: https://onlinelibrary.wiley.com/doi/10.1002/spe.2608. Acesso em: 11 mai. 2019.

BALDWIN, C. Y.; CLARK, K. B. **Design rules: The power of modularity**. *[S. I.]*: MIT press, 2000. v. 1.

BRIAND, L. C.; MORASCA, S.; BASILI, V. R. Property-based software engineering measurement. **IEEE transactions on software Engineering**, IEEE, v. 22, n. 1, p. 68–86, 1996.

BUCCHIARONE, A.; DRAGONI, N.; DUSTDAR, S.; LARSEN, S. T.; MAZZARA, M. From monolithic to microservices: An experience report from the banking domain. **Ieee Software**, IEEE, v. 35, n. 3, p. 50–55, 2018.

BUSCHMANN, F. **PatternOriented-Software-Architecture-A-System-of-Patterns-Volume-1**. *[S. I.]*: Ashish Raut, 1996. v. 1.

CHUN, B.-G.; IHM, S.; MANIATIS, P.; NAIK, M.; PATTI, A. Clonecloud: elastic execution between mobile device and cloud. In: ACM. **Proceedings of the sixth conference on Computer systems**. [*S. l.*], 2011. p. 301–314.

CISCO. **Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2017–2022 White Paper - Cisco**. 2019. Disponível em: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html. Acesso em: 10 out. 2019.

CLEMENTS, P. C. Software architecture in practice. **Diss. Software Engineering Institute**, 2002.

CONCRETE. **Concrete | Kubernetes**. 2018. Disponível em: https://www.concrete.com.br/2018/02/22/tudo-o-que-voce-precisa-saber-sobre-kubernetes/. Acesso em: 13 jul. 2019.

CUERVO, E.; BALASUBRAMANIAN, A.; CHO, D.-k.; WOLMAN, A.; SAROIU, S.; CHANDRA, R.; BAHL, P. Maui: making smartphones last longer with code offload. In: ACM. **Proceedings of the 8th international conference on Mobile systems, applications, and services**. *[S. I.]*, 2010. p. 49–62.

DINH, H. T.; LEE, C.; NIYATO, D.; WANG, P. A survey of mobile cloud computing: architecture, applications, and approaches. **Wireless communications and mobile computing**, Wiley Online Library, v. 13, n. 18, p. 1587–1611, 2013.

DOU, A.; KALOGERAKI, V.; GUNOPULOS, D.; MIELIKAINEN, T.; TUULOS, V. H. Misco: a mapreduce framework for mobile systems. In: ACM. **Proceedings of the 3rd international conference on pervasive technologies related to assistive environments**. *[S. I.]*, 2010. p. 32.

DRAGONI, N.; GIALLORENZO, S.; LAFUENTE, A. L.; MAZZARA, M.; MONTESI, F.; MUSTAFIN, R.; SAFINA, L. Microservices: yesterday, today, and tomorrow. In: **Present and Ulterior Software Engineering**. *[S. I.]*: Springer, 2017. p. 195–216.

DRAGONI, N.; LANESE, I.; LARSEN, S. T.; MAZZARA, M.; MUSTAFIN, R.; SAFINA, L. Microservices: How to make your application scale. **arXiv preprint arXiv:1702.07149**, 2017.

ESPOSITO, C.; CASTIGLIONE, A.; CHOO, K.-K. R. Challenges in delivering software in the cloud as microservices. **IEEE Cloud Computing**, IEEE, v. 3, n. 5, p. 10–14, 2016.

FAN, C.-Y.; MA, S.-P. Migrating monolithic mobile application to microservice architecture: An experiment report. In: IEEE. **AI & Mobile Services (AIMS), 2017 IEEE International Conference on**. *[S. I.]*, 2017. p. 109–112.

FERNANDO, N.; LOKE, S. W.; RAHAYU, W. Mobile cloud computing. **Future Gener. Comput. Syst.**, v. 29, n. 1, p. 84–106, jan. 2013. ISSN 0167-739X.

FISCHER, H. **A history of the central limit theorem: From classical to modern probability theory**. *[S. I.]*: Springer Science & Business Media, 2010.

FOWLER, M. **Patterns of enterprise application architecture**. *[S. I.]*: Addison-Wesley Longman Publishing Co., Inc., 2002.

FOWLER, M. **The Single Responsibility Principle**. 2009. Disponível em: http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle. Acesso em: 3 out. 2019.

FOWLER, M.; LEWIS, J. **Microservices**. 2014. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: 10 jan. 2018.

GIURGIU, I.; RIVA, O.; ALONSO, G. Dynamic software deployment from clouds to mobile devices. In: SPRINGER. **ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing**. *[S. I.]*, 2012. p. 394–414.

GOMES, F. A.; REGO, P. A.; ROCHA, L.; SOUZA, J. N. de; TRINTA, F. Caos: A context acquisition and offloading system. In: IEEE. **2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)**. *[S. I.]*, 2017. v. 1, p. 957–966.

GOMES, F. A.; VIANA, W.; ROCHA, L. S.; TRINTA, F. A contextual data offloading service with privacy support. In: ACM. **Proceedings of the 22nd Brazilian Symposium on Multimedia and the Web**. *[S. I.]*, 2016. p. 23–30.

HAT, R. **O que é Docker? | Red Hat**. 2019. Disponível em: https://www.redhat.com/pt-br/topics/containers/what-is-docker. Acesso em: 13 jul. 2019.

JAMSHIDI, P.; PAHL, C.; MENDONÇA, N. C.; LEWIS, J.; TILKOV, S. Microservices: The journey so far and challenges ahead. **IEEE Software**, IEEE, v. 35, n. 3, p. 24–35, 2018.

JÚNIOR, O. A. Arquitetura de micro serviços: uma comparação com sistemas monolíticos. Universidade Federal da Paraíba, 2017.

JUSTINO, Y. d. L. **Do monolito aos microsserviços: um relato de migração de sistemas legados da Secretaria de Estado da Tributação do Rio Grande do Norte**. Dissertação (Mestrado) – Brasil, 2018.

KALSKE, M. Transforming monolithic architecture towards microservice architecture. 2019.

KEMP, R.; PALMER, N.; KIELMANN, T.; BAL, H. Cuckoo: a computation offloading framework for smartphones. In: SPRINGER. **International Conference on Mobile Computing, Applications, and Services**. *[S. I.]*, 2010. p. 59–79.

KHALIDD, A. Identifying smartphone users based on their activity patterns via mobile sensing. 2017.

KNOCHE, H.; HASSELBRING, W. Using microservices for legacy software modernization. **IEEE Software**, IEEE, v. 35, n. 3, p. 44–49, 2018.

KUMAR, K.; LU, Y.-H. Cloud computing for mobile users: Can offloading computation save energy? **Computer**, IEEE, v. 43, n. 4, p. 51–56, 2010.

LEVCOVITZ, A.; TERRA, R.; VALENTE, M. T. Towards a technique for extracting microservices from monolithic enterprise systems. **arXiv preprint arXiv:1605.03175**, 2016.

LINDVALL, M.; TESORIERO, R.; COSTA, P. Avoiding architectural degeneration: An evaluation process for software architecture. In: IEEE. **Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on**. *[S. I.]*, 2002. p. 77–86.

MARCOS, A. A distributed environment to support cooperative software development. **"HIGH PERFORMANCE NETWORKING, V", Serge Fdida (Ed.), IFIP Transactions C-26**, Elsevier Science BV (North-Holland), p. 375–390, 1994.

MEDVIDOVIC, N.; TAYLOR, R. N. Software architecture: foundations, theory, and practice. In: IEEE. **Software Engineering, 2010 ACM/IEEE 32nd International Conference on**. *[S. I.]*, 2010. v. 2, p. 471–472.

MUSTAFA, O.; GÓMEZ, J. M.; HAMED, M.; PARGMANN, H. Granmicro: A black-box based approach for optimizing microservices based applications. In: **From Science to Society**. *[S. I.]*: Springer, 2018. p. 283–294.

NETFLIX. **Netflix Open Source Software Center**. 2018. Disponível em: https://netflix.github.io/. Acesso em: 2 abr. 2018.

PAHL, C.; JAMSHIDI, P. Microservices: A systematic mapping study. In: SCITEPRESS. **CLOSER (1)**. *[S. I.]*, 2016. p. 137–146.

PAPAZOGLOU, M. Service -oriented computing: Concepts, characteristics and directions. In: . *[S. I.]*: IEEE, 2003.

PARMAR. **Monolithic vs MicroService Architecture**. 2014. Disponível em: https://goo.gl/8ZVivb. Acesso em: 13 abr. 2018.

PFLEEGER, S. **Engenharia de software: teoria e prática**. Prentice Hall, 2004. ISBN 9788587918314. Disponível em: https://books.google.com.br/books?id=MKhmPgAACAAJ.

PRESSMAN, R. **Engenharia de Software - 7.ed.:**. McGraw Hill Brasil, 2009. ISBN 9788580550443. Disponível em: https://books.google.com.br/books?id=y0rH9wuXe68C.

PUTMAN, J. R.; BARRY, W. **Open distribution software architecture using RM-ODP**. *[S. I.]*: Prentice Hall PTR, 2000.

QIAN, H.; ANDRESEN, D. Jade: Reducing energy consumption of android app. **International Journal of Networked and Distributed Computing**, Atlantis press, v. 3, n. 3, p. 150–158, 2015.

RIBEIRO, B. R. C. **Estudo comparativo entre arquiteturas monolíticas e de micro serviços**. Tese (Doutorado), 2017.

RICHARDS, M. **Microservices vs. Service-Oriented Architecture**. *[S. I.]*: O'Reilly Media, Inc., 2016.

RICHARDSON, C. **Pattern: Decompose by subdomain**. 2018. Disponível em: https://microservices.io/patterns/decomposition/decompose-by-subdomain.html. Acesso em: 11 abr. 2018.

SANTOS, G. B. dos; REGO, P. A.; TRINTA, F. Uma proposta de solução para offloading de métodos entre dispositivos móveis. In: SBC. **Anais Estendidos do XXIII Simpósio Brasileiro de Sistemas Multimídia e Web**. *[S. I.]*, 2017. p. 76–81.

SANT'ANNA, C.; FIGUEIREDO, E.; GARCIA, A.; LUCENA, C. On the modularity assessment of software architectures: Do my architectural concerns count. In: AOSD. **Proc. International Workshop on Aspects in Architecture Descriptions (AARCH. 07), AOSD**. *[S. I.]*, 2007. v. 7.

SARKAR, S.; VASHI, G.; ABDULLA, P. Towards transforming an industrial automation system from monolithic to microservices. In: IEEE. **2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)**. *[S. I.]*, 2018. v. 1, p. 1256–1259.

SATYANARAYANAN, M.; BAHL, P.; CACERES, R.; DAVIES, N. The case for VM-based cloudlets in mobile computing. **Pervasive Computing, IEEE**, v. 8, n. 4, p. 14–23, 2009. ISSN 1536-1268.

SHAW, M.; GARLAN, D. *et al.* **Software architecture**. *[S. I.]*: prentice Hall Englewood Cliffs, 1996. v. 101.

SOMMERVILLE, I. **Engenharia de software**. PEARSON BRASIL, 2011. ISBN 9788579361081. Disponível em: https://books.google.com.br/books?id=H4u5ygAACAAJ.

SOUSA, G.; RUDAMETKIN, W.; DUCHIEN, L. Software product lines for multi-cloud microservices-based applications. In: ACM. **6th International Workshop on Cloud Data and Platforms (CloudDP)**. *[S. I.]*, 2016.

SPRING. **Spring Cloud**. 2018. Disponível em: http://projects.spring.io/spring-cloud/. Acesso em: 2 abr. 2018.

SUN, Y.; NANDA, S.; JAEGER, T. Security-as-a-service for microservices-based cloud applications. In: IEEE. **Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on**. *[S. I.]*, 2015. p. 50–57.

TAIBI, D.; LENARDUZZI, V.; PAHL, C. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. **IEEE Cloud Computing**, IEEE, v. 4, n. 5, p. 22–32, 2017.

TAIBI, D.; LENARDUZZI, V.; PAHL, C.; JANES, A. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In: ACM. **Proceedings of the XP2017 Scientific Workshops**. *[S. I.]*, 2017. p. 23.

THÖNES, J. Microservices. **IEEE Software**, IEEE, v. 32, n. 1, p. 116–116, 2015.

TRIEBEL, W. **Itanium architecture for software developers**. *[S. I.]*: Intel Press, 2000.

VALIPOUR, M. H.; AMIRZAFARI, B.; MALEKI, K. N.; DANESHPOUR, N. A brief survey of software architecture concepts and service oriented architecture. In: IEEE. **2009 2nd IEEE International Conference on Computer Science and Information Technology**. *[S. I.]*, 2009. p. 34–38.

VILLAMIZAR, M.; GARCÉS, O.; CASTRO, H.; VERANO, M.; SALAMANCA, L.; CASALLAS, R.; GIL, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: IEEE. **Computing Colombian Conference (10CCC), 2015 10th**. *[S. I.]*, 2015. p. 583–590.

XIA, F.; DING, F.; LI, J.; KONG, X.; YANG, L. T.; MA, J. Phone2cloud: Exploiting computation offloading for energy saving on smartphones in mobile cloud computing. **Information Systems Frontiers**, Springer, v. 16, n. 1, p. 95–111, 2014.

YUGOPUSPITO, P.; PANDUWINATA, F.; SUTRISNO, S. Microservices architecture: case on the migration of reservation-based parking system. In: IEEE. **2017 IEEE 17th International Conference on Communication Technology (ICCT)**. *[S. I.]*, 2017. p. 1827–1831.

ZHAO, B.; XU, Z.; CHI, C.; ZHU, S.; CAO, G. Mirroring smartphones for good: A feasibility study. In: SPRINGER. **International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services**. *[S. I.]*, 2010. p. 26–38.

APÊNDICE A – ENTREVISTA COM ESPECIALISTAS

- A plataforma CAOS é composta por quais componentes?
- Os componentes possuem dependências entre eles? Se sim, quais?
- Quais tecnologias foram utilizadas para construção do CAOS?
- Quais protocolos são utilizados para possibilitar a comunicação entre os dispositivos móveis e a solução CAOS?
- Como é realizado o processo de Offloading na plataforma CAOS? Porque é necessária a utilização de uma máquina virtual nesse processo?
- Para você, quais serviços presentes na solução poderia funcionar de forma isolada e independente?